Accelerating Test Case Reduction

by

Golnaz Gharachorlu

M.Sc., The University of British Columbia, 2015
B.Sc., University of Tehran, 2008

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

© Golnaz Gharachorlu 2023
SIMON FRASER UNIVERSITY
Spring 2023

Declaration of Committee

Name:	**Golnaz Gharachorlu**
Degree:	**Doctor of Philosophy**
Thesis title:	**Accelerating Test Case Reduction**
Committee:	**Chair:** Keval Vora Assistant Professor, Computing Science
	Nick Sumner Supervisor Associate Professor, Computing Science
	Steven Ko Committee Member Associate Professor, Computing Science
	Yuepeng Wang Examiner Assistant Professor, Computing Science
	Reid Holmes External Examiner Associate Professor, Computer Science The University of British Columbia

Abstract

Test cases play an important role in testing and debugging software. Large complex test cases are hard to comprehend and can hinder these tasks. Thus, reducing test cases is crucial for understanding and fixing defects in software. Given a test case with a property of interest, such as demonstrating a bug in the software under test, the goal of *test case reduction* is to create a smaller variant of the test case that still exhibits the bug. To generate this smaller variant, a reducer searches over a set of smaller candidates by applying transformations to the original test case. For each candidate, the reducer queries an oracle to verify whether the desired properties still hold. If they do, the search continues from the new smaller variant. Reducers stop when they see no more reduction opportunities or time out.

Even when automated, test case reduction is slow and time-consuming due to repeated trial and error with smaller candidates. This causes interruptions that adversely affect a developer's productivity. The goal of this dissertation is to accelerate test case reduction by proposing new techniques that address some of the limitations in the field. In particular, we suggest generalized techniques that reduce test cases of various domains by traversing their parse trees and applying reduction to the nodes.

To this end, we improve the theoretical bounds and empirical performance of the well-known Delta Debugging algorithm by converting its quadratic worst case time complexity into linear. We propose novel tree traversal orders that remove more of the test case earlier. We train machine learning models, capable of avoiding candidates that do not adhere to a domain's validity constraints. We further leverage models that guide reduction towards generating candidates that are likely to be valid. These guiding models make their suggestions based on the reduction progress.

Our empirical results on a set of real-world test cases from multiple domains are promising and demonstrate the practical value of our techniques. More specifically, we obtain an average improvement of around 60% in reduction time compared to the state of the art when reducing C, Rust and Go programs.

Keywords: Test Case Reduction; Program Reduction; Machine Learning; Delta Debugging; Debugging; Software Testing

Dedication

To my mother for all her love, support, and kindness and to the memory of my father whom I miss every day.

Acknowledgements

I would like to express my sincere gratitude to my PhD advisor Dr. Nick Sumner for all his support and guidance during my PhD years. I am so grateful to have the opportunity of working with him. Not only did I have the chance to work with someone so knowledgeable in the field but also his patience and expertise allowed me to see my progress day by day.

I would like to deeply thank Dr. Steven Ko for always taking the time to offer me insightful advice and remarks.

A special thanks also goes out to the examiners Dr. Yuepeng Wang and Dr. Reid Holmes, and the chair Dr. Keval Vora, for kindly agreeing to be on my exam committee and providing me with constructive feedback.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Testing and debugging are essential parts of software maintenance [3]. In testing, multiple test cases (inputs) are fed into the software under test (SUT) to examine its behavior. For instance, randomized testing is a process in which a fuzzer stress tests the SUT by generating randomized and potentially large test cases and feeding them into the SUT to look for possible failures [4]. When a test case fails, it indicates a bug in the SUT that needs to be fixed. A person debugging the software should then identify the defects in the software program that are the root causes of the observed failure in order to fix the bug.

The success and efficiency of the debugging process strongly depends on the test case [5]. A very large test case with many parts that are not relevant to the failure is hard to understand and makes debugging a tedious task. In fact, in many software issue trackers, only bugs that have a minimal reproducible test case are investigated [6]. As a result, removing the irrelevant portions of a test case is of great importance.

The goal behind reducing a test case is to obtain a smaller and simpler test case from the possibly large original test case such that the smaller test case still exhibits the property of interest such as a bug or failure observed by the original test case. The smaller test case with potentially fewer dependencies is typically more easily reproducible and can make debugging easier. In addition, it can help avoid copyright issues the large test cases may have in their publication and thus can be added to the test suite of the software under test [7]. Finally, a bug report with a smaller test case is more general, meaning that it is likely to subsume other bug reports with test cases that only differ in irrelevant details [1].

Figure 1.1 illustrates the process of testing and debugging software with initial and reduced test cases. The software under test is a program written in the Python programming language, and the test cases are the set of `Food` objects that are fed into the program as inputs. The failure is an assertion error on line 25. In order to fix the bug, the causes of the failure should be correctly identified. To this end, statements covered and executed by the input test case are considered as potential causes. These statements, highlighted in yellow, can help the person debugging the program to narrow down her search space. As can be seen, the initial test case in Figure 1.1 (a) with `Food` objects that are irrelevant to the

```
from dataclasses import dataclass

@dataclass
class Food:
    group: str
    calorie: float
    name: str

# initial test case
foods = [ Food("vegetable", 20, "celery"), Food("main", 200, "rice"),
Food("fruit", 80, "apple"), Food("dessert", 300, "ice cream"),
Food("vegetable", 30, "lettuce"), Food("main", 250, "bread"),
Food("fruit", 100, "banana") , Food("dessert", 400, "cake")]


def checkCalorieVeg(food):
    if food.name == "celery":
        assert (food.calorie == 20)
    if food.name == "lettuce":
        assert (food.calorie == 30)
def checkCalorieDessert(food):
    if food.name == "ice cream":
        assert(food.calorie == 300)
    if food.name == "cake":
        assert(food.calorie == 350)   failure
def checkCalorieMain(food):
    if food.name == "rice":
        assert(food.calorie == 200)
    if food.name == "bread":
        assert(food.calorie == 250)
def checkCalorieFruit(food):
    if food.name == "apple":
        assert(food.calorie == 80)
    if food.name == "banana":
        assert(food.calorie == 100)
for food in foods:
  if food.group == "vegetable":
        checkCalorieVeg(food)
    if food.group == "dessert":
        checkCalorieDessert(food)
    if food.group == "main":
        checkCalorieMain(food)
    if food.group == "fruit":
        checkCalorieFruit(food)

```

(a) Initial test case

```
from dataclasses import dataclass

@dataclass
class Food:
    group: str
    calorie: float
    name: str

# reduced test case
foods = [ Food("dessert", 400, "cake") ]

def checkCalorieVeg(food):
    if food.name == "celery":
        assert (food.calorie == 20)
    if food.name == "lettuce":
        assert (food.calorie == 30)
def checkCalorieDessert(food):
    if food.name == "ice cream":
        assert(food.calorie == 300)
    if food.name == "cake":
        assert(food.calorie == 350)   failure
def checkCalorieMain(food):
    if food.name == "rice":
        assert(food.calorie == 200)
    if food.name == "bread":
        assert(food.calorie == 250)
def checkCalorieFruit(food):
    if food.name == "apple":
        assert(food.calorie == 80)
    if food.name == "banana":
        assert(food.calorie == 100)
for food in foods:
    if food.group == "vegetable":
        checkCalorieVeg(food)
    if food.group == "dessert":
        checkCalorieDessert(food)
    if food.group == "main":
        checkCalorieMain(food)
    if food.group == "fruit":
        checkCalorieFruit(food)
```

(b) Reduced test case

Figure 1.1: Testing and debugging a Python program. Executed program lines are highlighted in yellow and the input test case is highlighted in blue.

failure executes more statements of the program, making the debugging task more difficult. In contrast, the smaller test case in Figure 1.1 (b) distills the statements that contribute to the failure, yielding a smaller search space to identify and fix the bug.

In the above example, we described a scenario in which the test case is a set of input objects and the software under test (SUT) is a Python program. However, there are a wide variety of test cases and systems under test. A test case can be a program itself when testing a compiler [4] or it can be a sequence of user interactions with a web application [8] or a sequence of queries for a database [9, 10]. Thread schedules can be test cases for concurrent programs [11] and GUI event traces in Android applications [12] are another example of test cases.

Manual reduction of test cases is tedious and time-consuming if not infeasible. For example, manually reducing a test case that triggers a non-deterministic bug in a multi-threaded application is not possible [11, 13]. As a result, automated test case reduction has drawn much attention during the past decades [1, 14, 15, 13, 16, 17, 18, 19, 20, 21].

To automatically find a smaller variant of the test case that still preserves the property of interest (e.g., it still exhibits the failure in the software), the reduction algorithm explores a search space of candidates by applying a set of transformations to the original test case and generating smaller variants. The reduction transformations and the way they are applied are defined individually by each algorithm. *Removing* parts of the test case or *replacing* larger portions with smaller ones are the main reduction operations performed by the existing techniques [1, 13, 15, 22, 10]. When a candidate is generated by applying a transformation, the reduction algorithm verifies whether it preserves the property of interest by querying an oracle. If the reduced variant preserves the property of interest (also referred to as a successful variant), it replaces the original test case, and the search and reduction process continue on the reduced variant. Reduction is an *anytime* algorithm, meaning that it can stop at any time and return the smallest variant obtained so far. In practice, reducers stop when they (1) see no more reduction opportunities or (2) time out [1, 15, 13].

Often, the problem of finding a smaller test case with the property of interest is approached through Delta Debugging (DD) [1], a longstanding and effective algorithm for test case reduction that essentially generalizes binary search. However, for inputs with significant structure, generic Delta Debugging can perform poorly, requiring significant time and not performing much reduction [14, 23]. For compilers in particular, where the test cases must be valid programs, this has led to specialized techniques like Hierarchical Delta Debugging (HDD) [14, 23], domain specific reducers like C-Reduce [13], and most recently to syntax guided domain agnostic reducers as seen in Perses [15]. Despite these advancements, test case reduction may still take hours to run, hindering productivity and scalability. More specifically, reduction processes that take hours instead of minutes can disrupt developer workflow and reduce efficiency [24]. Moreover, they can limit the scalability of emerging uses for test case reduction [25, 26]. Thus, accelerating test case reduction turns into a significant and intriguing study area.

The goal of this dissertation is to address some of the problems in this domain and propose solutions to improve the *efficiency* of test case reduction while preserving its *effectiveness*. A more efficient reduction can speed up the debugging process by reducing the number of long disruptions a developer may encounter during a day. This can help developers to maintain their productivity by not switching tasks [27].

We measure the efficiency by using two metrics that are widely used in the state of the art techniques [1, 14, 13, 15]:

1. Total test case reduction *time* (wall-clock time).

2. The total number of reduced variants or candidates generated during reduction (both successful and unsuccessful ones). Since a verifying test by an oracle needs to be performed on each reduced variant to check whether it preserves the property of interest, we simply refer to this metric as the number of *tests*, *oracle queries* or *oracle calls*.

In general, a reduction technique that performs fewer tests within shorter reduction time is more efficient.

Also similar to previous works [1, 14, 13, 15], we define the *size* of the final reduced test case as the metric to measure effectiveness. A smaller size indicates a higher effectiveness and reduction power for the technique. The size can be represented in terms of the number of elements that comprise a test case. For instance, a C program used as a test case for a compiler consists of tokens that are the smallest meaningful units in a program. Hence, the total number of tokens remaining in the final reduced program can be used as a metric to measure the effectiveness of the program reducer.

By monitoring the reduction time, we identify the bottlenecks of existing techniques and propose novel methods that speed up reduction while generating reduced test cases that are comparable in size to those generated by the state of the art.

Additionally, this dissertation seeks to offer solutions that are not specifically tailored to operate in one particular domain. These *generalized* techniques also referred to as *domain agnostic* apply reduction operations on the nodes of the parse tree or abstract syntax tree (AST) of the given test case. As a result, they are reusable on multiple domains with little or no extra effort, they do not require extensive knowledge of the domain, and can easily be applied when resources are limited.

With the goal of accelerating generalized test case reduction techniques, we pursue the following five main directions:

(1) We propose and evaluate a value guided reduction technique called One Pass Delta Debugging that skips performing tests with low likelihood of success in practice [2]. By slightly modifying the original Delta Debugging algorithm, we reduce the number of tests in the worst case behavior of Delta Debugging from $O(n^2)$ to just $O(n)$. This enhancement can be beneficial to any reducer whose infrastructure is based on Delta Debugging [14, 15, 13]. We originally introduced and published this idea at the 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS 2018) [2].

(2) We propose and evaluate a priority aware test case reduction technique called PARDIS [28]. Its concept of priority awareness originates from Perses [15], the latest state of the art domain agnostic program reducer, in which nodes in the parse tree of the test case are ranked in a queue in a descending order of their size to be targeted by the reducer. However, we demonstrate that Perses targets the prioritized nodes and tries to reduce them in an inefficient manner. PARDIS and its variant PARDIS HYBRID propose different methods of prioritizing nodes in the queue that can help to remove larger portions of the test case earlier, leading to a faster convergence towards the minimal test case. We originally introduced and published this idea at the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE 2019) [28].

(3) MODEL GUIDED PARDIS [29] is another variation of PARDIS proposed and evaluated in this dissertation. This new technique is built upon PARDIS but it employs models trained by machine learning algorithms to enable PARDIS to predict and avoid conducting an invalid reduction operation. In particular, we are interested in predicting and filtering *semantically invalid tests*. These tests violate semantic constraints of the test case domain and have drawn little or no attention among generalized reducers. For instance, removing a function definition before removing its call site is a semantically invalid test performed by many domain agnostic reducers, including Perses and original PARDIS. A reducer with too many semantically invalid candidates in its search space has to spend a lot of time running these tests without seeing any progress in reduction. We originally introduced and published this idea at the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR 2021) [29].

(4) Next, we propose TYPE BATCHED REDUCER [30] to further improve the performance of domain agnostic reduction techniques. In particular, we note that current domain agnostic reducers such as Perses, PARDIS and its variants traverse the parse tree in orders that can hinder successful reduction. These reducers traverse the tree from the top down to visit nodes with a larger number of descendants earlier. However, removing such nodes can often fail due to dependencies within the test case. To mitigate this problem, TYPE BATCHED REDUCER partitions the nodes of the tree into *batches* and uses machine learning to select the batches of nodes that are estimated to be the most effective at reducing the test case. In other words, by selecting the most effective nodes to reduce at a given point in time during reduction, the reducer is guided towards portions that are more likely to be successfully reduced. Unlike traditional traversal based techniques such as Hierarchical Delta Debugging [14], Perses and PARDIS, the traversal orders suggested by TYPE BATCHED REDUCER enable visiting nodes at different levels and locations within the tree. We originally introduced and published this idea at the 32nd ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023) [30].

(5) Finally, selecting the most advantageous nodes by Type Batched Reducer increases the probability of successfully removing them. As a result, we propose and apply a Probabilistic Joint Reduction technique [30] to simultaneously reduce over multiple portions of the tree at once. This idea was also originally introduced and published at the 32nd ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023) [30].

By exploring the above directions, we aim to significantly improve the speed of test case reduction through a combination of 1) skipping likely uninformative tests, 2) enabling early removal of high impact test case regions, 3) avoiding invalid tests, 4) prioritizing tests that are more likely to be valid, and 5) learning which portions of the test case can get removed together without making the property of interest disappear.

Our results are promising, showing an average improvement of around 60% in reduction time when reducing real-world test cases from C, Rust, and Go domains without harming the reduction power.

The rest of the dissertation is organized as follows:

In the next chapter, we provide background information on test case reduction, including formal definitions and fundamentals of some existing techniques. Chapter 3 presents our One Pass Delta Debugging algorithm, its evaluation and the conditions under which it could function effectively. Chapter 4 describes Pardis and its variant Pardis Hybrid, our two priority aware test case reduction techniques and compares them against Perses [15], the latest state of the art domain agnostic reducer. The integration of machine learning and test case reduction is the subject of Chapter 5. To decrease the number of invalid tests run by current reducers, we propose two sets of models in this chapter. The first set of models integrated into Pardis (also referred to as Model Guided Pardis) can help the reducer to predict and avoid performing invalid tests. The second set of models introduced in Type Batched Reducer suggests the most advantageous portions of the test case to reduce at a given point in time during reduction. In Chapter 6, we investigate some other aspects of test case reduction such as comparing our best domain agnostic reducer with C-Reduce [13], the powerful and effective domain specific C reducer and further discuss the threats to validity of the results and potential future directions. The dissertation closes with related work and conclusions.

Chapter 2

Background and Motivation

In this chapter, we first provide some formal definitions and terms in Section 2.1 that are shared among existing test case reduction techniques and the novel techniques proposed in this dissertation. Next, in Section 2.2, we describe Delta Debugging (DD) [1], a longstanding approach to automated test case reduction that was proposed more than two decades ago by Zeller and Hildebrandt. The generic process of Delta Debugging in searching for smaller inputs with a property of interest has inspired various techniques to either improve [14, 23, 31, 19, 2, 32] or apply Delta Debugging in their intended applications [33, 15, 13, 8, 11, 12]. We present some limitations of Delta Debugging and study one of its improvements called Hierarchical Delta Debugging (HDD) [14] in Section 2.3 that is more suitable for reducing test cases with structure such as program source code. Despite the improvements by HDD, it still suffers from some drawbacks also discussed in this section. In Section 2.4, we elaborate on reducing test cases in form of program source code referred to as *program reduction* and describe Perses [15], the latest state of the art domain agnostic program reducer. Perses is a syntax based queue driven reducer proposed by Sun et al. that is mostly known as a program reducer in the literature. However, it is capable of reducing test cases other than programs from domains with structure such as XML files. Despite its capability and generality, Perses has also limitations that are not addressed in the literature. In Section 2.5, we discuss some of these limitations and motivate solutions for them. Similar to Perses, our solutions are generalized and domain agnostic and can be used to reduce any test cases that have structure, including programs and XML files.

2.1 Preliminaries

The aim of *test case reduction* is to reduce a test case with some property of interest such that the reduced *variant* still preserves the property. Often, the property of interest is that the test case exhibits a particular failure or bug. From here on, we refer to a test case satisfying a particular property as *inducing the failure* without loss of generality. By applying a set of *transformations* on the *elements* of the given test case, the reduction technique searches for smaller candidate variants and verifies whether they still induce the failure. To this end, an *oracle* is *queried* on each variant to perform the verification. A variant that still triggers the failure replaces the original test case and the reduction continues on the smaller new variant. Test case reduction is an *anytime* algorithm, meaning that it can stop at any time and return the smallest variant generated so far as the reduced test case. In practice, reducers terminate when there are no more candidates available in their *search space* or within a timeout. The returned reduced test case is normally *1-minimal.*

In the following, we provide formal definitions for the terms used above:

Definition 2.1.1. Given τ as a test case with $|\tau| = n$, $e_1, ..., e_n$ are elements of τ if $e_1, ..., e_n \in \tau$ and $e_1 \bigoplus e_2 \bigoplus ... \bigoplus e_n = \tau$ and e_is are pairwise disjoint and $|e_i| = 1$, $\forall e_i \in \tau$.

An element is the smallest unit of a test case that can be customized based on the domain of the test case. For instance, an element of a sequence of user actions in a web browser can be defined as a single action. An element of a C program as a test case for a C compiler can be a single character, a token that is the smallest meaningful unit of a program, a line of code or a node in the parse tree or abstract syntax tree (AST) of the program.

Definition 2.1.2. Given τ as a test case, $\tau\prime$ is a configuration or variant of τ if $\tau\prime \subseteq \tau$.

Note that $\bigoplus$ in Definition 2.1.1 preserves the order of elements. Thus, a variant of a test case is a subset of the test case in which the elements of the subset preserve the same order as the elements in the original test case.

Definition 2.1.3. Given reducer R and test case τ, the set of transformations $\mathbb{M}$ is the set of reduction operations applied on elements of τ by R.

Reduction operations are customized based on the implementation of the reducer and the domain of the test case. In general, *removing* elements from the test case or *replacing* them with smaller elements are the main reduction operations performed by existing reducers [1, 14, 15, 22, 10]. Applying a transformation $m \in M$ on test case τ generates a candidate variant $\tau\prime \subseteq \tau$ to be verified by the oracle. The search space of the reducer consists of all variants or configurations generated by the reducer.

Definition 2.1.4. Given a test case τ and a reducer with search space $\mathbb{S}$, an oracle function is the Boolean function $\psi : \mathbb{S} \rightarrow \{True, False\}$ such that $\psi(\tau\prime) = True$ if variant $\tau\prime \in \mathbb{S}$ triggers the failure and $\psi(\tau\prime) = False$ otherwise. By definition, $\psi(\tau) = True$ when τ is the initial failure-inducing test case and $\psi(\emptyset) = False$.

The verification process performed by an oracle is referred to as an *oracle query*, an *oracle call* or simply a *test.*

Definition 2.1.5. Given τ as an original test case with $\psi(\tau) = True$, the goal of test case reduction is to find a minimal variant τ_{min} from the search space $\mathbb{S}$ such that $\tau_{min} = argmin_{\tau\prime \in \mathbb{S}}\ f(\tau\prime)$ and $\psi(\tau_{min}) = True$ where $f(\tau\prime)$ is a cost function with respect to a metric such as $|\tau\prime|$.

Finally, the reduced test case generated by a reducer is not a *globally* minimum test case because obtaining such a test case from an infinite set of potential transformations $\mathbb{M}$ is exponential [15]. Moreover, a reducer applies transformations from $\mathbb{M}$ one by one that makes the reduced test case not be *locally* minimum either. To mitigate the problem, a more relaxed version of minimality called 1-minimality is introduced.

Definition 2.1.6. A test case τ is 1-minimal if $\psi(\tau) = True$ and $\forall e_i \in \tau$, $\psi(\tau - e_i) = False$ where $i = 1, 2, ..., |\tau|$.

This means that every atomic element in a 1-minimal test case is required to trigger the failure. In other words, if any one single element is not present in the test case, the failure will disappear. In practice, 1-minimal test cases are small enough to successfully provide insights on a failure [1, 14, 15, 34, 35].

2.2 Delta Debugging

Given a failure-inducing test case, the well-established test case reduction technique, Delta Debugging explores smaller test case variants based on a greedy search algorithm that is similar to binary search [1].

Consider the example in Figure 2.1. The initial failing test case in this example is the set of numbers $\tau_{\pmb{\chi}}$=`{1, 2, 3, 4, 5, 6, 7, 8}`. Suppose that the failure-inducing part of the test case is the set `{1, 7, 8}` such that a test case variant exhibits the failure if and only if `{1, 7, 8}` is a subset of the variant. Delta Debugging first partitions the test case into halves, generating two subsets Δ_1 =`{1, 2, 3, 4}` and Δ_2 =`{5, 6, 7, 8}`. It then passes these subsets to the oracle function to verify whether either of them can trigger the failure:

1. if $\psi(\Delta_1) = True$, then the test case $\tau_{\pmb{\chi}}$ is reduced to Δ_1.
2. else if $\psi(\Delta_2) = True$, then the test case $\tau_{\pmb{\chi}}$ is reduced to Δ_2.
3. else if $\psi(\Delta_1) = False$ and $\psi(\Delta_2) = False$, granularity is increased.

If the first or second branch is true, $\tau_{\pmb{\chi}}$ *reduces to subset* Δ_1 or Δ_2, respectively and Delta Debugging will continue recursively on the new test case. However, if the third branch is executed, none of the subsets on their own can trigger the failure. Hence, Delta Debugging *increases granularity* by partitioning each subset into halves, so instead of having two subsets,

Step		Test Case (τ')								$\psi(\tau')$	Description
0	$\tau_{✗}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**True**	**initial test case**
1	$\Delta_1 = \nabla_2$	1	2	3	4	.	.	.	.	False	test subset
2	$\Delta_2 = \nabla_1$	.	.	.	.	5	6	7	8	False	test subset
3	Δ_1	1	2	.	.	.	.	.	.	False	granularity increased test subset
4	Δ_2	.	.	3	4	.	.	.	.	False	test subset
5	Δ_3	.	.	.	.	5	6	.	.	False	test subset
6	Δ_4	.	.	.	.	.	.	7	8	False	test subset
7	∇_1	.	.	3	4	5	6	7	8	False	test complement
8	∇_2	**1**	**2**	**.**	**.**	**5**	**6**	**7**	**8**	**True**	**reduce to complement**
9	Δ_1	1	2	.	.	.	.	.	.	False	continued with same granularity test subset[1]
10	Δ_2	.	.	.	.	5	6	.	.	False	test subset[1]
11	Δ_3	.	.	.	.	.	.	7	8	False	test subset[1]
12	∇_1	.	.	.	.	5	6	7	8	False	test complement
13	∇_2	**1**	**2**	**.**	**.**	**.**	**.**	**7**	**8**	**True**	**reduce to complement**
14	$\Delta_1 = \nabla_2$	1	2	.	.	.	.	.	.	False	continued with same granularity test subset[1]
15	$\Delta_2 = \nabla_1$	.	.	.	.	.	.	7	8	False	test subset[1]
16	Δ_1	1	.	.	.	.	.	.	.	False	granularity increased test subset
17	Δ_2	.	2	.	.	.	.	.	.	False	test subset
18	Δ_3	.	.	.	.	.	.	7	.	False	test subset
19	Δ_4	.	.	.	.	.	.	.	8	False	test subset
20	∇_1	.	2	.	.	.	.	7	8	False	test complement
21	∇_2	**1**	**.**	**.**	**.**	**.**	**.**	**7**	**8**	**True**	**reduce to complement**
22	Δ_1	1	.	.	.	.	.	.	.	False	continued with same granularity test subset[1]
23	Δ_2	.	.	.	.	.	.	7	.	False	test subset[1]
24	Δ_3	.	.	.	.	.	.	.	8	False	test subset[1]
25	∇_1	.	.	.	.	.	.	7	8	False	test complement[1]
26	∇_2	1	.	.	.	.	.	.	8	False	test complement
27	∇_3	1	.	.	.	.	.	7	.	False	test complement
28	**Result**	**1**	**.**	**.**	**.**	**.**	**.**	**7**	**8**	**True**	**Done**

[1]can be retrieved from a cache.

Figure 2.1: An example of the Delta Debugging algorithm, reducing test case `{1, 2, 3, 4, 5, 6, 7, 8}` with the failure-inducing portion to be subset `{1, 7, 8}`.

Δ_1 and Δ_2, we will have four subsets, two generated by dividing Δ_1 into halves and two generated by dividing Δ_2. In our example, $\psi(\{1, 2, 3, 4\}) = False$ and $\psi(\{5, 6, 7, 8\}) = False$. Hence, Delta Debugging increases granularity by partitioning `{1, 2, 3, 4}` into `{1, 2}` and `{3, 4}` and `{5, 6, 7, 8}` into `{5, 6}` and `{7, 8}`. Each of these subsets is passed to the oracle to verify whether any of them can reproduce the failure on their own (steps 3-6 in Figure 2.1). If none of these subsets can trigger the failure which is the case in our example, Delta Debugging considers *reducing to the complements* of these subsets. A complement of subset Δ_i, $i = 1, ..., n$ where n is the number of subsets at the current granularity, is defined as $\nabla_i = \tau_{✗} - \Delta_i$. If the test case can be successfully reduced to a complement (such as complement `{1, 2, 5, 6, 7, 8}` in step 8 of our example), Delta Debugging will continue by performing subset and complement tests at the same granularity (steps 9-13) until either performing another successful reduction (step 13) or increasing granularity again (step 16).

Note: Reducing to complement ∇_i is the same as successfully removing subset Δ_i from the test case where $i = 1, ..., n$ and n is the number of subsets (or complements) of the test case at the given granularity.

Delta Debugging terminates when the test case cannot be further partitioned (i.e., until reaching subsets of the smallest possible size). This size is the finest granularity of the reduction and varies for different problems and domains. It can be customized by the user. For instance, a user may want to reduce a file of words by its characters. In this case, the granularity is at the character level and the smallest subset has only one character. Another user may want to reduce the same file by words or even sentences. In this case, each word (or sentence) will be considered as one atomic element that cannot be partitioned further and the reduction will be performed on the list of words (or sentences) until reaching subsets with only one word (or one sentence). In our example of Figure 2.1, the smallest possible subset contains a single number.

Now, we give a more general representation of the Delta Debugging algorithm steps described above:

Let n be the number of subsets of the failure-inducing test case $\tau_{\text{✗}}$. The Delta Debugging algorithm will execute one of the following four branches:

1. *Reduce to subset:* Test each subset Δ_i, $i = 1, 2, ..., n$. If any of the subsets Δ_i triggers the failure, replace $\tau_{\text{✗}}$ with Δ_i. Otherwise:
2. *Reduce to complement:* Test each complement $\nabla_i = \tau_{\text{✗}} - \Delta_i$, $i = 1, 2, ..., n$. If any of the complements ∇_i triggers the failure, replace $\tau_{\text{✗}}$ with ∇_i and continue reducing ∇_i with $n - 1$ subsets. Otherwise:
3. *Increase granularity:* Try steps one and two again with $2n$ subsets. If $2n > |\tau_{\text{✗}}|$, try $|\tau_{\text{✗}}|$ subsets.
4. *Done:* Stop the algorithm if granularity can no longer be increased (i.e., we have reached the finest granularity). Return the reduced test case.

Figure 2.2 presents the formal definition of the three main actions (test subset, test complement, increase granularity) of the Delta Debugging algorithm *ddmin*.

Given the test case $\tau_{\text{✗}}$ to reduce, Delta Debugging partitions the test case into two subsets using $ddmin_2(\tau_{\text{✗}}, 2)$ which performs different types of reduction tests in sequence. First, subset tests are performed and if a failure-inducing subset is found (line 1 of $ddmin_2$), $\tau_{\text{✗}}$ is replaced with the subset and $ddmin_2$ is recursively called on the new test case. Line 2 of $ddmin_2$ performs complement tests while line 3 increases granularity when all complements have been tested. The algorithm terminates when the test case cannot be further partitioned (i.e. the smallest possible subsets have been tested along with their complements).

input : $\tau_{✗}$ and ψ with $\psi(\tau_{✗}) = True$
output : $\tau_{✗}'$ such that $\tau_{✗}' \subseteq \tau_{✗}$ and $\psi(\tau_{✗}') = True$
$ddmin(\tau_{✗}) = ddmin_2(\tau_{✗}, 2)$ where:

$$ddmin_2(\tau_{✗}', n) = \left\{ \begin{array}{ll} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in 1, 2, ..., n \ s.t. \ \psi(\Delta_i) = True \\ ddmin_2(\nabla_i, max(n-1, 2)) & \text{else if } \exists i \in 1, 2, ..., n \ s.t. \ \psi(\nabla_i) = True \\ ddmin_2(\tau_{✗}', min(|\tau_{✗}'|, 2n)) & \text{else if } n < |\tau_{✗}'| \text{ (Increase granularity)} \\ \tau_{✗}' & \text{else Done} \end{array} \right\}$$

note : Δ_is are pairwise disjoint and $|\Delta_i| \approx |\tau_{✗}'|/n$

Figure 2.2: The Delta Debugging algorithm [1].

Subset vs. Complement Tests. The intuition behind testing both subsets and complements in the Delta Debugging algorithm is to increase both *efficiency* and *effectiveness* in reducing test cases. Efficiency is the speed of the reduction while effectiveness is the ability to perform more reduction. Subset tests are greedy attempts performed by Delta Debugging to enable a potentially successful large reduction by a single step. However, since subsets are smaller than complements, they are less likely to contain the failure-inducing portion. In contrast, complement variants are larger and more likely to reproduce the failure. However, if a complement test succeeds, it will prune a smaller part of the test case compared to a successful reduction to subset. To summarize, subset tests are less likely to succeed but if they do, they will remove a large portion of the test case efficiently. On the other hand, complement tests are more likely to succeed but if they do, they will remove a smaller portion of the test case compared to a successful subset test, making additional tests required for more pruning.

Time Complexity. The best case time complexity of Delta Debugging is logarithmic in the size of the test case being reduced. This size is the same as the number of elements within the test case. The best case scenario occurs when every single partition considered by Delta Debugging triggers the failure. In contrast, the worst case time complexity of Delta Debugging is quadratic. This quadratic behavior arises from a concept called *revisiting* in which subsets with previously unsuccessful removal from a test case variant will be revisited to verify whether they can successfully get removed from a *similar* variant. In Figure 2.1 for instance, step 7 of the algorithm performs an unsuccessful removal trial of subset `{1, 2}` from variant `{1, 2, 3, 4, 5, 6, 7, 8}`. When step 8 successfully removes `{3, 4}`, the algorithm revisits subset `{1, 2}` in step 12 to try removing it from the new variant `{1, 2, 5, 6, 7, 8}` which is again unsuccessful. In Chapter 3, we will discuss the process of revisiting subset removals in detail and propose a version of Delta Debugging called One Pass Delta Debugging with linear worst case time complexity that avoids performing revisits.

2.2.1 Limitations of Delta Debugging

Delta Debugging is a longstanding reduction algorithm that is widely available and easily applied due to its generality. It is in particular useful for regression testing since it can pinpoint the changes that cause a regression test to fail [34]. However, this algorithm was proposed more than two decades ago to provide a name and a vocabulary for discussing test case reduction. It suffers from drawbacks especially when it comes to reducing test cases with structure, such as programs. Here, we discuss some of the limitations of the Delta Debugging algorithm.

Unresolved Tests in Structured Domains

So far, we have presented the oracle function outcome as $True$ or $False$ based on observing the failure in the reduced variant or not observing it, respectively. However, not every variant of a test case is valid with respect to the test case domain. These variants will make the oracle outcome *indeterminate* or *unresolved*. Since Delta Debugging does not leverage any domain based knowledge when reducing a test case, it can drastically increase the number of unresolved tests when applied on domains with strict requirements. Programs are one example of this type of domain.

Consider a C program as a test case shown in Listing 2.1 with a run time exception caused by division by zero. This program contains multiple functions. To minimize it by Delta Debugging, we need to define a granularity level. For instance, we can decide to apply reduction on lines of code such that each line of code is considered as one element and the finest granularity consists of subsets that contain a single line of code.

To reduce this test case while defining lines of code as elements, Delta Debugging starts with partitioning the test case into halves, generating two smaller programs that are both invalid. The first half from line 1 to 10 includes a part that is an incomplete function definition (`void bar()`). The second half from line 11 to 20 is also an invalid program due to calling functions that are not defined. Delta Debugging will continue recursively on smaller portions until it reaches the portions that contain a single line of code that are again mostly invalid programs. For example, a line may contain a single curly brace or an incomplete function signature.

Although we can apply Delta Debugging in these cases, it is very unlikely to be able to reduce the original program to a smaller version efficiently. The reason is that lines of code are not good elements to define for program reduction. A function that spans over multiple lines as shown in our example makes Delta Debugging generate subsets of incomplete programs that cannot be compiled.

To mitigate this problem, Misherghi and Su proposed Hierarchical Delta Debugging (HDD) [14] that is more suitable for reducing structured test cases such as programs. Unlike Delta Debugging that does not utilize any domain knowledge, HDD leverages the structure

```
#include <stdio.h>
int foo(int a, int b)
{
  return a/b;
}
void irrelevant()
{
  printf("This function is irrelevant\n");
}
void bar()
{
  int division = foo(2,0);
  printf("%d", division);
}
int main()
{
  irrelevant();
  bar();
  return 0;
}
```

Listing 2.1: A line of code is not a good atomic element when reducing a program.

of the test case to speed up reduction and effectively prune large portions of the test case at an early stage. In Section 2.3, we fully describe the details of the HDD algorithm, its advantage over Delta Debugging for structured test cases, its limitations and improvements.

Unnecessary Tests

Delta Debugging performs some tests that are not required to preserve the 1-minimality of its reduced result. By looking more closely at the algorithm, it is realized that the entire *reduce to subset* step of the algorithm is not required to preserve 1-minimality [31]. In fact, as explained earlier in this section, this step is a greedy attempt by the algorithm to achieve a significant reduction by performing a single test. However, since these subsets are usually small, they are less likely to contain the failure-inducing portions. Thus, it is unlikely for them to successfully trigger the failure in practice. In addition, if a subset can trigger the failure, it will be eventually found by the algorithm when testing complements because irrelevant subsets will be removed by complement tests, leaving the only subset that triggers the failure in the test case.

The revisiting process briefly described at the beginning of this section is another form of potentially unnecessary tests by which Delta Debugging tries to remove subsets that it had previously been unsuccessful in removing them from a similar configuration. Skipping these tests that are unlikely to succeed can benefit the performance of Delta Debugging under certain circumstances [2, 18]. Chapter 3 thoroughly examines this possibility.

2.3 Hierarchical Delta Debugging

To address the problem of unresolved tests generated by Delta Debugging on structured domains as discussed in Section 2.2.1, Misherghi and Su proposed Hierarchical Delta Debugging (HDD) [14]. Here, we describe how HDD can improve Delta Debugging. Moreover, we provide insights on the limitations of HDD itself and discuss some works on its improvement.

2.3.1 HDD: A Better Reducer for Structured Domains

Hierarchical Delta Debugging (HDD) is the main major improvement on Delta Debugging for domains with structured test cases such as programs, XML and JSON files, video frames and any other input with nested and structured data. The reason behind proposing HDD is that partitioning a test case at arbitrary points (i.e., how Delta Debugging works) on a structured domain is far from an effective approach because of the large number of invalid test case variants generated during the reduction.

Using the structure of the test cases, HDD defines domain specific boundaries and groups relevant portions of the test case into one entity to generate fewer invalid test case variants. More precisely, HDD applies the original Delta Debugging algorithm (*ddmin*) on levels of the parse tree or abstract syntax tree (AST) of the failure-inducing test case. The tree is either provided or can be constructed by directly consuming a grammar. It is generated only once and will be modified during the reduction of the test case.

Starting from the top-most level that is the coarsest level of the tree and then proceeding with the finer levels, HDD applies Delta Debugging on each level, considering the tree nodes present at a level as a list of elements to reduce. To this end, Delta Debugging partitions the list of nodes into halves and performs subset tests, complement tests and increases granularity when required. You can refer to Section 2.2 for a complete description of the steps in the Delta Debugging algorithm. Performing these tests at each level generates different variants of the tree. A test case is retrieved (unparsed) from each variant and passed to an oracle function to determine and eliminate the nodes that are not relevant to the failure. Once Delta Debugging is done on one level, it returns a list of remaining nodes at that level that have not been removed. HDD then moves one level down in the parse tree to apply Delta Debugging on the children of the remaining nodes. The process continues until the finest level of the tree is reached.

Algorithm 1 presents the formal definition of the Hierarchical Delta Debugging algorithm.

Algorithm 1: The Hierarchical Delta Debugging (HDD) algorithm [14].

```
Input: $tree$ – The parse tree or AST of test case $\tau_{\text{✗}}$
Input: $\psi$ – Oracle for the property to preserve with $\psi(\tau_{\text{✗}}) = True$
Result: A minimum test case $\tau_{\text{✗}}' \subseteq \tau_{\text{✗}}$ s.t. $\psi(\tau_{\text{✗}}') = True$
1 $level \leftarrow 0$;
2 $nodes \leftarrow getNodes(tree, level)$;
3 while $nodes \neq \emptyset$ do
4     $minconfig \leftarrow ddmin(nodes, \psi)$;
5     $prune(tree, level, minconfig)$;
6     $level \leftarrow level + 1$;
7     $nodes \leftarrow getNodes(tree, level)$;
8 $\tau_{\text{✗}}' \leftarrow unparse(tree)$;
9 return $\tau_{\text{✗}}'$;
```

Starting from the coarsest level of the tree, HDD collects nodes present at that level (line 2) and applies Delta Debugging on them (line 4). On line 5, HDD removes deleted nodes from the list of nodes and moves one level down (line 6) to collect nodes at that level (line 7). The process terminates when there are no nodes available at the level of consideration. In other words, when HDD reaches and applies Delta Debugging on a level that is the finest level with nodes that are all leaves with no children.

In general, traversing the tree from the top down can enable HDD to visit and possibly remove nodes with larger portions of the test case earlier. For instance, if the test case is a C program, function definitions within the program will be at a coarse level of the tree, each represented by a single node. HDD will visit these nodes *before* their descendants that can be statements and local declarations within those function. As a result, HDD algorithm provides faster reduction for structured test cases compared to Delta Debugging in practice [14].

Time Complexity. In theory, HDD has the worst case time complexity of $O(n^2)$ that is equivalent to the worst case time complexity of Delta Debugging. This case occurs when the test case is a flat list of elements with no structure. In contrast, the best case scenario is a tree with n nodes and a constant branching factor of b such that for each parent, exactly one child remains in the configuration. HDD's time complexity in this case will be the product of the number of levels in the tree ($log_b n$) and the number of tests performed at each level (b^2) that is $O(b^2 log_b n)$ or simply $O(logn)$. In practice, it is unlikely to have a constant branching factor in the parse tree of a test case and the number of children remaining for each parent node may also vary.

2.3.2 Fixed Point Reduction: Preserving 1-minimality by HDD

The term 1-minimality in tree based reduction techniques including HDD, referred to as *1-tree-minimal*, is also defined by Definition 2.1.6 in Section 2.1 in which every single *node* of the tree is required to trigger the failure. The reduced test case generated by HDD is not

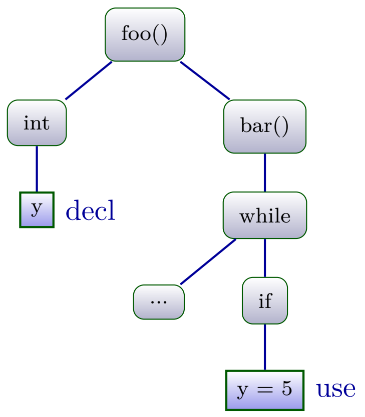


Figure 2.3: An example of a simplified tree in which HDD cannot preserve 1-minimality: The test for removing the declaration node of variable y is performed *before* removing the use node dependent on it and thus fails.

necessarily 1-minimal. The reason is that HDD performs tests level by level in a top down approach and does not revisit levels that have already been tested. However, the removal of a node at a lower level may *enable* the removal of a node at a higher level.

Consider a program with a declaration of a variable and its use in Figure 2.3. HDD visits the level with the declaration *before* the level with the use and cannot remove the declaration because it would cause an invalid program in which a variable is used without being declared. After proceeding to the lower levels and removing the use, the declaration node is now removable. However, the HDD algorithm is not able to revisit the node.

To solve this problem, two extensions to the HDD algorithm are proposed:

1. *HDD+:* HDD+ runs HDD for one iteration. Once terminated, it again tries removing nodes of the tree one by one, starting from the top-most level to the finest level. If at least one node is removed during this process, HDD+ starts another round and tries removing nodes one by one until it reaches a round where no node is removed.

2. *HDD*:* HDD* is called the fixed point version of the HDD algorithm. HDD* keeps calling the entire HDD algorithm on the tree in a loop. The loop terminates when a call to HDD cannot remove any further nodes.

Time Complexity. HDD+ has a worst case time complexity of $O(n^2)$ because the worst case time complexity of HDD is $O(n^2)$ and HDD+ adds at most n tests in n rounds which happens when the last node is removed in every round. HDD* has the worst case time complexity of $O(n^3)$ because it runs HDD at most n times. The better time complexity of HDD+ can make it a better approach to use for preserving 1-minimality. Moreover, a new direction for test case reduction emerged after HDD+ was proposed. This new direction introduced effective and efficient *node by node* reduction strategies implemented by Perses, the latest state of the art domain agnostic reducer [15] and our enhancement of it which we will discuss later in Section 2.4 and Chapter 4.

2.3.3 Limitations of HDD

Although HDD family of algorithms discussed so far (HDD, HDD+ and HDD*) consistently demonstrates better efficiency and effectiveness than Delta Debugging on structured domains [14, 23], it still suffers from some limitations that we discuss in this section.

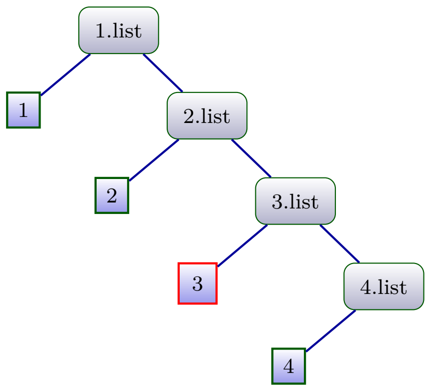
Unresolved Tests in HDD

Although HDD can significantly decrease the number of unresolved tests compared to Delta Debugging, it still generates a large number of these tests. The reason is that HDD leverages the structured domain knowledge to only build a parse tree of the test case but it does not use that knowledge to guide reduction on the generated nodes. In other words, HDD applies Delta Debugging on the list of *all* nodes present at a level. However, some of these nodes may not be even removable due to the grammar constraints of the test case. For instance, HDD may try to remove the return type of a function or its name in a program. Generating a function without a return type or name is not valid based on the *syntax* of programming languages and will lead to an unresolved test.

Unbalanced Trees

An unbalanced tree increases the number of tests performed by HDD significantly. One reason is that there are fewer nodes at each level of these trees, decreasing the chance of HDD to reduce. Moreover, nodes at different levels in an unbalanced tree may be strongly related and require to be considered together as a list of elements for a successful reduction. As an example, context-free grammars use recursion to deal with lists of data. Such recursion can generate unbalanced trees.

Consider the trees in Figure 2.4. Suppose that failure is caused by node $\boxed{3}$. Moreover, suppose that each element is dependent on all its previous elements. For instance, node $\boxed{1}$ cannot get removed if any of the nodes $\boxed{2}$, $\boxed{3}$ or $\boxed{4}$ is present in the tree. Node $\boxed{2}$ cannot get removed before $\boxed{3}$ and $\boxed{4}$ and node $\boxed{3}$ cannot get removed if $\boxed{4}$ is present. In the unbalanced tree (left), the recursive grammar rule list has generated a non-flattened tree that is not suitable for reduction by HDD. In this tree, first, the level with nodes $\boxed{1}$ and $\boxed{\text{2.list}}$ is visited. HDD cannot remove any of the nodes present at this level. Node $\boxed{1}$ cannot get removed due to the presence of $\boxed{2}$, $\boxed{3}$ and $\boxed{4}$ in the tree and $\boxed{\text{2.list}}$ cannot get removed due to containing the failure-inducing node, node $\boxed{3}$. Next, HDD performs similar unsuccessful reduction trials on nodes $\boxed{2}$ and $\boxed{\text{3.list}}$. The next level to target is the level with $\boxed{3}$ and $\boxed{\text{4.list}}$ in which $\boxed{\text{4.list}}$ can successfully get removed but $\boxed{3}$ cannot because removing it makes the failure disappear. The minimal test case returned by performing HDD on this tree is the list `{1, 2, 3}` which is not 1-minimal. On the other hand, if the tree is balanced (right), all elements will be present at the same level, enabling HDD to



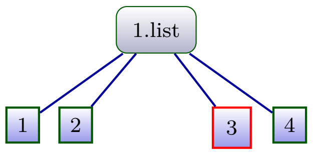


Figure 2.4: An unbalanced tree (left) generated by grammar recursive rules vs. a balanced tree (right) that provides more pruning opportunities.

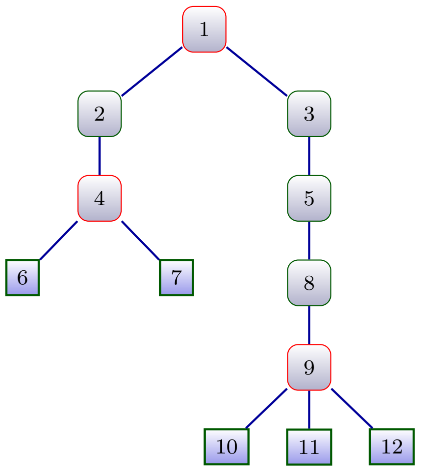


Figure 2.5: A tree with chains. Branching nodes shown in red are the only nodes required to be tested.

successfully remove all elements except for the failure-inducing node. The reduced test case generated by performing reduction on the balanced tree is `{3}` which is 1-minimal.

Unnecessary Tests

HDD performs tests that are not required to preserve 1-minimality. These tests are performed on nodes with only one child in a parse tree. These nodes comprise possibly long chains as shown in Figure 2.5. In parse trees, tokens of a test case occur only at the leaves of the tree. Hence, we will not miss any opportunities to remove tokens if we skip the nodes in a chain and only consider nodes with branches in the tree. The only internal (non-leaf) nodes in our example that may have a real impact on the size of the reduced test case are nodes $\boxed{1}$, $\boxed{4}$ and $\boxed{9}$ shown in red.

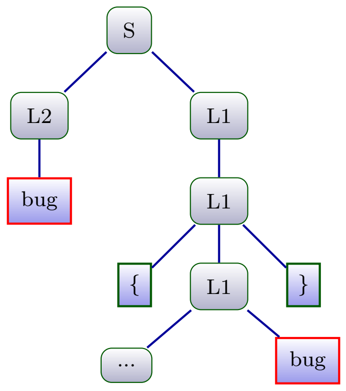


Figure 2.6: HDD cannot remove the pair of curly braces.

Limited Reduction Operations

HDD either tries to remove nodes or as we will see later in Section 2.3.4, it tries to replace a node with its minimal valid string. However, there are nodes in a tree that require a more complex reduction operation to be reduced. For instance, there are non-consecutive nodes at the same level that need to be considered for reduction together. Otherwise, individual attempts to remove them one by one will generate invalid variants. An example is when HDD tries to remove an opening curly brace (`{`) individually without considering its closing counterpart (`}`). Figure 2.6 depicts a tree where HDD cannot remove the braces and will include them in the final result even though they are not relevant to the bug.

In the next section, we discuss some improvements presented in the literature to solve or mitigate the aforementioned limitations.

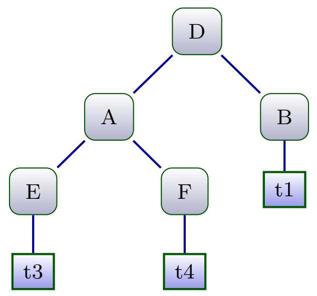
2.3.4 Improvements on HDD

Various improvement measures over HDD have been proposed in the literature since the introduction of the HDD algorithm more than a decade ago [23, 19, 10, 22, 36, 16, 17, 37]. In this section, we explore some of these measures that address the limitations described in Section 2.3.3.

Preserving Syntactic Validity

As mentioned in Section 2.3.3, one of the major challenges in using HDD is still the large number of unresolved tests caused by violating the grammar rules of the test case.

To ensure the syntactic validity of test case variants during reduction, Misherghi presents an extended version of HDD that is capable of preserving syntactic validity [23]. The proposed algorithm is based on a minimal-length string computation method. Figure 2.7 depicts a simple grammar and an example parse tree generated by it. The letters (A to F) represent non-terminal rules while t_i $i = 1, ..., 4$ are terminals or leaves that are tokens of the test case. Suppose that the subscript value i in t_i is the length of the string represented by that



D → A B | B | C
A → E F
B → t_1
C → t_2
E → t_3
F → t_4

Figure 2.7: A simple grammar and an example tree generated by this grammar. The minimal valid string for node D is t_1.

terminal (i.e., t_1 has a minimal string with length 1, t_2 has length 2, and so forth). In this grammar, alternate rules are separated using |. For instance, rule D can imply either A and B as its children or only B or only C. All three variants will be syntactically valid with respect to the grammar.

The minimal string for each node is computed recursively starting from the terminals of the grammar. Terminals have a minimal string equal to themselves. So t_i nodes with $i = 1, ..., 4$ have a minimal string with length i. Non-terminal nodes E, F and B have the same minimal string as t_i because their rules imply a single terminal. Node A has a minimal string that is the concatenation of strings from E and F (t_3t_4 with length 7). Finally, node D should compute its minimal string using its three alternate rules as follows:

If $F(n)$ is the function that computes the minimal string of node n, we have:

$F(D) = min\{F(AB), F(B), F(C)\} = min\{F(A) + F(B), F(B), F(C)\} = min\{7 + 1, 1, 2\} = 1$

So even though the tree generated by rule D is using A B as its children, since B alone is an alternate rule of D with a smaller string, we can replace the string represented by D with t_1 and generate a smaller test case that is syntactically valid.

Minimal string computation can decrease the number of invalid tests by providing information on the smallest possible entity a node in a tree and its corresponding grammar rule can have without violating the syntactic validity of the test case. This information is computed once ahead of time using the grammar rules and can be used as many times as required during reduction. Using this information, each node will be tried to be replaced with its minimal string. For instance, a complex arithmetic expression as a condition of an `if statement` can be replaced with a single value `0` or `1`, making reduction both 1) efficient by avoiding syntactically invalid tests that try to remove the condition and 2) effective by replacing a complex component with a simple entity.

Note: Replacing a node with a minimal string equal to *empty string* is equivalent to removing that node.

Extended Context-Free Grammars

As described in Section 2.3.3, recursive rules in context-free grammars cause unbalanced trees that are far from ideal for HDD. To mitigate this problem, Hodován and Kiss proposed *modernized* Hierarchical Delta Debugging [19] that uses an *extended* form of context-free grammars to replace recursive rules with rules that have quantifiers. More specifically, modernized HDD identifies all recursive rules in a grammar and replaces their right-hand side with one of the three quantifiers: Optional `(?)`, Kleene-Star `(*)` and Kleene-Plus `(+)` .

$$\boxed{\mathsf{A} \rightarrow \mathsf{B} \mid \mathsf{A}\ \mathsf{B} \quad \Longrightarrow \quad \begin{array}{l} \mathsf{A} \rightarrow \mathsf{B\_plus} \\ \mathsf{B\_plus} \rightarrow \mathsf{B}+ \end{array}}$$

Figure 2.8: A recursive context-free grammar rule and its extended quantified form.

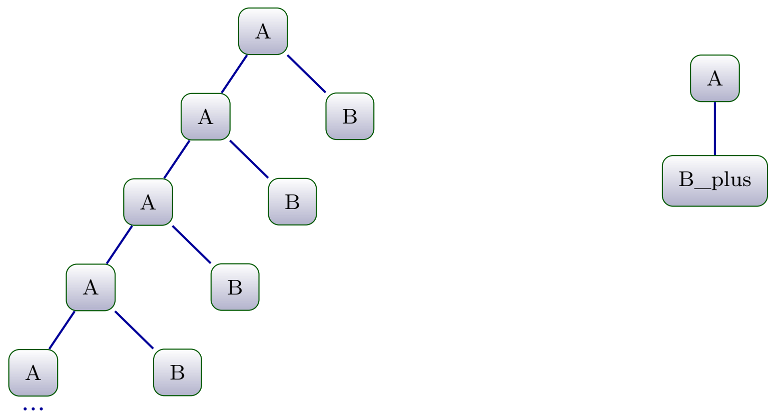


Figure 2.9: Trees generated by rules in Figure 2.8. Extended context-free grammars convert an unbalanced tree (left) to a balanced flattened one (right).

Rules extended by quantifiers are called *quantified rules* and the nodes of the tree generated by these rules are referred to as *quantified nodes*. `?` indicates that a node has an optional child (0 or 1) and `*` and `+` indicate 0 or more and 1 or more children for a node, respectively. Not only do extended context-free grammars flatten unbalanced trees but they also provide more reduction opportunities and make them explicit. For instance, rules with `?` and `*` can be safely removed (replaced with an empty string) and rules with the `+` quantifier can be replaced with the minimal string of one of their children. Figure 2.8 and Figure 2.9 depict a context-free grammar rule and its extended quantified form along with the trees generated by them, respectively. As can be seen, use of the quantified rules in the grammar flattens the unbalanced tree generated by recursive rules.

Sun et al. [15] further explore the idea of leveraging quantifiers in program reduction in their reducer tool called Perses. We will discuss Perses in more detail in Section 2.4.

Further Reduction Operations: Node Replacements

As discussed in Section 2.3.3, HDD is incapable of pruning some parts of the test case that are not relevant to the failure due to its limited reduction operations. For instance, HDD cannot remove the pair of curly braces in Figure 2.6. As another example, consider the following code:

```
if (cond1 || cond2) { exhibitBug(); }
```

To reduce this code, HDD will first try to remove the entire `if statement` which is not a successful reduction since the exhibition of the failure is dependent on the code nested within the body of the `if statement`. It then tries to remove the condition of the `if statement` which will yield an invalid program. The more advanced version of HDD with minimal string computation [23] discussed earlier in this section is also not able to entirely remove the condition of the `if statement` and will generate a minimal test case similar to `if (1) { exhibitBug(); }` in which the condition is replaced with its minimal valid string. To distill the only part relevant to the failure (i.e., `exhibitBug();`), reduction operations other than simple removal and string replacement are required.

To enable more reduction, Morton and Bruno proposed FlexMin [10] that extends the set of reduction operations to include both removal and another operation referred to as *node substitutions*. If a node at a higher level of the tree gets replaced with one of its descendants and still satisfies the oracle, it can perform reductions that will not be possible otherwise. For instance, in Figure 2.6, the node L_1 which is the direct parent of the right-most bug in the tree can replace its parent node, leaving out the curly braces while still preserving the bug. In the example of `if statement` with `exhibitBug();`, the nested statement, `exhibitBug();`, can be lifted up, removing the entire `if statement` and its condition. Performing node substitutions can increase the effectiveness of the reducer by generating smaller test cases [37]. However, with respect to efficiency, it may degrade the performance since trying every single descendant as a replacement candidate is syntactically valid (due to transitivity of grammar rules that will be more discussed in Section 2.4.1) but may not be scalable for large test cases.

To mitigate this problem, some restrictions are applied when looking for node replacement candidates. For instance, FlexMin and its basis infrastructure SIMP [9] that is a previous work by Bruno define boundaries based on the transitivity of grammar rules and their labels. In particular, in their approach, only descendants that have *the same* grammar rule's label as their ancestor will be considered as replacement candidates in a top down traversal order. As other solutions, Perses [15] limits its length of search space to a specific boundary within the tree and Herfert et al. [22] propose a filtering mechanism on replacement candidates based on learning from data in their approach called generalized tree reduction (GTR). GTR limits its search space to the children of a node but instead of using the labels in grammar rules, it leverages a corpus of data that has been automatically collected from a large number of test cases in the same domain. Using this collection, GTR filters out those candidate replacements that will restructure the tree such that the new structure has not been observed in the programs of the corpus.

In this dissertation, we provide results for both *removal only* and *removal and replacement* when comparing our techniques with the state of the art reducers.

Squeezing Trees

In Figure 2.5, we presented a tree structure with long chains that makes HDD perform unnecessary tests. In an attempt to improve HDD's performance, another work proposed by Hodován and Kiss [36] introduced a simple but effective algorithm to squeeze the parse tree vertically. This algorithm skips performing tests on nodes present in the middle part of a chain that have a minimal string identical to the last node of the chain, referred to as the branching node in Figure 2.5. Using this algorithm, the efficiency of HDD improves by skipping tests that can not have any impact on the size of the reduced test case.

2.4 Program Reduction

Programs are specific kinds of test cases that are useful for testing and debugging software such as compilers and interpreters [13, 15]. In particular, large fuzzer-generated programs are widely employed during fuzzer stress tests of software in which the software under test is bombarded by various randomized input test cases to examine software's behavior during potentially unexpected situations [4]. As a result, program reduction becomes essential to generate small and easy to comprehend programs to facilitate the debugging task [7]. Moreover, program reduction has also been useful for minimizing the attack surface of programs [38, 39, 40, 41, 35], reducing resource consumption [25], and helping to understand neural models of code [42, 26].

Although Hierarchical Delta Debugging and the subsequent works for its improvement discussed in Section 2.3.4 address the problem of test case reduction on structured test cases, including programs and propose solutions that are effective and efficient to some degree, the several applications of program reduction make it deserve especial attention from the community. In this section, we provide some background on one of the latest state of the art program reducers, Perses [15]. Perses is a domain agnostic syntax based queue driven approach that is suitable for reducing programs of multiple language domains. In addition, as mentioned earlier in this chapter, Perses is also capable of reducing test cases other than programs from domains with structure such as XML files. However, we continue to refer to Perses as a *program reducer* as initially introduced by its creators.

Compared to domain specific tools such as C-Reduce [13] that is capable of effectively reducing programs of domain C, Perses or any other domain agnostic reducer generate potentially larger final reduced outputs due to their general and domain agnostic reduction operations (refer to Section 6.1 for some results). However, the generality, reusability and availability of domain agnostic reducers on multiple domains without requiring significant additional effort and expertise make them suitable options for efficiently reducing programs of multiple languages.

2.4.1 Perses: A Domain Agnostic Program Reducer

Perses [15] is a general syntax guided program reducer. Given the programming language grammar, Perses can reduce any structured input in any programming language such as a C test case causing a compiler to crash or a Java program with a specific property of interest.

Considering $\mathbb{P}$ as the set of all possible programs in the search space of Perses to explore during reduction. $\mathbb{P} = \mathbb{P}_{valid} \cup \mathbb{P}_{invalid}$ such that $\mathbb{P}_{valid}$ is the set of programs that do not violate *syntactic* rules of the grammar while $\mathbb{P}_{invalid}$ is the set of *syntactically* invalid programs encountered during reduction. Perses has an empty $\mathbb{P}_{invalid}$ due to leveraging syntactic knowledge of grammars during reduction. For example, removing the return type of a function declaration would not be valid because the C grammar specifies that the return type is required. Such syntactically invalid program candidates are removed from the search space of Perses. Moreover, Perses has a larger $\mathbb{P}_{valid}$ compared to other state of the art reducers such as modernized HDD (see Section 2.3.4). This larger set of valid programs provides more reduction opportunities to explore, making it possible for Perses to reduce more effectively.

In contrast to Hierarchical Delta Debugging that traverses the parse tree of a test case level by level and applies reduction on all the nodes present at each level, Perses leverages a priority queue to traverse the parse tree node by node. The number of tokens that are descendants of a node defines the priority of a node in the queue such that a larger number of tokens indicates a higher priority. This can be a more helpful strategy than reducing an entire level before moving to the next one. The reason is that when reducing a level, there may be other nodes at other levels of the parse tree such that reducing them boosts the performance of the reducer. For instance, those nodes may have a large number of nodes and tokens beneath them and removing them will lead to smaller test cases within fewer tests or they may depend on other nodes and removing them early can break those dependencies and enable further reduction.

Priority Queue: A Prioritized Traversal of the Parse Tree

Perses defines the number of tokens that are descendants of the node in the parse tree as the priority metric used in its priority queue. A node with a larger number of tokens will be attempted *before* a node with a smaller number of tokens. Remember that tokens are the smallest meaningful units in a program that occur only at leaves of a parse tree. Perses starts by adding the root of the tree to the priority queue. If reducing a node is unsuccessful, Perses adds its children to the queue to be processed later with respect to their priority. If a reduction is successful, Perses updates the parse tree and proceeds with the next node in the queue until the queue becomes empty or Perses times out.

Choosing the next available node from the queue, Perses performs one of the following reduction operations on it:

1. Removing its children by applying Delta Debugging on them.

2. Replacing the node with one of its descendants.

More specifically, Perses categorizes nodes of the parse tree into the two groups of *quantified* and *regular* nodes. As explained earlier in Section 2.3.4, nodes with a quantifier such as a Kleene-Star (*), Kleene-Plus (+) or Optional (?) quantifier are quantified nodes. Any non-terminal node that is not a quantified node is a regular node. By selecting an appropriate type of reduction for each node based on its type, Perses keeps an empty set of invalid programs and a large set of valid programs during reduction. In the following, we explain the details.

Children Removal

Perses applies deletion only on nodes that can evaluate to an empty string. Hence, it can preserve an empty set of syntactically invalid programs. These nodes include all children of a Kleene-Star (*) node, all children except for one child of a Kleene-Plus (+) node and the only child of an Optional node (?). In other words, Perses applies Delta Debugging on the list of elements that are syntactically independent. For instance, a statement_star node generated by a Kleene-Star quantified rule, can have zero or more statements as its children. These children are syntactically independent. By applying Delta Debugging on the list of children of statement_star, Perses removes as many statements as possible without violating any syntactic rules, keeping $\mathbb{P}_{invalid}$ empty.

Node Replacement and Compatibility Rules

In addition to applying deletion on the children of quantified nodes, Perses applies a replacement reduction operation on regular nodes such that the node under reduction will be tried to get replaced with a *compatible* descendant node. We have fully described the node replacement operation in Section 2.3.4. Here, we discuss the compatibility rules Perses leverages to find the node replacement candidates.

Suppose that A is the expected grammar rule of a node based on its parent (e.g., a node with a parent statement_star is expected to have statement as its type). Node B is compatible with node A and can replace it if any of the following is true:

1. $A = B$: Both A and B have the same rule type in the grammar. For instance, they are both statements.

2. B is subsumed by A ($B <: A$): If B can be derived from A directly or transitively (e.g., either $A \rightarrow B$ or ($A \rightarrow C$ and $C \rightarrow B$)). Note that in order for this case of compatibility to be true, A should be able to produce B alone rather than B along with other byproducts.

   As an example, consider the following C grammar rule:

   statement→ if_statement | decl_statement | expr_statement | ...

   An if_statement is subsumed by statement. Hence, an if_statement node can replace a statement node and still generate a syntactically valid program.

3. If parent of A is a quantified rule (e.g. $A\_star \rightarrow A*$), $B <: A$ and B is the quantified part of a quantified rule (e.g. we have $B\_star \rightarrow B*$), then A can be replaced with B_star without violating the grammar rules because both its parent and B_star can have zero or more children.

Figure 2.10 depicts an example of replacement operations performed by Perses based on the compatibility rules described above. The property to preserve is the node in red and suppose that the node to replace is 2.if_stmt. This simplified tree is generated by the following condensed rules:

stmt_star→ stmt*
stmt→ if_stmt | compound_stmt

Perses first tries to replaces 2.if_stmt with 3.compound_stmt because they are compatible based on the second compatibility rule discussed above. In more detail, the expected rule of 2.if_stmt based on its parent 1.stmt_star is stmt and the grammar rule stmt→ if_stmt | compound_stmt infers that 3.compound_stmt is subsumed by stmt. After this successful operation, the next operation will replace 3.compound_stmt with 4.stmt_star which are compatible based on the third compatibility rule. The parent of 3.compound_stmt is 1.stmt_star (remember that 3.compound_stmt has replaced 2.if_stmt in the previous operation and has a new parent) which is a Kleene node and 4.stmt_star is a Kleene node too with the quantified part stmt that is compatible with the expected rule of 3.compound_stmt that is also stmt (the first rule of compatibility).

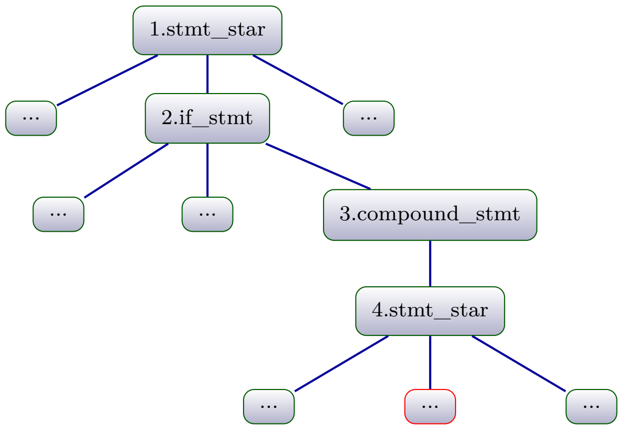


Figure 2.10: A tree example with node replacement opportunity.

Applying replacement reduction templates by Perses keeps the syntactically invalid set of programs empty during reduction and also provides a larger set of valid programs to explore.

Remember that there are state of the art techniques such as FlexMin [10] and GTR [22] discussed in Section 2.3.4 that also perform replacement operations. However, their replacement strategy is less powerful compared to Perses. In particular, these two state of the art techniques are not capable of replacing a list of elements with another list of elements even though the types of the elements in the two lists are compatible. For instance, consider a list of elements with type C which has a nested list of elements of type B and B is subsumed by C). More specifically, we have the following:

```
(C C C (B B B B)* C C C C C)*
```

Perses can replace the large outer list with the smaller inner list of B elements while FlexMin and GTR cannot explore this smaller variant.

Perses Normal Form (PNF) Grammar

In addition to the priority queue and different reduction templates, Perses uses a framework for automatically converting any grammar in Backus-Naur Form (BNF) to a grammar with quantified rules called the Perses Normal Form (PNF). There are three intentions behind this conversion:

1. Having a consistent form of grammar across different test cases.

2. Avoiding recursive occurrences of production rules in the parse tree of a test case and replacing them with quantified rules such that more balanced parse trees that are more suitable for reduction will be generated (see Figure 2.9 for illustration).

3. Explicitly annotating the *syntactically* removable portions of a parse tree by leveraging quantifiers. As a result, reduction can be guided by deleting children of quantified nodes or replacing a node with a quantified parent with another quantified rule.

To convert a BNF grammar to an equivalent PNF, Perses first removes all epsilon production rules (except for one that can define an empty input) by transforming the BNF grammar rules to equivalent ones without an epsilon. Moreover, it removes rules that are not reachable in a preprocessing step. It then transforms the grammar by defining all rules with direct left or right recursions. It means that rules that are *transitively* recursive will be transformed into rules that are *directly* recursive. Finally, Perses normalizes the grammar by using Kleene-Star (*), Kleene-Plus (+) and Optional (?) quantifiers to replace recursive rules with normalized rules. (see Figure 2.8 for an example of converting a recursive rule into a quantified rule).

2.4.2 C-Reduce: A Domain Specific Program Reducer

C-Reduce proposed by Regehr et al. [13] is a powerful tool capable of effectively reducing programs of domain C. By leveraging specific knowledge of the C programming language, C-Reduce defines a large set of reduction transformations, particularly tailored to the C programming language that domain agnostic reducers are typically not capable of performing them. By applying these transformations, C-Reduce can explore a larger search space to find the minimal test case.

While C-Reduce tends to produce smaller programs due to its larger search space, it requires significant effort and expertise to implement which limits its reusability and availability for reducing inputs in other languages. Moreover, its larger search space can adversely impact the reduction speed and efficiency. In this dissertation, our focus is on improving *domain agnostic* reducers, in particular Perses. These reducers provide an exciting direction for us because they are more efficient at reducing test cases across various domains and can be developed and reused without specific domain knowledge or expertise.

2.5 Problem Statement

As mentioned in Chapter 1, the principal objective of this dissertation is to address the problem of *inefficient test case reduction* and propose solutions to mitigate it. Perses is one of the latest well-known state of the art domain agnostic reducers and we believe that addressing its shortcomings can indeed improve the performance of existing test case reduction practices. To this end, the list of issues we explore in this dissertation is defined by the limitations of Perses outlined in this section. We demonstrate throughout the dissertation that each of these limitations contributes to the degradation of reduction performance and propose and evaluate novel techniques to lessen their negative effects in practice.

The list of problems we aim to address and the overview of solutions we propose in this dissertation are as follows:

Problem 1: Perses applies Delta Debugging on the list of children of quantified nodes. The worst case time complexity of Delta Debugging is *quadratic* (see Section 2.2). When such lists are long, significant time can be devoted to this task.
Solution 1: In Chapter 3, we propose One Pass Delta Debugging, a modification of the original Delta Debugging algorithm. One Pass Delta Debugging has a *linear* worst case time complexity as opposed to Delta Debugging's quadratic time complexity.

Problem 2: Perses has a *suboptimal* priority mechanism. This suboptimality is twofold:
Problem 2.1: Perses defines priority of nodes in its queue such that a significant amount of reduction time may be spent with little or no progress in reduction. We refer to this limitation as *priority inversion* and fully explain it in Section 4.1. In general, priority inversion occurs when a low priority task is scheduled instead of a high priority task.
Solution 2.1: In Chapter 4, we propose Pardis and its variant Pardis Hybrid with different methods of prioritizing nodes in the queue that can help to remove larger portions of the test case earlier, leading to a faster convergence towards the minimal test case.
Problem 2.2: Perses defines the priority of a node in the queue *solely* based on its number of token descendants and explores parse trees in orders that can hinder successful reduction.
Solution 2.2: In Section 5.2, we propose Type Batched Reducer that leverages information other than the number of token descendants to prioritize nodes. Using the information that is easily achievable by the grammar, Type Batched Reducer makes use of machine learning models to find the most advantageous batch of nodes to reduce at a given point in time during reduction.

Problem 3: Despite preserving syntactic validity, Perses generates *Semantically* invalid program variants during reduction. Exploring these invalid variants wastes the reduction time and effort.
Solution 3.1: In Section 5.1, we propose Model Guided Pardis that employs machine learning models, capable of predicting and *avoiding* the performance of semantically invalid tests.
Solution 3.2: Our Type Batched Reducer proposed in Section 5.2 addresses this problem from a different perspective by selecting the most advantageous nodes with lower likelihood of semantic invalidity and *guiding* the approach towards reducing them.

Problem 4: Despite applying Delta Debugging on the list of children of quantified nodes, Perses lacks an effective and suitable *joint reduction* mechanism in which multiple nodes can get removed successfully together.
Solution 4: In Section 5.2.3, we propose Probabilistic Joint Reduction, a technique that works by learning over the nodes of the most advantageous batch returned by Type Batched Reducer and trying to further prioritize the removal of nodes with higher likelihood of removal success.

In the next section, we describe our experimental design for studies across the dissertation.

2.6 Experimental Setup

We design multiple studies throughout this dissertation. Some studies are the *overall evaluation* of the techniques in which we compare our novel methods with the existing reducers. The other studies are *explanatory*, conducted to motivate some property, elaborate on results or investigate some potential directions.

In the following, we provide information on our benchmark, execution environment and the list of performance metrics we use to compare different reduction techniques.

2.6.1 Benchmark

To prevent bias as much as possible on our part, we use the benchmark provided by Perses and available on their Github repository [43]. Perses' benchmark consists of 38 programs or test cases from three different domains: 26 test cases are C programs, produced by Csmith [4], a tool for generating random C programs that statically and dynamically conform to the C99 standard [44], 9 test cases are Rust programs and 3 are Go programs. All these programs cause real-world bugs in compilers such as GCC, Clang, Rust and Go. Out of these 38 programs, we could replicate the behavior of 28 programs in our execution environment. Our benchmark of these 28 programs consists of 17 C, 8 Rust and 3 Go programs shown in Table 2.1 with their original sizes with respect to the number of tokens, nodes of the parse tree and bytes.

To demonstrate the generality of our techniques, we also include an XML file in our benchmark. This XML file triggers failure in XMLProc [45] , an XML parser written in Python. Extending our benchmark with an XML file can help us to verify the performance of our techniques in reducing structured test cases other than programs.

The grammars we use to build parse trees of test cases are also the ones available on Github repository of Perses [43]. For the overall evaluation studies, we use the full benchmark set (i.e., 28 programs and one XML file). For explanatory studies, we choose a sample chosen randomly from our benchmark using the python random generator module [46] and use it consistently across the dissertation. We embed at least one program from each category into our sample benchmark. In particular, our sample benchmark consists of 9 programs: 2 C programs causing Clang compiler to fail, 3 C programs causing GCC compiler to fail, 3 Rust and 1 Go programs exhibiting bugs in Rust and Go compilers. The programs included in our sample benchmark are highlighted in yellow in Table 2.1.

Henceforth, we use the terms *test case reduction* and *program reduction* interchangeably because the majority of our test cases are large programs causing bugs in real-world compilers.

Table 2.1: Benchmark used in our experiments across different chapters of the thesis to consistently compare various reduction techniques.

SUT/Test Case	**Original Sizes**		
	tokens(#)	nodes(#)	bytes(#)
clang-22382	21,069	167,352	65,840
clang-22704	184,445	1,519,746	597,993
clang-23309	38,648	320,379	118,222
clang-25900	78,961	646,343	245,077
clang-27747	173,841	1,412,020	410,891
clang-31259	48,800	403,890	137,203
gcc-59903	57,582	469,297	166,786
gcc-60116	75,225	624,170	218,251
gcc-61383	32,450	246,936	110,655
gcc-61452	26,733	218,911	88,593
gcc-61917	85,360	687,028	254,778
gcc-64990	148,932	1,260,679	439,659
gcc-65383	43,943	365,149	125,271
gcc-66186	47,482	390,895	139,111
gcc-71626	6,134	39,547	14,465
gcc-71632	141	966	183
gcc-77624	1,306	5,415	4,856
geomean	29,769	228,004	84,543
median	47,482	390,895	137,203
rust-44800	802	5,179	2,078
rust-63791	8,144	61,008	16,455
rust-65934	107	507	217
rust-69039	191	1,058	589
rust-77002	348	3,080	607
rust-77993	4,989	29,655	17,127
rust-78336	980	4,972	2,980
rust-78622	157	815	227
geomean	659	3,978	1,534
median	575	4,026	1,343
go-28390	146	523	294
go-29220	127	551	323
go-30606	449	1,604	1,270
geomean	203	773	494
median	146	551	323
urls.xml	679	1,501	4,217

Test cases included in our sample benchmark are highlighted in yellow.

2.6.2 Execution Environment

We run our experiments on Ubuntu 16.04 inside a docker container with Intel(R) Core(TM) i5-7300U CPU @ 2.6 GHz and 8 GB of RAM. Our techniques are implemented in C++. Both the implementation and the data are available online [47].

2.6.3 Performance Metrics

Similar to existing techniques [15, 14, 1, 13, 19], we use the following performance metrics to compare the performance of different reduction techniques:

1. The number of *oracle queries or tests* (Q)

2. *reduction quality* or *size* of the final reduced test case (R)

3. *reduction time* that is the wall-clock time required to reduce the test case (T)

4. *reduction efficiency, rate or speed* that is the average number of tokens removed per second (E).

In general, a smaller reduced size obtained by fewer oracle queries within a shorter reduction time with higher efficiency shows a better performance.

The next chapters explain and evaluate our novel techniques for test case reduction.

Chapter 3

One Pass Delta Debugging

This chapter addresses the first problem in our list of problems in Section 2.5 and explores the following question:

> Can we convert the quadratic worst case time complexity of Delta Debugging into linear to improve its efficiency in practice without adversely affecting its effectiveness?

We originally introduced and published the ideas presented in this chapter at the 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS 2018) [2].

Recall the background information on the steps of the Delta Debugging algorithm presented in Section 2.2. The original Delta Debugging works by first partitioning a test case into *chunks*, also referred to as *subsets*, and then attempting to remove or retain individual subsets to produce a smaller test case. Removing one part of a test case may enable *other* parts of the test case to also be removed. Thus, when Delta Debugging successfully removes a subset of the test case (i.e., a successful step 2 of the $ddmin_2$ algorithm in Figure 2.2, referred to as *reduce to complement*), all remaining subsets are *revisited*, as they may become removable themselves. When no further subsets can be removed, the subsets are recursively partitioned and the process continues. Pathologically, the last subset visited may successfully get removed, causing all remaining subsets to be visited again. Because successfully removing a subset may cause all already visited subsets to be revisited and tested again, this revisiting behavior causes the algorithm to perform $O(n^2)$ tests in the worst case where n is the size of the test case. For large test cases, this can be costly, easily taking hours to reduce a single test case [13, 1].

The dependencies among elements of a test case are one of the causes of an element being non-removable when it is *first* visited, yet being successfully removed in *subsequent* visits when its dependents are removed. To better understand the concept of revisiting caused by dependencies among elements of a test case, remember our set of numbers, $\tau_{\text{✗}}$=`{1, 2, 3, 4, 5, 6, 7, 8}`, being reduced by Delta Debugging in Section 2.2. Again, each number represents an element of the test case that Delta Debugging tries to remove. This time, suppose that the minimal failure-inducing portion of the test case is the subset `{1, 3, 5,`

Step		Test Case τ'								$\psi_1(\tau')$	Discription
0	$\tau_{\text{✗}}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**True**	**initial test case**
1	$\Delta_1 = \nabla_2$	1	2	3	4	.	.	.	.	False	test subset
2	$\Delta_2 = \nabla_1$	.	.	.	.	5	6	7	8	False	test subset
3	Δ_1	1	2	.	.	.	.	.	.	False	granularity increased test subset
4	Δ_2	.	.	3	4	.	.	.	.	False	test subset
				...							
5	∇_1	.	.	3	4	5	6	7	8	False	test complement
6	∇_2	1	2	.	.	5	6	7	8	False	test complement
				...							
7	Δ_1	1	.	.	.	.	.	.	.	False	granularity increased test subset
8	Δ_2	.	2	.	.	.	.	.	.	False	test subset
				...							
9	∇_1	.	2	3	4	5	6	7	8	False	test complement
10	∇_2	1	.	3	4	5	6	7	8	False	test complement
				...							
11	∇_8	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**.**	**True**	**reduce to complement**
12	∇_1	.	2	3	4	5	6	7	.	False	revisiting
13	∇_2	1	.	3	4	5	6	7	.	False	revisiting
				...							
14	∇_6	**1**	**2**	**3**	**4**	**5**	**.**	**7**	**.**	**True**	**reduce to complement**
15	∇_1	.	2	3	4	5	.	7	.	False	revisiting
				...							
16	**Result**	**1**	**.**	**3**	**.**	**5**	**.**	**7**	**.**	**True**	**Done**

Figure 3.1: Quadratic time complexity in original Delta Debugging. Revisiting steps after each reduce to complement are highlighted. The even numbers are removed in backward order. Oracle ψ_1 is defined in Figure 3.2.

$$\psi_1(\tau) = \begin{cases} True & \text{if } (1 \wedge 3 \wedge 5 \wedge 7) \text{ in } \tau \\ False & \text{else} \end{cases}$$

Dependencies: $2 \leftarrow 4 \leftarrow 6 \leftarrow 8$

$$\psi_2(\tau) = \begin{cases} True & \text{if } (1 \wedge 3) \text{ in } \tau \\ False & \text{else} \end{cases}$$

$$\psi_3(\tau) = \begin{cases} True & \text{if } (3 \wedge 8) \vee (4 \wedge 6) \text{ in } \tau \\ False & \text{else} \end{cases}$$

Figure 3.2: Oracles used in Figure 3.1 and Figure 3.3. Dependency $\leftarrow$ for ψ_1 means that removing the element on the left side of $\leftarrow$ before removing the element on the right will not induce the failure.

`7}` in which every single number is necessary to trigger the failure. Moreover, suppose that there is dependency among numbers `2`, `4`, `6` and `8` such that `2` cannot get removed without first removing `4`, nor `4` without first removing `6` and nor `6` without removing `8`. In this case, all tests until *ddmin* reaches the finest granularity do *not* induce the failure as shown in Figure 3.1. Then the complements of the last removable elements succeed in reverse order, requiring all of the remaining complements to be tried again. More specifically, if the oracle returns $True$ for a complement test (e.g., step 11 in Figure 3.1), a revisiting process is performed without repartitioning to re-run all previous complement tests in that granularity with the new updated test case.

Three rows are highlighted in Figure 3.1 to depict this revisiting process clearly. After successfully removing `8` from the test case and reducing it to `{1, 2, 3, 4, 5, 6, 7}` in step 11, `1` is reconsidered for removal from the new test case, `{1, 2, 3, 4, 5, 6, 7}` in the next step. Similarly, elements `2`, `3`, `...`, `7` are also reconsidered for removal from this new set of numbers. This revisiting occurs after each successful removal of a subset that is the reduce to complement step.

In this example, the quadratic behavior can be eliminated by simply performing *ddmin* in reverse order on the test case, but generally, the dependencies between parts of a test case need not occur in only one direction. Instead, we may ask, what is the benefit of this revisiting in practice? Can we practically avoid this $O(n^2)$ process under certain circumstances? To answer, we describe three conditions under which performing the revisiting process in Delta Debugging becomes unnecessary.

3.1 Conditions for Skipping Revisiting

The revisiting behavior is a key part of ensuring the minimality guarantees that Delta Debugging provides [1]. By skipping this process, one might expect that the test case reduction becomes grossly less effective. In this section, we discuss *why* skipping revisits might be okay. To this end, we identify three key reasons: *common dependence order*, *unambiguity* and *deferred removal.* Satisfaction of these three key formal and empirical conditions of test case reduction can enable the process to produce minimal or near-minimal test cases while skipping revisiting of already considered subset removals.

3.1.1 Common Dependence Order.

Just as in Figure 3.1, sometimes there is a preferred ordering that can improve the overall efficiency of Delta Debugging. For example, when removing statements from a C program, uses of identifiers must be removed before their declarations in order to satisfy validity constraints. Since the uses mostly occur after their definitions, removing the lines of a C file in reverse order is likely to remove the uses first and thus enable removing declarations when they are first visited. When the common ordering of dependencies for a particular test

Test Case									$\psi_2(\tau_{✗})$	$\psi_3(\tau_{✗})$
τ :	1	2	3	4	5	6	7	8	True	True
τ_1 :	1	2	3	4	5	6	.	.	True	True
τ_2 :	1	2	3	4	.	.	7	8	True	True
$\tau_1 \cap \tau_2$:	1	2	3	4	.	.	.	.	True	**False**

Figure 3.3: Satisfaction and dissatisfaction of unambiguity with ψ_2 and ψ_3 (defined in Figure 3.2), respectively.

case structure is known, the complement trials can be ordered to respect the dependencies. We refer to this condition as the *common dependence order*.

3.1.2 Unambiguity.

Unambiguity is a property first proposed in an initial version of Delta Debugging [34] called dd^+.

Definition 3.1.1 (**Unambiguity**). A failing test case τ is unambiguous if $\forall \tau_1, \tau_2 \subseteq \tau, \psi(\tau_1) = True \wedge \psi(\tau_2) = True \implies \psi(\tau_1 \cap \tau_2) = True$.

Informally, an ambiguous test case (the one in which unambiguity *does not* hold) has multiple causes for its failure and a reduced test case satisfying *any* of these causes will be failure-inducing. Consider the example in Figure 3.3. ψ_2 in Figure 3.2 returns *True* if both `1` and `3` are present in the test case and returns *False* otherwise. Test case τ with this oracle is unambigious since both `{5, 6}` and `{7, 8}` can be removed from τ individually, and the test case generated by combining the results of these two removals still induces the failure. Now consider τ with ψ_3. For this oracle, the property of interest is the presence of either `{3, 8}` or `{4, 6}`. Definition 3.1.1 does not hold in this scenario because:

$$\psi_3(\{\texttt{1, 2, 3, 4, 5, 6}\}){=}True \wedge \psi_3(\{\texttt{1, 2, 3, 4, 7, 8}\}){=}True$$
$$\textbf{but } \psi_3(\{\texttt{1, 2, 3, 4}\}){=}False.$$

Note that an unambiguous test τ enables reasoning about subtests τ_1 and τ_2 independently because the results can be recombined using intersection, which is independent of order. In other words, $\tau_1 \cap \tau_2 = \tau_2 \cap \tau_1$. As a result, if unambiguity holds for a test case, the order in which reduce to complement operations are performed is irrelevant and revisiting need not be performed. The relevance of revisit tests not being necessary increases in parallel Delta Debugging [31] where multiple subsets may be simultaneously tried for removal.

3.1.3 Deferred Removal.

The last condition is *deferred removal*. When this property is satisfied, a potentially successful subset removal that is skipped, will be considered for removal in the form of two smaller subsets when the granularity is increased through refinement in the natural flow

ID	n	Test Case τ												$\psi(\tau)$
1	6	.	.	3	4	5	6	7	8	9	10	11	12	False
2	6	1	2	.	.	5	6	7	8	9	10	11	12	False
				...										
6	**6**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**.**	**.**	**True**
7	5	.	.	3	4	5	6	7	8	9	10	.	.	False
8	5	1	2	.	.	5	6	7	8	9	10	.	.	False
9	5	1	2	3	4	.	.	7	8	9	10	.	.	False
10	5	1	2	3	4	5	6	.	.	9	10	.	.	False
11	**5**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**.**	**.**	**.**	**.**	**True**
12	4	.	.	3	4	5	6	7	8	.	.	.	.	False
13	4	1	2	.	.	5	6	7	8	.	.	.	.	False
14	4	1	2	3	4	.	.	7	8	.	.	.	.	False
15	4	1	2	3	4	5	6	.	.	.	.	.	.	False
16	8	.	2	3	4	5	6	7	8	.	.	.	.	False
				...										

Figure 3.4: Complement tests with revisiting enabled. Removed portions are `{11, 12}` and `{9, 10}`

.

ID	n	Test Case τ												$\psi(\tau)$
1	6	.	.	3	4	5	6	7	8	9	10	11	12	False
2	6	1	2	.	.	5	6	7	8	9	10	11	12	False
				...										
6	**6**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**.**	**.**	**True**
7	10	.	2	3	4	5	6	7	8	9	10	.	.	False
				...										
15	**10**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**.**	**10**	**.**	**.**	**True**
16	**10**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**.**	**.**	**.**	**.**	**True**

Figure 3.5: Complement tests with revisiting disabled, removing the same elements as reduction with enabled revisiting due to satisfaction of the deferred removal property. Removed portions are `{11, 12}`, `{9}`, and `{10}`.

of the Delta Debugging algorithm (line 3 of *ddmin*$_2$ in Figure 2.2). Formally, we have the following:

Definition 3.1.2 (Deferred Removal). Suppose that τ is a failing test case and $\Delta = \{e_1, e_2\} \subseteq \tau$ is a subset of τ such that $\psi(\tau - \Delta) = True$. Deferred removal is fully satisfied $\iff$ $\psi(\tau - \{e_1\}) = True$ and $\psi(\tau - \{e_2\}) = True$ where $\{e_1\}$ and $\{e_2\}$ are individually removed from τ in the selected order of removing subsets that is consistent across all granularities.

Consider the examples in Figure 3.4 and Figure 3.5. Figure 3.4 depicts reduction by original Delta Debugging in which revisiting is performed while reduction in Figure 3.5 does not perform revisiting. Suppose that at some point during reduction, we have test case variant `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}` with $n = 6$ as the number of subsets present at the granularity. Moreover, suppose that `{9, 10}` cannot get removed before `{11, 12}` and both of them are the only portions irrelevant to the failure. The highlighted rows in Figure 3.4 are the revisiting tests performed after Delta Debugging successfully removes `{11, 12}` and `{9, 10}` from the test case. While these tests are skipped in Figure 3.5, the removed elements are the same in both figures. The reason is that row with ID 11 in Figure 3.4 is replaced with rows 15 and 16 in Figure 3.5 that remove the same elements at subsequent finer granularities when revisiting is disabled. The term *deferred* in deferred removal property does, in fact, refer to this *delayed* removal performed at later granularities.

Note that deferred removal comes at a cost. Every time that refinement occurs, a subset that could have been removed but was not turns into two smaller subsets that will be tested independently. Thus, the cost of removing a missed subset may grow exponentially with the number of refinements. However, there is a logarithmic number of refinements (granularities) in the process which makes the worst case cost linear. At the same time, revisiting as in original Delta Debugging would require reconsidering all reduce to complement opportunities, leading to the $O(n^2)$ behavior but possibly at much coarser granularities where n is smaller. It is not clear what the performance trade off is in practice. We empirically explore this trade off in Section 3.3. In the next section, we present the formal definition of a Delta Debugging algorithm that does not perform revisiting tests. We call this algorithm ONE PASS DELTA DEBUGGING [2].

3.2 One Pass Delta Debugging: The Algorithm

Inspired by the observation that revisiting subsets after a reduce to complement operation may not be necessary, we consider a variant of Delta Debugging that considers each subset and its complement *only once* per granularity. We call this variant ONE PASS DELTA DEBUGGING (OPDD) [2] and present it in Figure 3.6. The key changes from the original *ddmin* presented in Figure 2.2 are highlighted. Intuitively, the major difference in OPDD is in the handling of reduce to complement cases. Instead of recursively restarting the reduction process for each reduce to complement opportunity, OPDD successively tries removing each remaining *unvisited* subset from the current minimal test case.

input : $\tau_{\text{✗}}$ and ψ with $\psi(\tau_{\text{✗}}) = True$
output : $\tau_{\text{✗}}'$ such that $\tau_{\text{✗}}' \subseteq \tau_{\text{✗}}$ and $\psi(\tau_{\text{✗}}') = True$
$opdd(\tau_{\text{✗}}) = opdd_2(\tau_{\text{✗}}, 2)$ where:

$$opdd_2(\tau_{\text{✗}}', n) = \begin{cases} opdd_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \mid \psi(\Delta_i) = True \\ refine(foldl(complement?, (\tau_{\text{✗}}', n), [1, .., n])) & \text{else if } \exists i \in \{1, \ldots, n\} \mid \psi(\nabla_i) = True \\ refine(\tau_{\text{✗}}', n) & \text{else} \end{cases}$$

$$complement?((\tau_{\text{✗}}', n), i) = \begin{cases} (\tau_{\text{✗}}' - \Delta_i, n-1) & \text{if } \psi(\tau_{\text{✗}}' - \Delta_i) = True \\ (\tau_{\text{✗}}', n) & \text{else} \end{cases}$$

$$refine(\tau_{\text{✗}}', n) = \begin{cases} opdd_2(\tau_{\text{✗}}', \min(|\tau_{\text{✗}}'|, 2n)) & \text{if } n < |\tau_{\text{✗}}'| \\ \tau_{\text{✗}}' & \text{else} \end{cases}$$

Figure 3.6: The ONE PASS DELTA DEBUGGING algorithm [2].

This is captured by applying a standard left fold (*foldl*) of *complement*? on the list of remaining subsets in Figure 3.6. *complement*? simply tries to remove a particular subset from $\tau_{\text{✗}}'$ and returns the smaller test when it still reproduces the failure. *foldl* is a higher-order function that combines the results of *complement*? executions in order from left to right. After applying the left fold, OPDD calls *refine* on the result to either refine (increase) the granularity or finish. This avoids a recursive invocation of $opdd_2$ that would reconsider removing particular subsets from the test case again.

Correctness. We consider a test case reduction algorithm correct when it is guaranteed to produce a test case that (1) induces the failure and (2) is not larger than the original test case. OPDD is an *anytime algorithm* that preserves correctness during all its executed steps meaning that in any step, the current version of the test case induces the failure. This holds regardless of whether the conditions described in Section 3.1 are satisfied or not.

Proof by induction: In the first step of the algorithm, the original test case triggers the failure. If the test case still induces the failure in step k, the test case in step $k+1$ either induces the failure or not. If it does, test case in step $k+1$ is trivially failure-inducing. If it does not, OPDD does not update the test case since a $True$ outcome of an oracle is used as an invariant in OPDD for identifying failure-inducing test cases. Hence, the test case in step $k+1$ will remain the same as the one in step k. □

Time Complexity. The interesting measure of algorithmic complexity is how the number of oracle queries grows with the size of the test case. Just like the original Delta Debugging, OPDD performs a logarithmic number of granularity refinements with a constant amount of work for each partition at a particular granularity that is querying the oracle for each partition and its complement. As a result, the worst case number of oracle queries of OPDD is $O(n)$ where n is the size of the test case. In contrast, the revisiting of original Delta Debugging leads to quadratic complexity in the worst case as observed in Figure 3.1. In the best case scenario, both Delta Debugging and OPDD have logarithmic time complexity because both algorithms in that case always perform a reduce to subset by dividing the test case into halves. In practice, the closer the real-world cases are to the worst case scenario, the potential improvement in efficiency, brought by OPDD, becomes more noticeable. The next section compares the performance of OPDD against original Delta Debugging for real-world test cases and bugs in our benchmark presented in Section 2.6.1.

3.3 Evaluation

In this section, we explore the performance characteristics of One Pass Delta Debugging in practice with a particular eye to the conditions and properties mentioned so far: common dependence order, unambiguity and deferred removal. We implement OPDD on Perses infrastructure by replacing its original Delta Debugging with our proposed variant of Delta Debugging. We observe that not only can we pragmatically exploit common dependence order, unambiguity and deferred removal, but they can significantly impact the measured running time of test case reduction in practice.

We continue this section by exploring the following research questions:

- **RQ1.** Is there a common dependence order among elements of test cases in practice?
- **RQ2.** Does empirical evidence suggest practical satisfaction of unambiguity?
- **RQ3.** What is the performance of OPDD in terms of metrics introduced in Section 2.6.3 that are reduced test case size, reduction time, the number of oracle queries and the reduction efficiency?
- **RQ4.** How can satisfaction of deferred removal play a role in OPDD test case reduction process? Does it help OPDD to generate reduced test cases comparable in size to those generated by original Delta Debugging?

In the following, we answer each one of the above questions by conducting explanatory and evaluation experiments.

3.3.1 RQ1. Common Dependence Order in Practice

As described previously in this chapter, if the preferred dependence order among elements of a test case is known, complement tests can be ordered in a way such that performing revisiting becomes unnecessary. Here, we perform a study on our smaller sample benchmark to examine whether a common dependence order among their elements exists in practice.

To conduct this study, we consider two natural orderings for subset removals:

1. *Forward* (→) that is removing subsets from the beginning of the test case to the end.
2. *Backward* (←) that is removing subsets from the end of the test case to the front.

Next, we measure the number of *successful revisiting subset removals* for each ordering when running the original Delta Debugging algorithm. Recall that a successful subset removal means that the test case without the subset satisfies the oracle by triggering the failure. The number of successful revisiting subset removals is the number of subsets that were *not* successfully removed in the *first* round of running the oracle, but after updating the test case, they were successfully removed in a revisiting round. The idea is that the ordering with *fewer* successful revisiting subset removals is the preferred ordering. For instance in Figure 3.1, the preferred ordering is backward with no successful revisits while the forward ordering causes quadratic behavior.

The second and third columns of Table 3.1 depict the results of this experiment. As can be seen, for all the five test cases that have different number of successful revisiting subset removals for their orderings, the *backward* ordering has fewer number of successful revisits, making it the preferred ordering for our sample benchmark test cases. This is indeed expected when reducing compiler test cases. As an example, the use of a variable is usually after its declaration and should get removed first to make removal of the declaration possible.

This evidence can help us to infer that the common dependence order among programming language test cases is likely to be *backward*. Hence, if we remove subsets in backward order, we may not need to perform revisiting tests. However, we can see cases that even for backward dependence orders, successful revisits still occur. In particular, we can see successful removals in revisiting rounds for test cases `clang-25900`, `gcc-61383` and `rust-63791`.

As a result, we *cannot* conclude that if subsets of a test case are removed in backward order, then performing revisiting becomes completely unnecessary. We later show that deferred removal helps to address cases where a common dependence ordering fails.

Table 3.1: Number of successful revisiting subset removals in forward and backward orderings along with unambiguity satisfaction results for our sample benchmark.

Test Case	**# Successful revisits**		**# Granularities with successful removal**			**Unambiguity ratio (%)**
	Forward (→)	Backward (←)	trivial (single removal)	successful combination	unsuccessful combination	
clang-25900	5	1	25	22	0	100
clang-31259	7	0	24	20	0	100
gcc-60116	13	0	47	43	0	100
gcc-61383	36	1	21	44	0	100
gcc-77624	0	0	8	3	0	100
rust-63791	4	1	19	1	0	100
rust-65934	0	0	1	0	0	100
rust-77993	0	0	10	4	0	100
go-30606*	0	0	0	0	0	N/A

* This test case has no list-based (Kleene-Star and Kleene-Plus) successful reduction by Delta Debugging. Only single Optional children are removed.

3.3.2 RQ2. Unambiguity in Practice

As described earlier in Section 3.1.2, if test cases are unambiguous, the removal of different subsets should be independent from each other, making revisiting unnecessary. Thus, OPDD should produce the same results as original Delta Debugging.

To examine the occurrence of unambiguity in practice, for each granularity reached by Delta Debugging in test cases of our sample benchmark, we verify whether Definition 3.1.1 holds or not. Recall that Delta Debugging partitions a test case into n disjoint subsets where n defines the granularity. At any given granularity with test case variant $\tau = \cup_{i=1}^{n} s_i$, we compute $\psi(\tau - s_i)$ for all subsets individually (LHS of Definition 3.1.1), maintaining a list S of subsets that have successfully been removed from τ. Next, when all subsets are examined individually, we construct a cumulative test case T, consisting of subsets in S and then we compute $\psi(\tau - T)$ (RHS of Definition 3.1.1). If $\psi(\tau - T) = True$, then unambiguity holds at the given granularity with respect to subsets in S. Otherwise, unambiguity does not hold.

For each test case, we compute the total number of granularities with at least one individual successful subset removal that were reached and analyzed during complement tests of Delta Debugging. Note that unambiguity *trivially holds* for granularities with exactly *one* successful subset removal (the fourth column in Table 3.1). The fifth and sixth columns show the number of granularities with at least two individual successful removals. As can be seen, unambiguity strongly holds for all test cases since no unsuccessful removal of combined subsets in S occurs in our sample benchmark. This evidence strongly suggests that test cases are likely to be unambiguous in practice.

3.3.3 RQ3. OPDD Performance

So far, we observed that both common dependence order and unambiguity have some empirical support in our suite of sample test cases. This provides some intuition that OPDD may be successful in practice. To verify, we implement One Pass Delta Debugging to compare its performance against original Delta Debugging with respect to our performance metrics presented in Section 2.6.3. We run this study on our full set of benchmark with 28 programs and one XML file. Again, we build both original Delta Debugging and OPDD on the infrastructure of Perses and select the backward ordering for removing subsets in both since as observed in Section 3.3.1, backward ordering may be the preferred ordering for reducing program source code.

Results are shown in Table 3.2. The best values for each metric are highlighted for each test case. Interestingly, for all the test cases except for one (`rust-63791`), Perses with OPDD generates a reduced test case with *exactly the same size* of a test case produced by Perses with original Delta Debugging. For `rust-63791` that is the only test case that OPDD generates a larger output, the reducer times out after 4 hours. It is possible that the generated outputs by Perses DD and Perses OPDD would have the same size if unlimited reduction time was available.

These results strongly suggest that skipping revisiting tests *does not* harm the reduction power and effectiveness in practice. Moreover, we can see that the efficiency of Perses improves when revisiting tests are skipped using OPDD. In further detail, for C test cases on average, the number of oracle queries, the reduction time and reduction efficiency are 3,007 queries, 876 seconds and 31.56 removed tokens per second, respectively. These numbers are 3,264 oracle queries, 1,013 seconds of reduction time and 27.29 removed tokens per second for Perses with original Delta Debugging. This is an average improvement of 14% for reduction time of C test cases. We further compute a speed up metric by dividing the reduction time of Perses DD by the reduction time of Perses OPDD. The average speed up achieved by OPDD for C test cases is 1.16.

Similarly, OPDD improves the efficiency of reduction on Rust and Go test cases although this improvement is less noticeable. One reason is that the original size of the test cases in Rust and Go are much smaller than C test cases. As a result, there is less room for improvement when reducing test cases of these domains. However, OPDD still performs reduction faster than original Delta debugging for all three Go programs and similarly, for 6 out of 8 Rust test cases. To explain the only case in which reduction does not time out and Perses DD performs better (`rust-77002`), we can consider multiple reasons. One of them is the possibility of having *multiple minimal test cases*. As explained in Section 2.1, finding a

Table 3.2: A comparison between the performance of Perses using original Delta Debugging (Perses DD) and Perses using One Pass Delta Debugging (Perses OPDD). Node replacements disabled.

Test Case	$O(\#)$	Perses DD				Perses OPDD				Speed Up ($\times$)
		$R(\#)$	$Q(\#)$	$T(s)$	$E(\#/s)$	$R(\#)$	$Q(\#)$	$T(s)$	$E(\#/s)$	
clang-22382	21,069	1,144	5,114	3,220	6.19	1,144	4,674	3,165	6.30	1.02
clang-22704	184,445	746	4,104	1,645	111.67	746	3,728	1,423	129.09	1.16
clang-23309	38,648	2,321	9,173	2,882	12.60	2,321	8,659	2,667	13.62	1.08
clang-25900	78,961	798	4,205	1,393	56.11	798	3,777	1,217	64.23	1.14
clang-27747	173,841	612	3,743	1,613	107.40	612	3,411	1,374	126.08	1.17
clang-31259	48,800	920	4,189	1,605	29.83	920	3,805	1,524	31.42	1.05
gcc-59903	57,582	1,879	10,477	4,040	13.79	1,879	10,010	3,732	14.93	1.08
gcc-60116	75,225	1,281	9,066	4,706	15.71	1,281	8,392	4,496	16.45	1.05
gcc-61383	32,450	1,287	6,161	1,454	21.43	1,287	4,717	1,166	26.73	1.25
gcc-61452	26,733	1,023	5,548	3,563	7.22	1,023	5,165	3,515	7.31	1.01
gcc-61917	85,360	1,401	8,132	1,905	44.07	1,401	7,510	1,718	48.87	1.11
gcc-64990	148,932	1,120	6,350	1,932	76.51	1,120	5,859	1,735	85.19	1.11
gcc-65383	43,943	1,121	5,850	1,628	26.30	1,121	5,395	1,447	29.59	1.13
gcc-66186	47,482	1,299	4,889	1,244	37.12	1,299	4,539	1,129	40.91	1.10
gcc-71626	6,134	61	474	38	159.82	61	446	34	178.62	1.12
gcc-71632	141	82	190	62	0.95	82	190	60	0.98	1.03
gcc-77624	1,306	23	99	13	98.69	23	97	5	256.60	2.60
geomean	29,769	654	3,264	1,013	27.29	654	3,007	876	31.56	1.16
median	47,482	1,120	5,114	1,628	29.83	1,120	4,674	1,447	31.42	1.11
rust-44800	802	472	2,113	11,907	0.03	472	2,102	11,208	0.03	1.06
rust-63791	8,144	5,768	3,873	T/O	0.17	5,809	4,126	T/O	0.16	1.00
rust-65934	107	100	148	86	0.08	100	148	76	0.09	1.13
rust-69039	191	128	428	2,275	0.03	128	428	2,265	0.03	1.00
rust-77002	348	302	818	1,846	0.02	302	810	2,195	0.02	0.84
rust-77993	4,989	48	174	181	27.30	48	169	160	30.88	1.13
rust-78336	980	18	117	504	1.91	18	109	487	1.98	1.03
rust-78622	157	29	78	335	0.38	29	74	282	0.45	1.19
geomean	659	151	401	1,083	0.22	151	396	1,037	0.23	1.04
median	575	114	301	1,175	0.13	114	299	1,341	0.13	1.05
go-28390	146	84	212	33	1.88	84	211	30	2.07	1.10
go-29220	127	74	98	12	4.42	74	98	8	6.63	1.50
go-30606	449	423	629	155	0.17	423	629	149	0.17	1.04
geomean	203	138	236	39	1.12	138	235	33	1.33	1.20
median	146	84	212	33	1.88	84	211	30	2.07	1.10
urls.xml	679	30	37	2	324.50	30	37	2	324.50	1.00

O, R and Q denote number of tokens in the original test case, reduced one and total number of oracle queries performed by the reduction technique, respectively. T is the reduction time in seconds and E is the efficiency in terms of the number of tokens removed per second. The speed up is calculated by dividing the reduction time of Perses DD by the reduction time of Perses OPDD. Timeout (T/O) is set to 4 hours.

unique global minimum for large test cases is infeasible. As a result, different reducers such as original Delta Debugging and OPDD that have different search spaces may generate a different minimal test case or explore different candidates with varying oracle verification time. Hence, it is possible that a reducer's behavior in a few specific test cases in practice does not correspond to its predicted theoretical improvement. We discuss this possibility further in Section 6.2 and conduct a study in Section 4.5.3 to examine the oracle verification time and its impact on the overall reduction process. Finally, although we are running all our experiments on the same environment and try to minimize factors that can adversely affect the validity of our results, it may not be possible to prune all the potential risk factors.

We discuss this further again in Section 6.2. In general, based on the collected results for our benchmark, we believe that our One Pass Delta Debugging improves the efficiency of original Delta Debugging without harming its effectiveness.

Table 3.3: A comparison between the performance of Perses using original Delta Debugging (Perses DD) and Perses using One Pass Delta Debugging (Perses OPDD). Node replacements enabled.

Test Case	$O(\#)$	**Perses DD**				**Perses OPDD**				**Speed Up** $(\times)$
		$R(\#)$	$Q(\#)$	$T(s)$	$E(\#/s)$	$R(\#)$	$Q(\#)$	$T(s)$	$E(\#/s)$	
clang-22382	21,069	334	5,030	3,535	5.87	334	4,656	3,474	5.97	1.02
clang-22704	184,445	266	4,289	1,788	103.01	266	3,950	1,489	123.69	1.20
clang-23309	38,648	237	6,747	2,229	17.23	237	6,365	2,095	18.33	1.06
clang-25900	78,961	304	4,578	1,660	47.38	304	4,205	1,493	52.68	1.11
clang-27747	173,841	220	4,311	1,624	106.91	220	3,974	1,495	116.13	1.09
clang-31259	48,800	374	5,312	2,424	19.98	374	4,920	2,161	22.41	1.12
gcc-59903	57,582	465	9,908	4,572	12.49	465	9,680	4,489	12.72	1.02
gcc-60116	75,225	480	7,514	9,088	8.22	480	7,129	8,905	8.39	1.02
gcc-61383	32,450	321	5,649	1,805	17.80	321	4,487	1,472	21.83	1.23
gcc-61452	26,733	370	5,552	3,993	6.60	370	5,238	3,876	6.80	1.03
gcc-61917	85,360	327	5,869	1,829	46.49	327	5,399	1,652	51.47	1.11
gcc-64990	148,932	354	5,873	2,288	64.94	354	5,489	2,112	70.35	1.08
gcc-65383	43,943	228	5,383	1,819	24.03	228	5,002	1,723	25.37	1.06
gcc-66186	47,482	441	4,692	1,563	30.10	441	4,392	1,403	33.53	1.11
gcc-71626	6,134	60	535	36	168.72	60	507	35	173.54	1.03
gcc-71632	141	75	217	57	1.16	75	217	57	1.16	1.00
gcc-77624	1,306	22	196	10	128.40	22	194	13	98.77	0.77
geomean	29,769	232	3,315	1,119	25.27	232	3,093	1,058	26.72	1.06
median	47,482	321	5,312	1,819	24.03	321	4,656	1,652	25.37	1.06
rust-44800	802	464	3,057	T/O	0.02	464	2,728	T/O	0.02	1.00
rust-63791	8,144	6,203	3,395	T/O	0.13	6,269	3,285	T/O	0.13	1.00
rust-65934	107	100	238	510	0.01	100	238	559	0.01	0.91
rust-69039	191	120	826	4,372	0.02	120	826	4,248	0.02	1.03
rust-77002	348	286	3,900	11,018	0.01	286	3,892	9,650	0.01	1.14
rust-77993	4,989	16	175	705	7.05	16	170	786	6.33	0.90
rust-78336	980	15	106	438	2.20	15	98	379	2.55	1.16
rust-78622	157	29	77	319	0.40	29	73	310	0.41	1.03
geomean	659	127	571	2,176	0.12	127	550	2,139	0.12	1.02
median	575	110	532	2,539	0.08	110	532	2,517	0.08	1.02
go-28390	146	84	245	35	1.77	84	244	35	1.77	1.00
go-29220	127	60	165	16	4.19	60	165	18	3.72	0.89
go-30606	449	233	912	204	1.06	233	912	203	1.06	1.00
geomean	203	106	333	49	1.99	106	332	50	1.91	0.96
median	146	84	245	35	1.77	84	244	35	1.77	1.00
urls.xml	679	9	21	1	670	9	21	1	670	1.00

O, R and Q denote number of tokens in the original test case, reduced one and total number of oracle queries performed by the reduction technique, respectively. T is the reduction time in seconds and E is the efficiency in terms of number of tokens removed per second. The speed up is calculated by dividing the reduction time of Perses DD by the reduction time of Perses OPDD. Timeout (T/O) is set to 4 hours.

The results in Table 3.2 are collected when nodes in the parse trees of test cases are only tried for removal. Table 3.3 depicts the results of reduction when both node removal and node replacement trials as described in Section 2.4.1 are enabled. These results also show the same improvement caused by OPDD. Note that the reduced sizes in Table 3.3 are smaller than Table 3.2 due to more reduction operations performed by both Perses DD and Perses OPDD.

Finally, it is worth mentioning that the improvement caused by OPDD may be more significant when reducing larger lists of nodes with higher likelihood of unsuccessful tests. These lists are likely to be closer to the worst case scenario of original Delta Debugging and provide more opportunity for boosting their reduction when OPDD is used. We observed this in one of our previous works where we applied original Delta Debugging and OPDD on the Hierarchical Delta Debugging (HDD) framework [2]. Since HDD applies Delta Debugging on all the nodes present at a level, it generally attempts to reduce longer lists compared to Perses that applies Delta Debugging on lists of children of quantified nodes. Moreover, HDD is more likely to fail due to trying to remove irrelevant nodes of a level together [17]. As a result, HDD can provide more room for improvement [2].

In this dissertation, our focus is on the latest state of the art test case reduction techniques, such as Perses, that operate by traversing the parse tree in a more sophisticated fashion. However, regardless of the reduction approach and the degree of significance, One Pass Delta Debugging can enhance the performance of Delta Debugging and speed up any framework that is built upon it.

Next, we describe a study to explain the third property introduced in this chapter, the deferred removal property, in practice.

3.3.4 RQ4. Deferred Removal in Practice

By exploring our first research question in Section 3.3.1, we learned that backward ordering is likely to be the preferred ordering for removing subsets from test cases of programming language domain in practice. By choosing this ordering and applying One Pass Delta Debugging on our full set of benchmark in Section 3.3.3, we observed improvement in efficiency of test case reduction while producing reduced outputs of *the same* size compared to original Delta Debugging for most of the cases. However, we also observed in Table 3.1 that successful revisiting subset removals still occur in backward ordering for a few test cases in practice. This means that if we do not perform revisiting tests, we may miss some reduction opportunities even when removing subsets is performed in backward order. However, we mainly see the same output results for both One Pass Delta Debugging and original Delta Debugging? Why?

To answer the above question, this section evaluates a third property called *deferred removal*. We have previously provided the definition and details of this property in Section 3.1.3. In a nutshell, if this property holds, it means that removable elements skipped by OPDD in a granularity are likely to be captured and removed in subsequent finer granularities. The satisfaction of this property can assist in obtaining outputs of comparable size by One Pass Delta Debugging even when common dependence order or unambiguity properties do not completely hold in practice.

To examine whether deferred removal holds in practice, we select the three test cases in Table 3.1 that have successful revisiting subset removal in their backward ordering reduction.

These test cases are `clang-25900`, `gcc-61383` and `rust-63791`, each one of them with exactly one successful revisiting test when removing their subsets in backward order. Based on the results in Table 3.2, original Delta Debugging and OPDD generate outputs of the same size for the first two test cases while OPDD produces a slightly larger output for the third test case within the reducer's 4 hour timeout. For these three cases, we set up an experiment to record *when* (i.e., *at which granularity* in reducing a list) each unique element (here node in the parse tree) is removed from the test case in the reduction process of both OPDD and original Delta Debugging.

We observe three different interesting behaviors of OPDD for the three test cases:

clang-25900. In this case, OPDD removes the elements of the subset removed in a revisiting round by the original Delta Debugging algorithm at the immediate next finer granularity.

gcc-61383. In this case, original Delta Debugging removes a subset of size two in a revisiting round. One of the elements (nodes) of this subset is captured and removed in the immediate next granularity by OPDD. The other remaining node is removed when performing fixed point iterations of reduction. Details on fixed point rounds are provided in Section 2.3.2.

rust-63791. In this case, the elements of subsets skipped by OPDD are not captured and removed within the 4 hour time out. This can explain the slightly larger output generated by OPDD for this specific test case.

Based on the observations above, we can see that deferred removal occurs in practice. However, in the case that deferred removal does not completely hold, the fixed point iterations of tree based reduction can indeed help OPDD to generate outputs of the same or similar size compared to original Delta Debugging. These results indicate that in practice, we do not necessarily require a strict dependence order among elements of a test case or full unambiguity to be able to skip subset revisits without adversely affecting the performance of Delta Debugging in terms of the size of the reduced test case.

Finally, it is worth mentioning that although postponing the successful removal of a subset to a finer granularity may increase the number of tests by dividing a subset into two in the following granularity, sparsity of successful subset revisits in addition to larger number of unsuccessful revisits that are skipped should decrease the overall number of tests in practice as seen in Table 3.2 and Table 3.3.

3.4 Summary

In this chapter, we proposed a variant of the Delta Debugging algorithm with better theoretical time complexity and higher efficiency in practice. This variant called One Pass Delta Debugging (OPDD) has a *linear* worst case time complexity in contrast to the original Delta Debugging algorithm with *quadratic* worst case time complexity. By skipping tests that are less likely to succeed based on a simple observation of prior tests, OPDD performs only one pass of complement tests at each granularity. In our evaluation set, both OPDD and original Delta Debugging generated outputs of the same size although OPDD performed fewer tests within shorter reduction time. To explain this behavior, we investigated two properties existing in the literature called common dependence order and unambiguity, and proposed a new third property called deferred removal. We demonstrated that empirical satisfaction of these properties can help OPDD to achieve reduced test cases of the same or similar size as the original Delta Debugging. Reduction techniques using Delta Debugging and infrastructures built upon it can directly benefit from using our One Pass Delta Debugging technique by replacing their original Delta Debugging mechanism with OPDD. In Perses infrastructure, replacing Delta Debugging with our OPDD version led to an average speed up of 1.16x, 1.04x, and 1.20x when reducing C, Rust, and Go programs of our benchmark, respectively while generating outputs of the same size.

Chapter 4

Priority Aware Test Case Reduction

This chapter addresses the second problem in our list of problems in Section 2.5 with a focus on problem 2.1 and explores the following question:

Can we mitigate the problem of *priority inversion* in Perses by devising a *new* priority aware reduction technique that is capable of reducing the more beneficial portions of a test case at an early stage?

We originally introduced and published the ideas presented in this chapter at the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE 2019) [28].

4.1 Priority Inversion

Recall the background information on Perses, the latest state of the art domain agnostic program reducer in Section 2.4.1. In spite of using a priority queue of nodes in the parse tree of the program to reduce, Perses still suffers from a drawback in its priority mechanism that we call *priority inversion*. In general, priority inversion occurs when a low priority task is scheduled instead of a high priority task. In case of program reduction by Perses, priority inversion occurs when significant time may be spent trying to perform reduction operations on portions of the parse tree that will not have a large impact on the size of the reduced test case.

To better understand the concept of priority inversion, consider a quantified Kleene-Star (*) node with a large number of tokens as descendants that has multiple children, maybe hundreds such that each child has a small number of token descendants. Figure 4.1 depicts a simplified version of this scenario. As mentioned earlier in Section 2.4.1, Perses tries to remove the children of the quantified node by applying Delta Debugging on them.

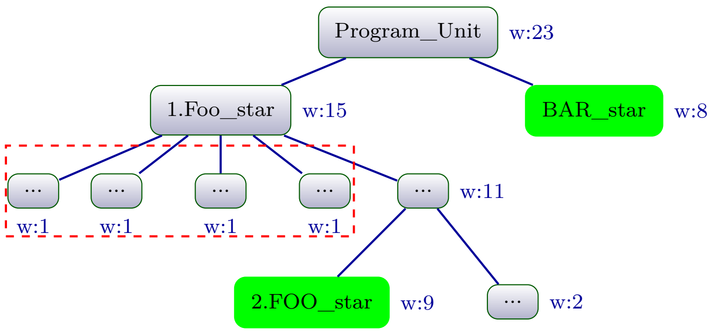


Figure 4.1: Priority inversion, an inefficient reduction prioritization by Perses: Perses spends reduction effort on the red portion while the green nodes have larger impact (number of token descendants) and should be prioritized.

Reducing a large list of children for a quantified node may take a long time due to the following reasons:

1. As mentioned in Section 2.2, Delta Debugging has a quadratic worst case time complexity with respect to the size of the list being reduced. A large number of children in the list can make the quadratic time complexity of Delta Debugging inefficient.

2. Reducing a child node with a small number of tokens removes a small portion of the test case, generating a possibly large remaining variant to be tested against an oracle. An oracle call on a larger variant is likely to be more expensive than a smaller variant due to multiple verification steps within the oracle, including the potential need to compile the variant. Moreover, removing a small portion is less likely to enable the removal of other portions.

Although our One Pass Delta Debugging proposed in Chapter 3 can mitigate the first problem, the second problem still remains. For example in Figure 4.1, while Perses spends a lot of time applying either original Delta Debugging or OPDD on children of `1.FOO_star` with small number of tokens (represented by w), there are `2.FOO_star` and `BAR_star` with larger number of tokens that their reduction is postponed by Perses.

Program p in Listing 4.1 and its parse tree in Figure 4.2 demonstrate the details of priority inversion. Suppose that the property of interest to preserve is printing "Hello World!" on line 14.

To search for a smaller program with this property of interest, Perses traverses the parse tree using a priority queue ordered by the number of token descendants of each node. Henceforth, we refer to this number as the *token weight* or simply *weight* of a node. For each test, Perses removes the node with the *maximum* weight from the work queue.

The queue starts out containing only the root node of the parse tree. Recall that Perses performs specific reduction operations on different types of nodes during traversal as explained in Section 2.4.1. For instance, on optional nodes, Perses tries to remove the optional

```
 1  int d = 10;
 2  struct S {
 3      int f1;
 4      int f2;
 5  };
 6  void foo() {
 7     struct S s1 = {1, 2};
 8     int i = 0;
 9     bool increment = true;
10     if (increment) {
11         i += 2*i + i + 1;
12     }
13     s1.f1 = d*i;
14     printf("Hello World!\n");
15  }
16  int main() {
17      foo();
18      return 0;
19  }
```

Listing 4.1: A C program with a statement to preserve on line 14.

child node. For list nodes, Perses minimizes the list of children using Delta Debugging. Any *remaining* children of the traversed node are then added to the priority queue in order to be traversed in the future.

In our example, Perses first removes the root node ① from the queue. Because ① is a quantified Kleene-Star node, its children are syntactically removable. Hence, Perses applies Delta Debugging on the list of children of ①. Different combinations of children are removed from ① and the result is checked by an oracle ψ to find a smaller program. First,

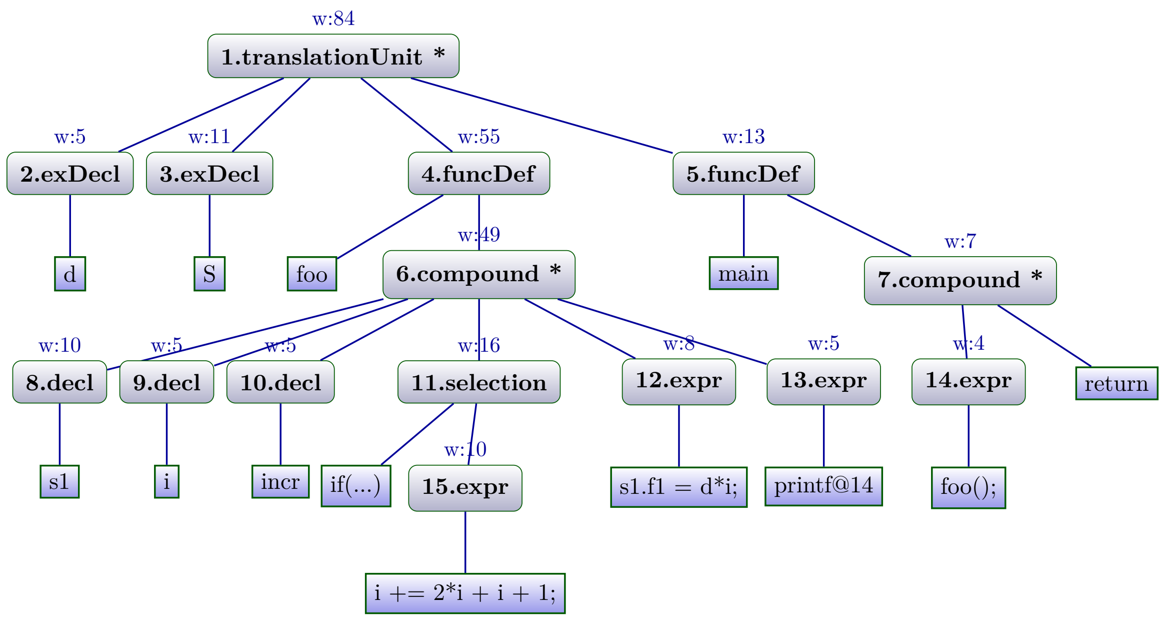


Figure 4.2: The simplified parse tree of program in Listing 4.1. Each internal node is annotated with an ID and its grammar rule type. * denotes a quantified Kleene-Star node.

node(s) to remove	ψ
{2,3,4,5}	False
{2,3}	False
{4,5}	False
{3,4,5}	False
{2,4,5}	False
{2,3,5}	False
{2,3,4}	False
{2}	False
{3}	False
{4}	False
{5}	False
{8,9,10,11,12,13}	False
{8,9,10}	False
{11,12,13}	False
{10,11,12,13}	False
{8,9,12,13}	False
{8,9,10,11}	False
{8,9}	False
{10,11}	True
{12,13}	False
{9,12,13}	False
{8,12,13}	False
{8,9,13}	False
{8,9,12}	True
{13}	False
{14}	False

(a) Perses

node to remove	ψ
{1}	False
{4}	False
{6}	False
{11}	True
{5}	False
{3}	False
{8}	False
{12}	True
{7}	False
{13}	False
{10}	True
{9}	True
{2}	True
{14}	False

(b) PARDIS

node(s) to remove	ψ
{1}	False
{4}	False
{6}	False
{11}	True
{5}	False
{3}	False
{8}	False
{12}	True
{7}	False
{13}	False
{10,9}	True
{2}	True
{14}	False

(c) PARDIS HYBRID

Figure 4.3: One round of removal tests in Perses, PARDIS and PARDIS HYBRID for the parse tree in Figure 5.2. Numbers are node IDs.

all children are removed and ψ is checked. After this fails, the first half of the children, nodes ② and ③ are removed, but ψ returns *False* because this removes required external declarations. Since removing the second half of the children, nodes ④ and ⑤ also fails, the process continues recursively. First, Delta Debugging tries shrinking the list by *only keeping* each individual child (i.e., subset tests in Delta Debugging), and next it tries *removing* each individual child (i.e., complement tests). Ultimately, none of the tests succeed, so all children are added to the queue, and reduction continues with node ⑥ that has the next largest weight. This node is a list node and Perses applies Delta Debugging on the list of its children. This process continues until the queue becomes empty or the reducer times out. The precise tests exercised in this process are illustrated in Figure 4.3 (a). Note that in spite of applying our more efficient version of Delta Debugging, OPDD, on the lists of children in this example, 19 steps elapse until a successful test occurs when using Perses. This can lead to inefficiency in test case reduction.

Here, the problem is that while the priorities used by Perses are controlled by the token weight, they determine how the *children* of the traversed nodes are removed. Thus, any node whose *parent* in the parse tree is a quantified node is given the same priority as all other elements in the list. This is because Delta Debugging recursively tries to minimize the *entire* list until no single element can be removed, regardless of the priorities of individual list elements. As a result, Perses must employ Delta Debugging on the entirety of the children of ① even though it would be more beneficial to focus on just one child, node ④.

To mitigate this problem, we propose Pardis [28], a test case reduction technique that more directly models the priorities. We note that in a quantified node, such as node ①, each child may be removed in a syntactically valid fashion. In other words, based on the grammar, children of a quantified node are syntactically removable, independent from each other. As a result, when traversing a syntactically removable node in the parse tree, we can simply try directly to remove it, adding its children if the removal fails.

For instance, in the running example, we would visit node ① first. Because ① cannot get removed, we would simply add its children to the priority queue. Note that all children of ① are syntactically removable, but ④ has the largest token weight. Thus, we next select ④ to traverse but removing ④ also fails. Children of ④ are added to the queue and from the given token weights, we next attempt to remove node ⑥[1] and after it fails, we add its children to the queue. Then, node ⑪ is visited and successfully removed. Next, Pardis tries to remove nodes ⑤, ③ and ⑧ in order and fails but node ⑫ successfully gets removed. Removing node ⑫ *enables the removal of* node ② when it is first visited by Pardis whereas Perses would require multiple traversals of the parse tree to remove it.

Similar to Perses, the process of selecting nodes and reducing them by Pardis continues until the queue becomes empty or Pardis times out. As can be seen in Figure 4.3 (b), just 4 steps elapse until the first successful test removes node ⑪ by Pardis.

This new priority aware approach can still have drawbacks, however. After focusing on the highest priority nodes, there may be many lower priority nodes remaining. For example, there are multiple remaining nodes of weight 5 in the tree of Figure 4.2. The above approach of Pardis considers each node *one at a time*, which can have poor performance when reducing such long lists. In our example, Pardis performs 4 tests to remove nodes ⑪, ⑫, ⑩ and ⑨ one by one. However, Perses is able to remove these nodes by performing only two tests due to its capability of removing groups of nodes together.

To enable performing tests on groups of nodes in our new technique, we also propose a *hybrid* version of Pardis that still prioritizes nodes by maximum token weight but additionally makes use of a list based reduction technique for spans of nodes that have *the same* parent and token weight. This hybrid approach named Pardis Hybrid is able to achieve the benefits of being priority aware while still avoiding the cost of considering each node of the parse tree individually. In our running example as shown in Figure 4.3 (c), Pardis Hybrid can remove nodes ⑨ and ⑩ together by performing only a single test. In Section 4.5, we show experimentally that these lower priority nodes occur in practice and that Pardis

[1]⑥ is a quantified node with *all* its children being syntactically removable. As a result, ⑥ itself is syntactically removable too.

Hybrid can be beneficial for some cases. The next sections present the algorithms behind our techniques in detail.

4.2 Pardis: The Algorithm

Similar to Perses, the core of Pardis maintains a priority queue of the nodes in a parse tree and traverses the nodes in order to process them. It also makes use of Perses Normal Form (PNF) grammar resulted by grammar transformations performed by Perses and discussed in Section 2.4.1. The key difference is that instead of using the token weight of a *parent* node to determine when its syntactically removable children may be removed, Pardis identifies *all* syntactically removable nodes and uses their token weights directly to prioritize the search. The core algorithm for this process is quite straightforward and presented in algorithm 2.

Algorithm 2: Pardis: Priority aware queue driven test case reduction [28].

```
Input: $\tau_{\text{✗}}$ – The test case to reduce as a parse tree
Input: $\psi : \mathbb{S} \to \mathbb{B}$ – Oracle for the property to preserve where $\mathbb{S}$ is the search space and
        $\psi(\tau_{\text{✗}}) = True$
Input: $\rho : \mathbb{V} \to \mathbb{N} \times \cdots \times \mathbb{N}$ – Prioritizer for tree nodes
Result: A minimum test case $\tau_{\text{✗}}' \subseteq \tau_{\text{✗}}$ s.t. $\psi(\tau_{\text{✗}}') = True$
1 $\tau_{\text{✗}}' \leftarrow \tau_{\text{✗}}$;
2 work $\leftarrow$ MaxPriorityQueue({$\tau_{\text{✗}}'$.root}, $\rho$);
3 while !work.empty() do
4     node $\leftarrow$ work.takeMax();
5     if node.isSyntacticallyRemovable() && $\psi(\tau_{\text{✗}}' - node)$ then
6         $\tau_{\text{✗}}' \leftarrow \tau_{\text{✗}}'$ - node;
7     else
8         work.insert(node.children);
9 return $\tau_{\text{✗}}'$;
```

Line 2 of the algorithm constructs the priority queue (a max-heap), initializing it with the root of the parse tree and using a parameterizable priority ρ. ρ is simply a function that takes a node and returns its priority as a tuple. The priority queue selects the element with a lexicographically maximal priority, so ties on the *first* element of the priority tuple are broken by the *second* element and so on. As seen in Figure 4.4, for Pardis, ρ_{Pardis} returns a pair of numbers, the token weight of the node and the position of the node in a decreasing, right-to-left, breadth first search. The specific breadth first order means that for parse tree p with n nodes, bfsOrder(p.root)=n, the last child c of p.root has bfsOrder(c)=n-1, and so on. Thus, if several nodes have the same token weight, the one highest in the parse tree and furthest to the right is selected next. This ordering decreases the chances of trying to remove a declaration before its uses [48].

Line 3 starts the core of the algorithm. While there are more nodes to explore in the queue, the node with the next highest priority is considered. If it is syntactically removable

Definitions:

tokensBelow(n) – returns the number of tokens beneath a parse tree node n.

bfsOrder(n) – returns the position of a parse tree node n in a decreasing, right to left, breadth first search.

Prioritizers:

$$\rho_{\textsc{Pardis}} = (\mathsf{tokensBelow(n)},\ \mathsf{bfsOrder(n)})$$

$$\rho_{perses} = \begin{cases} \textbf{let}\ \mathsf{parentWeight} \leftarrow \mathsf{tokensBelow(n.parent)}\ \textbf{if}\ \mathsf{n.parent}\ \textbf{else}\ \infty\ \textbf{in} \\ \textbf{let}\ \mathsf{parentOrder} \leftarrow \mathsf{bfsOrder(n.parent)}\ \textbf{if}\ \mathsf{n.parent}\ \textbf{else}\ \infty\ \textbf{in} \\ (\mathsf{parentWeight},\ \mathsf{parentOrder}) \end{cases}$$

$$\rho_{\textsc{Pardis Hybrid}} = \begin{cases} \textbf{let}\ \mathsf{parentOrder} \leftarrow \mathsf{bfsOrder(n.parent)}\ \textbf{if}\ \mathsf{n.parent}\ \textbf{else}\ \infty\ \textbf{in} \\ (\mathsf{tokensBelow(n)},\ \mathsf{parentOrder},\ \mathsf{bfsOrder(n)}) \end{cases}$$

Figure 4.4: Prioritizers used for Pardis, node at a time Perses, and Pardis Hybrid.

and can be successfully removed based on the oracle's outcome, we remove it from the parse tree, otherwise we add its children to the queue so that they will also be traversed.

While the algorithm is surprisingly simple, we have found it to perform significantly better than the state of the art Perses in practice. As we explore in Section 4.5, this results from prioritizing the search towards those portions of the input where reduction can have the greatest impact.

To more closely compare with Perses, consider a version of Perses that upon visiting a list or optional node, it only tries removing each child of that node once[2]. This *one node at a time* variant of Perses can also be implemented using algorithm 2 by carefully choosing the priority formula ρ. Because Perses considers removing the *children* of the nodes it traverses, it actually prioritizes the work queue using the token weight of the *parent* rather than the token weight of syntactically removable nodes being considered for removal. This leads to the alternative prioritizer ρ_{perses} presented in Figure 4.4. Observe that all children of a list node receive the same token weight, that of the entire list. This can inflate the priority of some nodes in the work queue and leads to poor performance.

Intuitively, using Pardis can have the following important benefits:

1. Stalls in reduction from unsuccessful rounds of Delta Debugging can be mitigated.

2. By removing large portions of a test case earlier on, each oracle query to ψ can take less time because smaller inputs tend to be faster to check (e.g., compiling and verifying a smaller program is typically faster).

3. Removing large portions earlier on can also enable removal of other nodes, leading to a faster convergence towards the reduced test case.

[2]We compare against *both* versions of Perses in Section 4.5

Like other test case reduction algorithms [13, 15, 14, 16, 22], algorithm 2 is used to compute a fixed point. That is, in practice the algorithm is repeated until no further reductions can be made. As in prior works, we omit this from our presentation for clarity. In theory, this means that the worst case complexity of the technique is $O(n^2)$ where n is the number of nodes in the parse tree. This arises when only one leaf of the parse tree is removed in each pass through the algorithm. In practice, most nodes are not syntactically removable, and we show in Section 4.5 that performance of PARDIS exceeds the state of the art techniques.

4.3 Pardis Hybrid: The Algorithm

PARDIS, the initial priority aware technique from algorithm 2 can also encounter performance bottlenecks. The original motivation for using Delta Debugging on lists of children in the parse tree was that it tries to remove multiple children at the same time. Although the worst case time complexity of Delta Debugging is quadratic or linear for its original and OPDD versions, respectively, its best case time complexity is $O(log(n))$ where n is the number of children in the list. Processing one node at a time, however, requires that every list element is considered individually, guaranteeing $O(n)$ time for one round of algorithm 2. Priority aware reduction that proceeds one node at a time faces a different set of inefficiencies that can still cause stalls in the reduction process.

Thus, we desire a means of removing multiple elements from lists at the same time while *still* preserving priority awareness. In order to achieve this, we developed PARDIS HYBRID, as presented in algorithm 3.

Algorithm 3: PARDIS HYBRID algorithm with priority aware list reduction [28].

```
Input: $\tau_{\text{✗}}$ – The test case to reduce as a parse tree
Input: $\psi : \mathbb{S} \to \mathbb{B}$ – Oracle for the property to preserve where $\mathbb{S}$ is the search space and
       $\psi(\tau_{\text{✗}}) = True$
Input: $\rho : \mathbb{V} \to \mathbb{N} \times \cdots \times \mathbb{N}$ – Prioritizer for tree nodes
Result: A minimum test case $\tau_{\text{✗}}' \subseteq \tau_{\text{✗}}$ s.t. $\psi(\tau_{\text{✗}}') = True$
1 $\tau_{\text{✗}}' \leftarrow \tau_{\text{✗}}$;
2 work $\leftarrow$ MaxPriorityQueue($\{\tau_{\text{✗}}'$.root$\}$, $\rho_{\text{PARDIS HYBRID}}$);
3 while !work.empty() do
4     nodes $\leftarrow$ work.takeWithSameWeightAndParent();
5     removable, non-removable $\leftarrow$ partitionRemovable(nodes);
6     removed, retained $\leftarrow$ minimize($\tau_{\text{✗}}'$, removable, $\psi$);
7     $\tau_{\text{✗}}' \leftarrow \tau_{\text{✗}}'$ - removed;
8     work.insert($\bigcup_{x \in \text{retained} \cup \text{non-removable}}$ x.children);
9 return $\tau_{\text{✗}}'$;
```

PARDIS HYBRID uses a modified prioritizer as presented in Figure 4.4 by which it first orders nodes by their token weights, then by parent traversal order, and then by node traversal order. The effect this has is that all children of the same parent with the same weight are grouped together. As a result, we can remove them from the priority queue together and perform list based reduction (like Delta Debugging) to more efficiently remove groups of elements in a list that have the same priority. Because the search is still primarily directed by the nodes' token weights, it still fully respects the priority mechanism introduced in this chapter.

Similar to Pardis, line 2 of algorithm 3 starts by creating the priority queue. Note that it specifically uses the prioritizer $\rho_{\text{Pardis Hybrid}}$, which groups children having the same token weight in the priority queue. As long as there are more nodes to consider, line 4 takes all nodes from the queue with the same weight and parent. If the weight of a node is unique, this simply returns a list of length 1. Line 5 filters out syntactically non-removable nodes from the test, and line 6 just applies list based reduction to any syntactically removable nodes. Lines 7 and 8 then remove the eliminated nodes from the tree and add the children of remaining nodes to the work queue. Again, this algorithm actually runs to a fixed point.

As fully explained in Chapter 3, while the worst case behavior of original Delta Debugging is $O(n^2)$ [1], this can be improved to $O(n)$ by giving up hard *guarantees* on minimality [2]. Since this reduction process is performed to a fixed point anyway, minimize on line 5 can make use of our $O(n)$ approach to list based reduction, OPDD, without losing 1-minimality in practice. As a result, the time complexity of Pardis Hybrid is the same as Pardis.

4.4 Syntactical Removability Pruning

As a further optimization, we observed that many oracle queries performed by Perses, Pardis or Pardis Hybrid are simply unnecessary. Specifically, recall that a node can be tagged as syntactically removable because it is an element of a list or a child of an optional node, as previously defined by Perses grammar transformations in Perses Normal Form. The complete algorithm for this tagging is in *TagRemovable* of algorithm 4. However, for example, a list of one element could contain another list of one element. In the parse tree, this appears as a chain of nodes, at least two of which are syntactically removable. Removing *any one* of these nodes removes the same tokens from the parse tree. Thus, it is only necessary to select a single removable node from any *chain* of nodes, and the others can be disregarded.

We exploit this through an optimization called *syntactical removability pruning*. We traverse every chain of nodes in the parse tree, preserving the removability of the highest node in the chain and eliminating removability from those below it. The complete algorithm is presented in *PruneRemovable* of algorithm 4. In effect, it is just a depth first search that removes redundant removability from nodes along the way instantaneously.

We investigate the potential impact of this pruning on the actual reduction process in the next section.

Algorithm 4: Tagging and pruning syntactically removable nodes.

```
1  Function TagRemovable($\tau_{\chi}$)
       Input: $\tau_{\chi}$ – The test case to reduce as a parse tree
2      foreach Node $n \in \tau_{\chi}$ do
3          if $n \in KleeneStar \cup KleenePlus \cup Optional$ then
4              foreach $c \in n.children$ do c.isSyntacticallyRemovable ← true;
5  Function PruneRemovable($\tau_{\chi}$)
       Input: $\tau_{\chi}$ – The test case to reduce as a parse tree
6      Function OptimizeBelow(n)
7          hasRemovable ← false;
8          Loop
9              if hasRemovable then
10                 n.isRemovable ← false;
11             else if n.isRemovable then
12                 hasRemovable ← true;
13             if $1 \neq |n.children|$ then
14                 break;
15             n ← n.getOnlyChild();
16         foreach $c \in n.children$ do OptimizeBelow(n);
17     OptimizeBelow($\tau_{\chi}$.root);
```

4.5 Evaluation

In this section, we evaluate the performance of Pardis and Pardis Hybrid and examine the impact of priority inversion on reduction by answering the following research questions:

- **RQ1.** How do Pardis and Pardis Hybrid perform compared to Perses in terms of reduction time and efficiency, number of oracle queries, and size of the reduced test case?
- **RQ2.** How does priority inversion affect the reduction efficiency? In particular, does reduction require more work with a traversal order suffering from priority inversion?

4.5.1 RQ1. Performance: Pardis and Pardis Hybrid vs. Perses

To thoroughly examine the performance of reduction when implementing the algorithms proposed in this chapter, we define six different variants of reduction techniques. Each one of these variants applies a different reduction algorithm such that each new algorithm adds one difference to the previous one. All these variants compute fixed point rounds of reduction such that reducers keep traversing the parse tree for multiple rounds until the last round cannot remove anything from the test case. We define these variants as follows:

- Perses DD: The original algorithm of Perses that applies original Delta Debugging on lists of children of quantified nodes.
- Perses OPDD: The modified algorithm of Perses that applies One Pass Delta Debugging proposed in Chapter 3 on lists of children of quantified nodes.
- Perses N: The one node at a time Perses that *does not* apply any versions of Delta Debugging on list elements but tries to remove them *one by one* using Perses' parent oriented priorities.
- Pardis w/o Pruning: This variant uses the Pardis algorithm but does not apply the pruning optimization proposed in Section 4.4.
- Pardis Original: Our proposed algorithm in this chapter that also applies the syntactical removability pruning optimization.
- Pardis Hybrid: The hybrid version of Pardis with syntactical removability pruning and OPDD as its version of Delta Debugging.

For completeness, similar to the evaluation of One Pass Delta Debugging in Section 3.3, we implement the above variants once with removal as the only reduction operation and once with both removal and replacement as reduction operations to be performed on nodes of the parse tree. Our benchmark is the full set of test cases presented in Table 2.1 and the performance metrics are presented in Section 2.6.3 and are the same as the metrics in the previous studies.

Results are presented in Table 4.1 and Table 4.2. Again, the best values for each metric are highlighted for each test case. As can be seen in Table 4.1, for all the C test cases, either Pardis or Pardis Hybrid outperform all variants of Perses in terms of the reduction time, the total number of oracle queries and the reduction efficiency. With respect to efficiency for C test cases, Pardis and Pardis Hybrid remove 46.58 and 58.21 tokens per second on average, respectively while the best Perses variant with respect to this metric (i.e., Perses OPDD) removes only 31.56 tokens per second on average. Moreover, Pardis variants generate smaller outputs compared to Perses variants on average for C test cases.

For Rust test cases, Pardis has the shortest reduction time in 5 out of 8 test cases. Additionally for test case `rust-63791` that all reduction variants time out, Pardis performs the most reduction in the available time compared to the other variants. Overall, on average, for C test cases, Pardis Hybrid shows the best results while Pardis, the original version is the fastest when reducing Rust test cases. For Go test cases, Pardis and Pardis Hybrid perform equally well and surpass the other variants.

Similarly, either Pardis or Pardis Hybrid outperforms the other variants in Table 4.2 with an average number of 37.26, 0.14 and 2.43 tokens removed per second for C, Rust and Go test cases, respectively. Reducing the XML file is quite fast in all six variants. However, Pardis and Pardis Hybrid perform fewer number of tests compared to the other variants in this table.

The results across variants suggest that the improvements caused by Pardis and Pardis Hybrid arise from priority awareness and syntactical removability pruning. In particular, the difference between the performance of Pardis w/o Pruning and Pardis, the original version can demonstrate the impact of algorithm 4, the pruning optimization, on reduction speed. Moreover, the better performance of Pardis w/o Pruning in comparison to Perses N shows that the new priority mechanism defined by Pardis can indeed improve the efficiency of the reduction process. We further discuss this in Section 4.5.2.

Table 4.1: A comparison between different variants of Perses and Pardis. Node replacements disabled.

Test Case	$O_{(\#)}$	Perses DD				Perses OPDD				Perses N				Pardis w/o Pruning				Pardis				Pardis Hybrid			
		$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$
clang-22382	21,069	1,144	5,114	3,220	6.19	1,144	4,674	3,165	6.30	944	3,292	4,337	4.64	867	2,878	3,962	5.10	867	1,977	3,856	5.24	867	2,373	2,567	7.87
clang-22704	184,445	746	4,104	1,645	111.67	746	3,728	1,423	129.09	646	5,100	6,891	26.67	326	4,682	4,336	42.46	326	4,055	4,243	43.39	326	1,886	1,367	134.69
clang-23309	38,648	2,321	9,173	2,882	12.60	2,321	8,659	2,667	13.62	2,111	6,673	2,221	16.45	2,129	4,773	1,530	23.87	2,129	2,897	998	36.59	2,129	3,543	1,045	34.95
clang-25900	78,961	798	4,205	1,393	56.11	798	3,777	1,217	64.23	787	2,723	1,087	71.92	883	2,422	789	98.96	883	1,582	577	135.32	883	1,948	550	141.96
clang-27747	173,841	612	3,743	1,613	107.40	612	3,411	1,374	126.08	596	2,409	1,127	153.72	498	1,820	835	207.60	498	1,271	656	264.24	498	1,470	624	277.79
clang-31259	48,800	920	4,189	1,605	29.83	920	3,805	1,524	31.42	920	3,010	1,357	35.28	608	2,262	1,230	39.18	608	1,474	783	61.55	608	1,644	751	64.17
gcc-59903	57,582	1,879	10,477	4,040	13.79	1,879	10,010	3,732	14.93	2,146	5,898	T/O	3.85	1,901	5,803	2,349	23.70	1,901	3,618	1,429	38.97	1,901	4,380	1,539	36.18
gcc-60116	75,225	1,281	9,066	4,706	15.71	1,281	8,392	4,496	16.45	378	6,329	4,050	18.48	1,361	6,162	3,297	22.40	1,361	3,718	1,907	38.73	1,331	4,631	1,964	37.62
gcc-61383	32,450	1,287	6,161	1,454	21.43	1,287	4,717	1,166	26.73	1,393	4,279	1,960	15.85	1,031	2,955	960	32.73	1,031	2,103	799	39.32	1,031	2,031	473	66.42
gcc-61452	26,733	1,023	5,548	3,563	7.22	1,023	5,165	3,515	7.31	1,142	4,336	3,477	7.36	1,023	2,649	2,574	9.99	1,023	1,685	2,444	10.52	1,023	2,084	1,887	13.62
gcc-61917	85,360	1,401	8,132	1,905	44.07	1,401	7,510	1,718	48.87	1,375	6,529	3,156	26.61	1,421	5,081	1,288	65.17	1,421	3,243	951	88.26	1,421	3,332	693	121.12
gcc-64990	148,932	1,120	6,350	1,932	76.51	1,120	5,859	1,735	85.19	1,150	4,619	1,831	80.71	1,127	4,203	1,279	115.56	1,127	2,768	944	156.57	1,127	3,385	879	168.15
gcc-65383	43,943	1,121	5,850	1,628	26.30	1,121	5,395	1,447	29.59	1,152	4,375	1,328	32.22	1,110	3,767	1,013	42.28	1,110	2,381	748	57.26	1,110	2,926	729	58.76
gcc-66186	47,482	1,299	4,889	1,244	37.12	1,299	4,539	1,129	40.91	1,297	3,623	1,067	43.28	1,288	3,009	717	64.43	1,288	1,877	544	84.92	1,288	2,231	537	86.02
gcc-71626	6,134	61	474	38	159.82	61	446	34	178.62	61	239	41	148.12	61	288	32	189.78	61	229	28	216.89	61	185	16	379.56
gcc-71632	141	82	190	62	0.95	82	190	60	0.98	82	132	57	1.04	82	197	60	0.98	82	114	55	1.07	82	121	56	1.05
gcc-77624	1,306	23	99	13	98.69	23	97	5	256.60	23	134	14	91.64	23	136	11	116.64	23	108	10	128.30	23	83	6	213.83
geomean	29,769	654	3,264	1,013	27.29	654	3,007	876	31.56	605	2,409	1,156	23.94	587	2,114	764	36.24	587	1,416	595	46.58	586	1,490	476	58.21
median	47,482	1,120	5,114	1,628	29.83	1,120	4,674	1,447	31.42	944	4,279	1,831	26.67	1,023	2,955	1,230	42.28	1,023	1,977	799	57.26	1,023	2,084	729	66.42
rust-44800	802	472	2,113	11,907	0.03	472	2,102	11,208	0.03	472	1,824	9,433	0.03	472	1,823	9,859	0.03	472	1,107	6,576	0.05	472	1,291	6,267	0.05
rust-63791	8,144	5,768	3,873	T/O	0.17	5,809	4,126	T/O	0.16	5,553	3,400	T/O	0.18	5,496	3,574	T/O	0.18	4,948	3,737	T/O	0.22	5,496	3,463	T/O	0.18
rust-65934	107	100	148	86	0.08	100	148	76	0.09	100	126	64	0.11	100	126	66	0.11	100	72	38	0.18	100	77	41	0.17
rust-69039	191	128	428	2,275	0.03	128	428	2,265	0.03	128	523	2,559	0.02	128	402	1,932	0.03	128	252	1,217	0.05	128	290	1,417	0.04
rust-77002	348	302	818	1,846	0.02	302	810	2,195	0.02	302	626	2,045	0.02	302	626	2,105	0.02	302	457	1,536	0.03	302	617	2,018	0.02
rust-77993	4,989	48	174	181	27.30	48	169	160	30.88	50	187	190	25.99	50	196	188	26.27	50	166	175	28.22	48	164	166	29.77
rust-78336	980	18	117	504	1.91	18	109	487	1.98	18	89	500	1.92	18	93	459	2.10	18	79	313	3.07	18	67	317	3.03
rust-78622	157	29	78	335	0.38	29	74	282	0.45	29	51	269	0.48	29	51	240	0.53	29	39	159	0.81	29	39	160	0.80
geomean	659	151	401	1,083	0.22	151	396	1,037	0.23	151	348	1,019	0.22	150	343	971	0.24	148	254	701	0.35	150	268	739	0.31
median	575	114	301	1,175	0.13	114	299	1,341	0.13	114	355	1,273	0.15	114	299	1,196	0.15	114	209	765	0.20	114	227	867	0.18
go-28390	146	84	212	33	1.88	84	211	30	2.07	84	201	27	2.30	84	201	26	2.38	84	127	18	3.44	84	134	18	3.44
go-29220	127	74	98	12	4.42	74	98	8	6.63	74	123	13	4.08	74	123	11	4.82	74	69	6	8.83	74	68	6	8.83
go-30606	449	423	629	155	0.17	423	629	149	0.17	423	669	161	0.16	423	669	162	0.16	423	421	100	0.26	423	429	101	0.26
geomean	203	138	236	39	1.12	138	235	33	1.33	138	255	38	1.15	138	255	36	1.22	138	155	22	1.99	138	158	22	1.99
median	146	84	212	33	1.88	84	211	30	2.07	84	201	27	2.30	84	201	26	2.38	84	127	18	3.44	84	134	18	3.44
urls.xml	679	30	37	2	324.50	30	37	2	324.50	30	41	2	324.50	30	41	2	324.50	30	39	2	324.50	30	35	2	324.50

O, R and Q denote number of tokens in the original test case, reduced one and total number of oracle queries performed by the reduction technique, respectively. T is the reduction time in seconds and E is the efficiency in terms of the number of tokens removed per second. Timeout (T/O) is set to 4 hours.

Table 4.2: A comparison between different variants of Perses and PARDIS. Node replacements enabled.

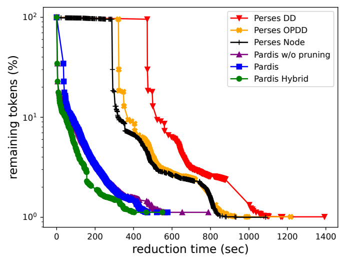
Test Case	$O_{(\#)}$	Perses DD				Perses OPDD				Perses N				Pardis w/o Pruning				Pardis				Pardis Hybrid			
		$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$
clang-22382	21,069	334	5,030	3,535	5.87	334	4,656	3,474	5.97	343	3,820	4,973	4.17	343	3,286	4,338	4.78	343	2,847	4,273	4.85	343	3,081	3,076	6.74
clang-22704	184,445	266	4,289	1,788	103.01	266	3,950	1,489	123.69	292	4,469	3,664	50.26	173	4,709	3,246	56.77	173	4,506	3,296	55.91	173	3,265	1,680	109.69
clang-23309	38,648	237	6,747	2,229	17.23	237	6,365	2,095	18.33	316	4,260	1,643	23.33	241	6,091	2,508	15.31	60	4,741	2,109	18.30	874	7,092	2,630	14.36
clang-25900	78,961	304	4,578	1,660	47.38	304	4,205	1,493	52.68	307	3,336	1,440	54.62	309	2,729	1,070	73.51	307	2,383	987	79.69	307	2,508	919	85.59
clang-27747	173,841	220	4,311	1,624	106.91	220	3,974	1,495	116.13	244	2,797	1,200	144.66	230	2,372	1,172	148.13	230	2,059	1,069	162.41	220	2,467	1,125	154.33
clang-31259	48,800	374	5,312	2,424	19.98	374	4,920	2,161	22.41	374	4,108	2,053	23.59	376	3,242	1,668	29.03	376	2,847	1,495	32.39	376	2,964	1,369	35.37
gcc-59903	57,582	465	9,908	4,572	12.49	465	9,680	4,489	12.72	465	6,851	3,595	15.89	449	6,746	9,675	5.91	392	5,479	8,947	6.39	392	5,789	8,955	6.39
gcc-60116	75,225	480	7,514	9,088	8.22	480	7,129	8,905	8.39	441	5,713	3,578	20.90	528	6,057	3,343	22.34	528	5,397	3,107	24.04	532	5,835	3,143	23.76
gcc-61383	32,450	321	5,649	1,805	17.80	321	4,487	1,472	21.83	267	3,869	2,255	14.27	292	2,961	1,373	23.42	288	2,532	1,284	25.05	288	2,191	916	35.11
gcc-61452	26,733	370	5,552	3,993	6.60	370	5,238	3,876	6.80	328	4,081	4,299	6.14	342	2,974	2,975	8.87	342	2,539	2,884	9.15	342	2,696	2,224	11.87
gcc-61917	85,360	327	5,869	1,829	46.49	327	5,399	1,652	51.47	358	5,002	3,431	24.77	325	4,006	1,304	65.21	329	3,580	1,265	67.22	329	3,350	988	86.06
gcc-64990	148,932	354	5,873	2,288	64.94	354	5,489	2,112	70.35	359	4,395	1,954	76.04	350	3,761	1,643	90.43	354	2,696	1,248	119.05	354	2,851	1,161	127.97
gcc-65383	43,943	228	5,383	1,819	24.03	228	5,002	1,723	25.37	229	4,192	1,872	23.35	273	2,824	1,230	35.50	279	2,130	1,151	37.94	279	2,217	1,029	42.43
gcc-66186	47,482	441	4,692	1,563	30.10	441	4,392	1,403	33.53	441	4,271	1,670	28.17	391	2,346	982	47.95	391	2,085	914	51.52	391	2,219	863	54.57
gcc-71626	6,134	60	535	36	168.72	60	507	35	173.54	60	300	40	151.85	60	355	35	173.54	60	299	32	189.81	60	255	14	433.86
gcc-71632	141	75	217	57	1.16	75	217	57	1.16	70	225	61	1.16	75	220	57	1.16	75	173	54	1.22	75	176	52	1.27
gcc-77624	1,306	22	196	10	128.40	22	194	13	98.77	22	231	18	71.33	22	233	16	80.25	22	205	15	85.60	22	180	11	116.73
geomean	29,769	232	3,315	1,119	25.27	232	3,093	1,058	26.72	234	2,589	1,158	24.50	227	2,288	987	28.63	207	1,933	912	31.01	242	1,967	758	37.26
median	47,482	321	5,312	1,819	24.03	321	4,656	1,652	25.37	316	4,108	1,954	23.59	309	2,974	1,373	35.50	307	2,539	1,265	37.94	329	2,696	1,125	42.43
rust-44800	802	464	3,057	T/O	0.02	464	2,728	T/O	0.02	464	2,698	T/O	0.02	464	2,451	T/O	0.02	464	2,437	T/O	0.02	464	2,307	T/O	0.02
rust-63791	8,144	6,203	3,395	T/O	0.13	6,269	3,285	T/O	0.13	6,209	3,341	T/O	0.13	6,188	3,533	T/O	0.14	7,923	1,339	T/O	0.02	6,332	2,996	T/O	0.13
rust-65934	107	100	238	510	0.01	100	238	559	0.01	100	216	518	0.01	100	216	503	0.01	100	166	368	0.02	100	171	424	0.02
rust-69039	191	120	826	4,372	0.02	120	826	4,248	0.02	120	1,027	5,125	0.01	120	799	4,079	0.02	120	689	3,528	0.02	120	727	3,695	0.02
rust-77002	348	286	3,900	11,018	0.01	286	3,892	9,650	0.01	286	3,712	9,106	0.01	286	3,712	9,110	0.01	286	3,646	9,657	0.01	286	3,819	9,684	0.01
rust-77993	4,989	16	175	705	7.05	16	170	786	6.33	16	175	751	6.62	16	181	674	7.38	16	169	1,079	4.61	16	165	781	6.37
rust-78336	980	15	106	438	2.20	15	98	379	2.55	15	77	303	3.18	15	81	340	2.84	15	73	352	2.74	15	61	299	3.23
rust-78622	157	29	77	319	0.40	29	73	310	0.41	29	52	229	0.56	29	52	282	0.45	29	40	178	0.72	29	40	172	0.74
geomean	659	127	571	2,176	0.12	127	550	2,139	0.12	127	518	2,005	0.12	127	505	1,995	0.12	131	401	1,908	0.10	128	437	1,831	0.14
median	575	110	532	2,539	0.08	110	532	2,517	0.08	110	622	2,938	0.08	110	508	2377	0.08	110	429	2,304	0.02	110	449	2,238	0.08
go-28390	146	84	245	35	1.77	84	244	35	1.77	84	234	32	1.94	84	234	32	1.94	84	163	24	2.58	84	170	23	2.70
go-29220	127	60	165	16	4.19	60	165	18	3.72	65	188	19	3.26	60	180	18	3.72	60	158	16	4.19	62	156	17	3.82
go-30606	449	233	912	204	1.06	233	912	203	1.06	233	930	205	1.05	233	930	204	1.06	213	794	179	1.32	213	802	187	1.26
geomean	203	106	333	49	1.99	106	332	50	1.91	108	345	50	1.88	106	340	49	1.97	102	273	41	2.43	104	277	42	2.35
median	146	84	245	35	1.77	84	244	35	1.77	84	234	32	1.94	84	234	32	1.94	84	163	24	2.58	84	170	23	2.70
urls.xml	679	9	21	1	670	9	21	1	670	9	21	1	670	9	21	1	670	9	19	1	670	9	19	1	670

O, *R* and *Q* denote number of tokens in the original test case, reduced one and total number of oracle queries performed by the reduction technique, respectively. *T* is the reduction time in seconds and *E* is the efficiency in terms of the number of tokens removed per second. Timeout (T/O) is set to 4 hours.

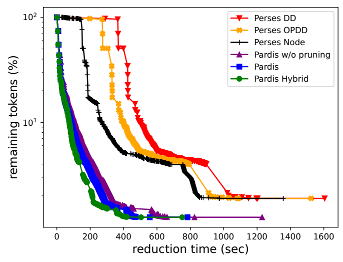
Table 4.3: The speed up achieved by each test case reduction technique with respect to Perses DD.

Test Case	Replacements disabled					Replacements enabled				
	Perses OPDD	Perses N	PARDIS W/O PRUNING	PARDIS	PARDIS HYBRID	Perses OPDD	Perses N	PARDIS W/O PRUNING	PARDIS	PARDIS HYBRID
clang-22382	1.02	0.74	0.81	0.84	1.25	1.02	0.71	0.81	0.83	1.15
clang-22704	1.16	0.24	0.38	0.39	1.20	1.20	0.49	0.55	0.54	1.06
clang-23309	1.08	1.30	1.88	2.89	2.76	1.06	1.36	0.89	1.06	0.85
clang-25900	1.14	1.28	1.77	2.41	2.53	1.11	1.15	1.55	1.68	1.81
clang-27747	1.17	1.43	1.93	2.46	2.58	1.09	1.35	1.39	1.52	1.44
clang-31259	1.05	1.18	1.30	2.05	2.14	1.12	1.18	1.45	1.62	1.77
gcc-59903	1.08	0.28	1.72	2.83	2.63	1.02	1.27	0.47	0.51	0.51
gcc-60116	1.05	1.16	1.43	2.47	2.40	1.02	2.54	2.72	2.93	2.89
gcc-61383	1.25	0.74	1.51	1.82	3.07	1.23	0.80	1.31	1.41	1.97
gcc-61452	1.01	1.02	1.38	1.46	1.89	1.03	0.93	1.34	1.38	1.80
gcc-61917	1.11	0.60	1.48	2.00	2.75	1.11	0.53	1.40	1.45	1.85
gcc-64900	1.11	1.06	1.51	2.05	2.20	1.08	1.17	1.39	1.83	1.97
gcc-65383	1.13	1.23	1.61	2.18	2.23	1.06	0.97	1.48	1.58	1.77
gcc-66186	1.10	1.17	1.74	2.29	2.32	1.11	0.94	1.59	1.71	1.81
gcc-71626	1.12	0.93	1.19	1.36	2.38	1.03	0.90	1.03	1.13	2.57
gcc-71632	1.03	1.09	1.03	1.13	1.11	1.00	0.93	1.00	1.06	1.10
gcc-77624	2.60	0.93	1.18	1.30	2.17	0.77	0.56	0.63	0.67	0.91
geomean	1.16	0.88	1.32	1.71	2.13	1.06	0.97	1.13	1.23	1.48
median	1.11	1.06	1.48	2.05	2.32	1.06	0.94	1.34	1.41	1.77
rust-44800	1.06	1.26	1.21	1.81	1.90	1.00	1.00	1.00	1.00	1.00
rust-63791	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
rust-65934	1.13	1.34	1.30	2.26	2.10	0.91	0.98	1.01	1.39	1.20
rust-69039	1.00	0.89	1.18	1.87	1.61	1.03	0.85	1.07	1.24	1.18
rust-77002	0.84	0.90	0.88	1.20	0.91	1.14	1.21	1.21	1.14	1.14
rust-77993	1.13	0.95	0.96	1.03	1.09	0.90	0.94	1.05	0.65	0.90
rust-78336	1.03	1.01	1.10	1.61	1.59	1.16	1.45	1.29	1.24	1.46
rust-78622	1.19	1.25	1.40	2.11	2.09	1.03	1.39	1.13	1.79	1.85
geomean	1.04	1.06	1.12	1.54	1.46	1.02	1.08	1.09	1.14	1.19
median	1.05	1.01	1.14	1.71	1.60	1.02	1.00	1.06	1.19	1.16
go-28390	1.10	1.22	1.27	1.83	1.83	1.00	1.09	1.09	1.46	1.52
go-29220	1.50	0.92	1.09	2.00	2.00	0.89	0.84	0.89	1.00	0.94
go-30606	1.04	0.96	0.96	1.55	1.53	1.00	1.00	1.00	1.14	1.09
geomean	1.20	1.03	1.10	1.78	1.78	0.96	0.97	0.99	1.19	1.16
median	1.10	0.96	1.09	1.83	1.83	1.00	1.00	1.00	1.14	1.09
urls.xml	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

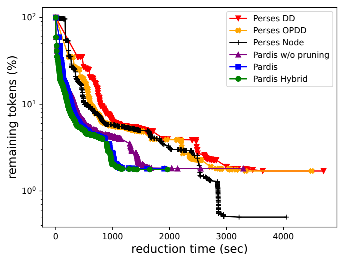
To further expand our evaluation, we compute a metric that measures the speed up gained by each variant with respect to Perses DD by dividing the reduction time of Perses DD by the reduction time of the variant. Table 4.3 depicts the results. Columns 2 to 6 represent the speed up metric computed by the reduction times available in Table 4.1 while columns 7 to 11 show speed up calculated by using data in Table 4.2. As can be seen, either PARDIS or PARDIS HYBRID results in the highest speed up in the majority of the cases. In fact, there are only four test cases where PARDIS or PARDIS HYBRID did not have the highest speed up. For two test cases, `rust-63791` and `urls.xml`, the speed up achieved is equal to 1 for all variants, meaning that no speed up is indeed achieved. The reason is that all reduction variants including Perses DD time out when reducing the first test case and they all reduce the second test case quite fast. As a result, the reduction time for all variants is the same in these two cases. For the other two test cases, `gcc-77624` and `rust-77993`, Perses OPDD has the highest speed up. By having a closer look at Table 4.1, we can see that these test cases also have a relatively fast reduction for all variants and the difference



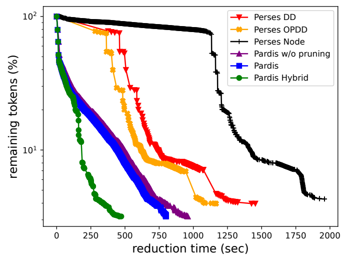
(a) clang-25900



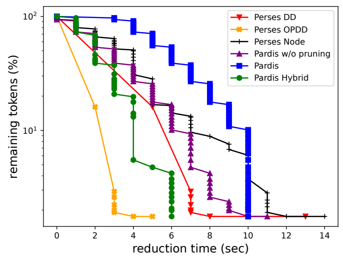
(b) clang-31259



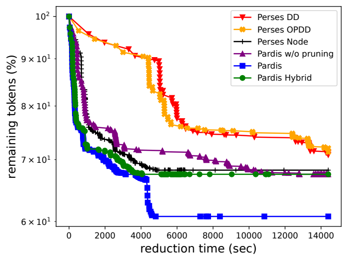
(c) gcc-60116



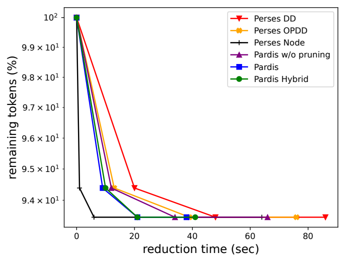
(d) gcc-61383



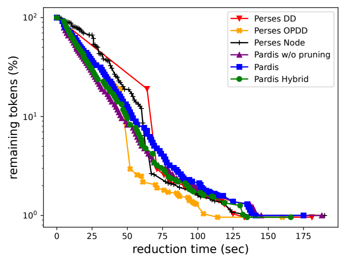
(e) gcc-77624



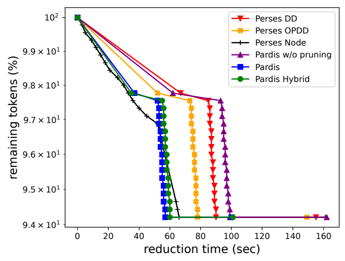
(f) rust-63791



(g) rust-65934



(h) rust-77993



(i) go-30606

Figure 4.5: Converging to a reduced test case in our sample benchmark using six variants of reduction techniques. Node replacements disabled.

in reduction time of variants is just a few seconds which means that Pardis and Pardis Hybrid are fast in reducing these test cases as well.

In addition, we graphed the reduction progress of each test case in our sample benchmark when using the six variants of reduction techniques. Figure 4.5 depicts the percentage of remaining tokens over time during reduction. Note that the y-axis is log scaled. In general, Pardis and Pardis Hybrid show much faster convergence towards a reduced test case compared to Perses variants in the majority of the cases.

Finally, recall that the only factor differentiating Perses N from Pardis w/o Pruning is the order in which the queue of nodes is traversed. Unlike Perses N, Pardis w/o Pruning does not suffer from priority inversion and guides the reduction process based on token weights of the nodes. As can be seen, this advantage leads to faster convergence to a reduced test case, especially in the majority of the large test cases. We examine the impact of priority inversion on reduction speed more rigorously in the next section.

4.5.2 RQ2. The Impact of Priority Inversion

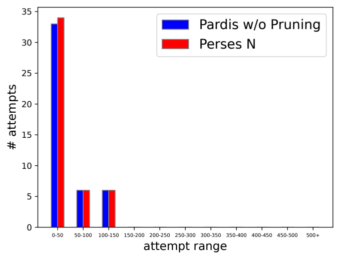
As presented in our motivating example in Section 4.1 and as shown empirically in Figure 4.5, faster convergence can be achieved by avoiding priority inversion using Pardis or Pardis Hybrid. This section further investigates the potential reason for this faster convergence.

One explanation to consider is that the priority awareness model in Pardis and Pardis Hybrid may decrease the amount of work required to remove a token. To explore this on our sample benchmark, we define a metric called *the number of removal attempts* for a token that is the number of times a single token is considered for removal. Based on the structure of the parse trees, removing any ancestor of a token in the tree will remove that token, so if a first attempt fails, a deeper ancestor may be attempted. We compute this for every token of the test case to get a sense of the work required for each token. A better traversal order of the parse tree should cause *fewer* overall token removal attempts.

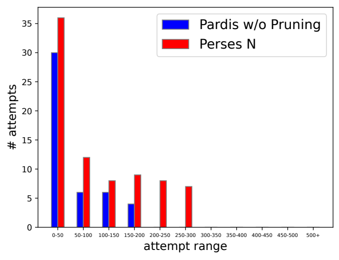
To measure only the impact of different traversal orders, we compare Pardis w/o Pruning with Perses N. As described in Section 4.5.1, they follow the exact same reduction rules and differ only in their traversal orders.

Figure 4.6 depicts histograms of the distributions of token removal attempts for Pardis w/o Pruning and Perses N on our sample benchmark. In particular, for large test cases such as `clang-25900`, `clang-31259`, `gcc-60116`, `gcc-61383`, and `rust-63791`, we observe either a higher overall number of token removal attempts for Perses N or a tendency in its distribution towards larger values of removal attempts. This indicates that Perses N requires more work to remove individual tokens in these cases. For smaller test cases, the number of attempts is almost the same for Perses N and Pardis w/o Pruning. The data presented in Table 4.1 also show that the performance of Perses N and Pardis w/o Pruning is almost the same when reducing these small test cases.

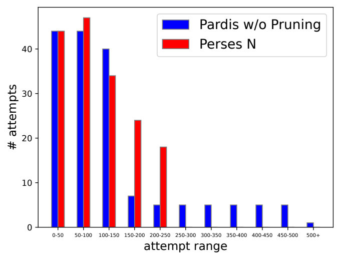
Additionally, for each test case in our sample benchmark with different number of removal attempts in Perses N and Pardis w/o Pruning, we statically measure that the difference between the two reduction variants is significant. We use a Wilcoxon signed-rank test [49] to determine whether the distribution of Perses N is indeed greater than that of Pardis w/o Pruning. The smallest p-values we collect are for test cases `clang-31259`, `gcc-60116`, and `gcc-61383` which indicate that Perses N has a substantially greater distribution of token removal attempts compared to Pardis w/o Pruning for these test cases.



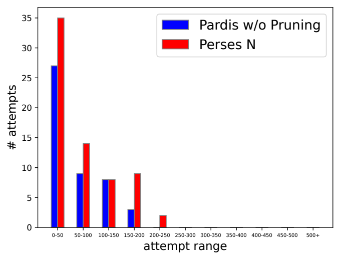
(a) clang-25900



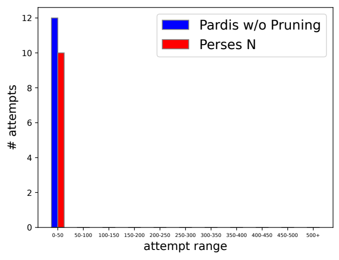
(b) clang-31259



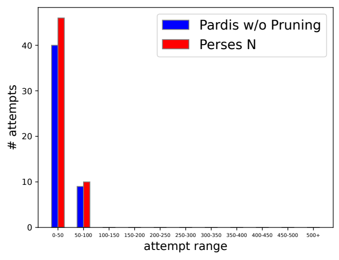
(c) gcc-60116



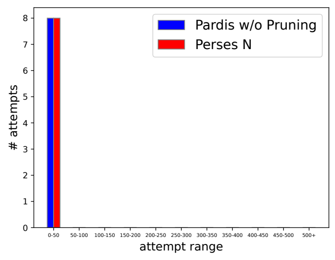
(d) gcc-61383



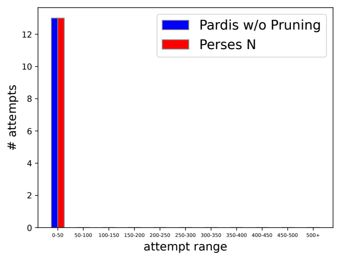
(e) gcc-77624



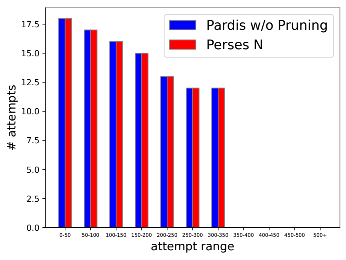
(f) rust-63791



(g) rust-65934



(h) rust-77993



(i) go-30606

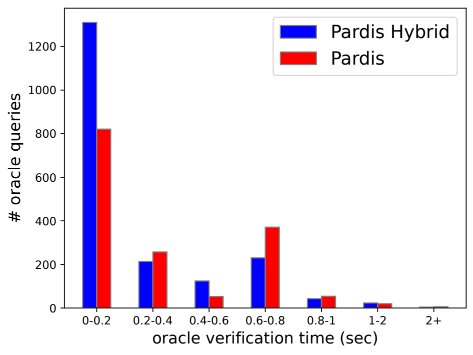
Figure 4.6: Distribution of token removal attempts for Pardis w/o Pruning and node at a time Perses (Perses N).

4.5.3 Pardis Hybrid: A Potential *Sweet Spot* in Reduction

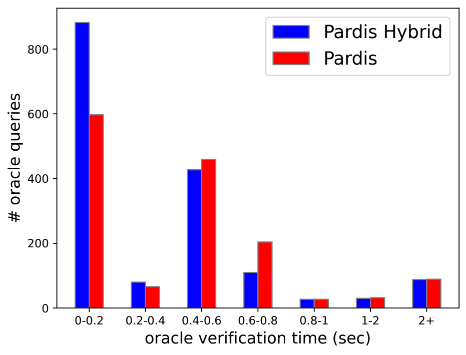
As discussed earlier, unlike Perses, Pardis Hybrid does not suffer from priority inversion because it prioritizes the search primarily on the token weight of nodes being considered for removal. Moreover, unlike Pardis, it does not strictly remove one node at a time and allows the removal of nodes with the same weight and the same parent as a group. Hence, it can be considered a sweet spot in reducing test cases. We conduct two studies that can further explore this idea.

1) Oracle Verification Time

The number of oracle queries is a common metric used in similar studies to reason about reduction efficiency since it directly impacts the total reduction time [15, 1, 14, 31, 2]. For



(a) clang-25900



(b) clang-31259

Figure 4.7: Distribution of oracle verification time for Pardis and Pardis Hybrid.

instance, as shown in Table 4.1, both Pardis and Pardis Hybrid perform fewer oracle queries and take less time than Perses.

However, the number of oracle queries is not the only factor involved. The time required to run each of these queries, or *oracle verification time*, also affects the total running time. For instance, in one Rust and 9 C test cases in Table 4.1, Pardis performs fewer oracle queries while Pardis Hybrid has a faster reduction time with larger number of oracle queries.

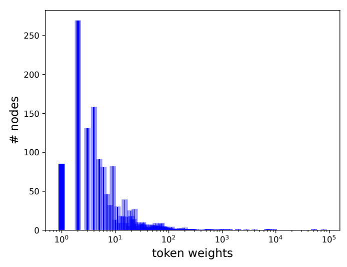
Oracle verification time can depend on multiple factors such as the size and complexity of the test case. Since Pardis Hybrid takes advantage of the possibility to remove more than one node at a time, it may try variants of the test case that are smaller and may be faster to verify compared to Pardis. These potentially less expensive tests can lead to faster reduction time of Pardis Hybrid when reducing some test cases in spite of its larger number of oracle queries.

To check this hypothesis, we select the two test cases, `clang-25900` and `clang-31259`, from our sample benchmark that satisfy the following criteria in their reduction results:

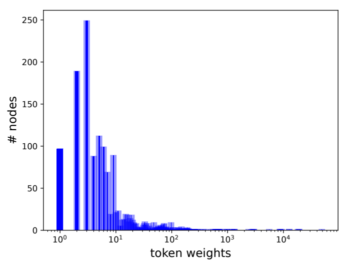
- The number of oracle queries performed by Pardis is smaller than Pardis Hybrid.
- Pardis Hybrid performs reduction in shorter time compared to Pardis.

Next, we record the running time of each oracle query during the reduction of these test cases.

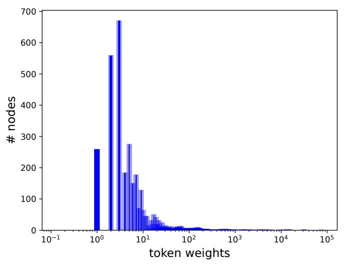
Figure 4.7 depicts the distribution of oracle verification times in Pardis and Pardis Hybrid for `clang-25900` and `clang-31259`. The distributions show that Pardis has more queries that take longer compared to Pardis Hybrid. In particular, Pardis has a considerably larger number of oracle queries with verification time in the range of 0.6 to 0.8 seconds. In contrast, the majority of oracle queries run by Pardis Hybrid take between 0 and 0.2 seconds which are the fastest tests. As a result, the shorter queries in Pardis Hybrid can



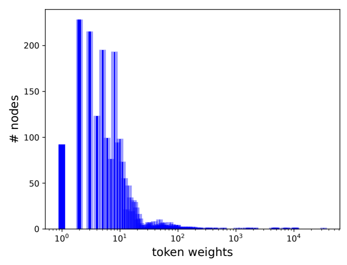
(a) clang-25900



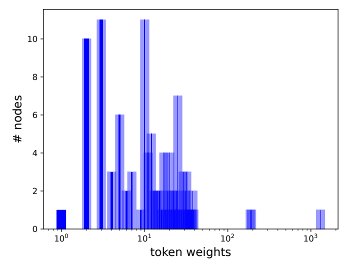
(b) clang-31259



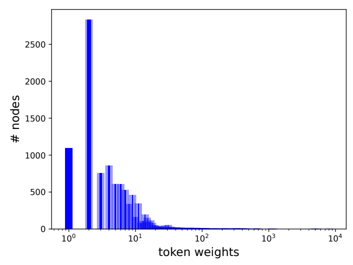
(c) gcc-60116



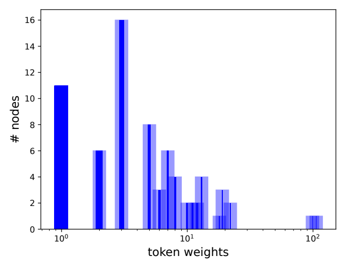
(d) gcc-61383



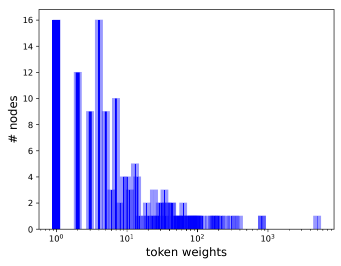
(e) gcc-77624



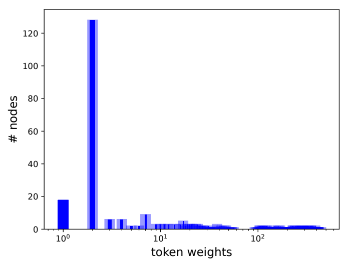
(f) rust-63791



(g) rust-65934



(h) rust-77993



(i) go-30606

Figure 4.8: Distribution of token weights of nodes visited during PARDIS reduction.

decrease its overall reduction time, making it reduce test cases with fewer queries compared to Perses and shorter queries compared to PARDIS. Thus, we can refer to it as a *sweet spot* in test case reduction.

2) Distribution of Token Weights

The motivation behind proposing PARDIS HYBRID as discussed in Section 4.1 was that if lists in a test case shrink after removing nodes with large unique token weights, applying Delta Debugging on list elements with the same weight can be beneficial. In fact, the more of the remaining nodes that share token weights, the more beneficial using Delta Debugging becomes since it provides the opportunity to remove those nodes in just one test. This can avoid the possibly time-consuming process of visiting nodes one by one.

To understand the distribution of token weights in practice, we run Pardis original, the node by node removal variant on test cases of our sample benchmark. Figure 4.8 shows the x-axis log scaled distributions of token weights for these test cases. As can be seen, all distributions are inclined towards smaller values of token weights. This indicates that there is a large number of nodes with small token weights visited during reduction by Pardis and combining them for removal may be advantageous.

Moreover, we compute the median of token weights for each test case to statistically measure the middle value of our data sets. For the large C test cases, `clang-25900`, `clang-31259`, `gcc-60116`, and `gcc-61383`, the median value is smaller than 8. This means that half of the nodes have one of only eight different small token weights and can benefit from the grouped removals. Similarly, the median values for `gcc-77624` and Rust test cases are 12 and less than 10, respectively. The median value of the only Go test case in our sample benchmark, `go-30606` is 24, which is also relatively small compared to the initial size of the test case that is 449 tokens.

These small median values can further motivate the use of Pardis Hybrid in practice.

4.6 Summary

In this chapter, we identified and explored one of the drawbacks of Perses [15], the latest state of the art syntax guided test case reducer. We referred to this drawback as *priority inversion* in which a low priority task is scheduled instead of a high priority task. We demonstrated empirically that priority inversion can adversely impact the overall reduction time. To mitigate this problem, we suggested Pardis, a priority aware queue driven test case reduction technique. Similar to Perses, Pardis preserves syntactic validity during reduction. However, it does not suffer from the priority inversion present in Perses. To further optimize our proposed technique, we introduced another variant of our reducer called Pardis Hybrid. This new variant adheres to the priority models provided by Pardis. In addition, it brings in the capability of removing several elements together. Using either Pardis or Pardis Hybrid when reducing our benchmark led to an average speed up of 2.13x, 1.54x, and 1.78x for C, Rust, and Go domains, respectively while preserving a similar reduction power compared to the state of the art Perses. More specifically, we generated outputs of even smaller size (by 10% on average) for C. For Rust, using Pardis and Pardis Hybrid generated outputs with 148 and 150 tokens on average, respectively while Perses produced outputs with 151 tokens on average. The average reduced size for the Go domain was 138 tokens for all three techniques.

Chapter 5

Machine Learning and Test Case Reduction

This chapter integrates machine learning algorithms into test case reduction to accelerate its performance. We begin by addressing the third problem in our list of problems in Section 2.5 and continue by providing solutions for problems 2.2 and 4.

We start by exploring the following question:

How can we mitigate the problem of semantically invalid candidate variants generated during test case reduction?

Recall that both Perses [15] and PARDIS [28], our tool presented in Chapter 4, preserve syntactic validity by constructing their priority work queues based on *syntactically* removable nodes. However, an oracle function typically checks multiple aspects of a candidate test case other than syntactic validity. These aspects along with the verification of the bug can make running such oracles an expensive task, especially for complex test cases and inputs with a long running time. Hence, it is desirable to avoid running oracles on test case variants that are not likely to preserve the property of interest.

One group of these variants are *semantically* invalid programs that can be divided into two groups:

- Programs in which the dependencies among their elements are broken. These programs cause compile time issues. For example, removing the declaration of a variable *before* its use generates a program that does not compile.
- Programs with undefined behavior that could cause run time issues.

Precise identification of semantic invalidity requires significant effort and expertise to develop domain specific tools. These tools will be aware of dependencies among elements of a program and can check for undefined behavior. However, as previously stated in Section 2.4.2, they are not reusable across domains which may hinder their availability. Hence,

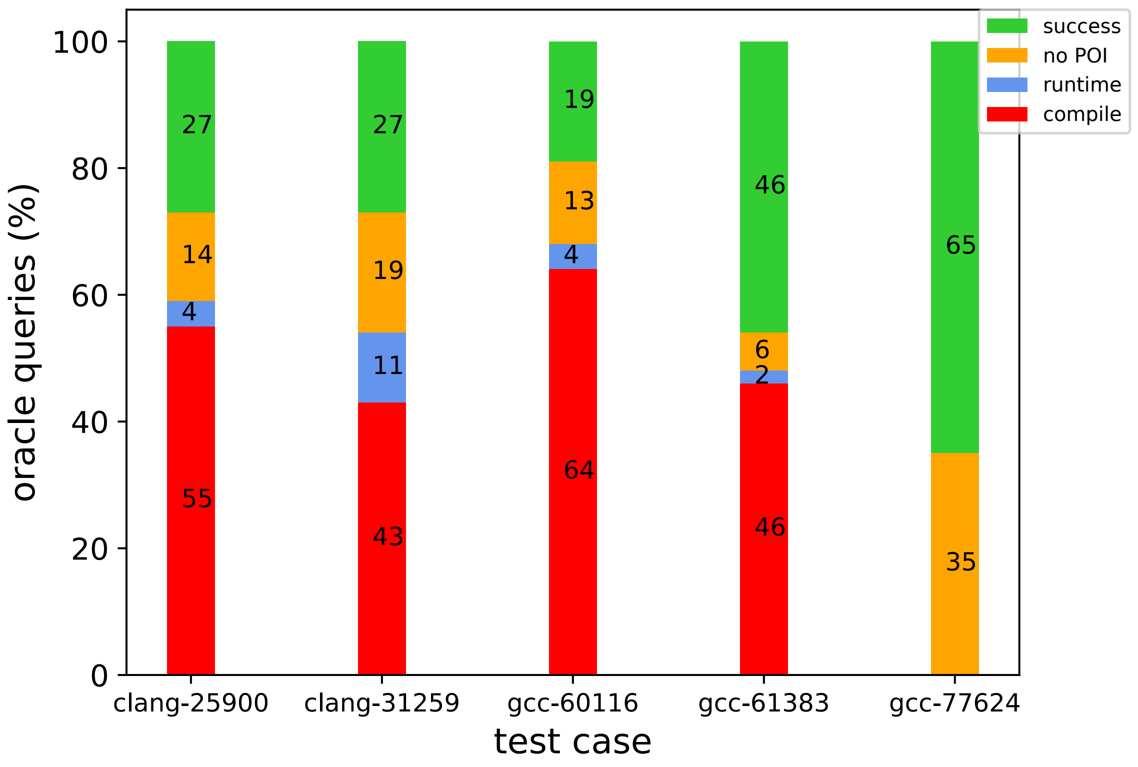


Figure 5.1: A breakdown of the outcomes of the oracle queries performed by PARDIS.

general purpose and domain agnostic techniques are desirable, and improving them is the focus of this dissertation, including this chapter.

To demonstrate that the problem of semantic invalidity is real, we provide a breakdown of oracle outcomes produced during reduction by PARDIS for the C test cases in our sample benchmark used throughout the dissertation. We select the C test cases because they are larger which can make them potentially more complex to reduce.

This breakdown is presented in Figure 5.1 with the following four categories of test outcomes:

1. Invalid due to compile time issues: These tests are performed on test case variants that cannot compile. Because PARDIS preserves syntactic validity during reduction, these tests are the direct cause of violating semantic validity rules such as breaking dependencies as described above.

2. Invalid due to run time issues: These tests are performed on test case variants with undefined behavior such as programs with memory issues.

3. Valid but without the property of interest (no POI): These tests are performed on test case variants that are both syntactically and semantically valid but they do not contain the property of interest. For example, they do not trigger the failure.

4. Valid tests with the property of interest: These tests are performed on test case variants with the property of interest and can successfully reduce the test case. An ideal search space for a reducer should consist of only these variants.

Excluding `gcc-77624` that simply contains a sequence of declarations with no compile time issues generated in its reduction, we can see that oracle queries with compile time issues caused by semantic invalidity comprise a large portion of the performed tests with 55%, 43%, 64%, and 46% of the total number of tests in `clang-25900`, `clang-31259`, `gcc-60116`, and `gcc-61383`, respectively. This further motivates us to develop solutions that mitigate the problem of semantic invalidity present in the state of the art reducers.

To this end, in Section 5.1 of this chapter, we propose a model-guided variant of Pardis. Model Guided Pardis [29] tries to *avoid* performing tests on compile time semantically invalid test case variants by consulting trained models before querying the oracle. In Section 5.2, we extend our learning mechanism by proposing Type Batched Probabilistic Joint Reduction to *guide* reduction towards generating test case variants that are more likely to 1) be semantically valid, and 2) contain the bug.

5.1 Avoiding Semantic Invalidity

We originally introduced and published the ideas presented in this section at the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR 2021) [29].

Recall our motivating example in Chapter 4 with a program listing and its parse tree. To keep this section self contained, we reiterate the figures on the next page.

As can be seen, there are multiple dependencies among nodes of this parse tree. For instance, node ⑫ containing the use of `s1` depends on node ⑧ which itself depends on node ③ which declares struct `S`. Similarly in another dependency, node ⑫ is dependent on node ② with respect to variable `d`. There are other dependencies such as the condition of the `if statement` with dependency on node ⑩ or the dependency of call site `foo()` of node ⑭ on the definition of `foo` beneath node ④. Although all nodes involved in each dependency are syntactically removable based on the C grammar rules, they need to get removed in an appropriate order to avoid semantic validity issues.

In this chapter, we prune the search space of Pardis by leveraging generalized models to *predict* semantic validity of a node's removal. Specifically, we introduce a new reduction technique called Model Guided Pardis [29] that leverage models to predict whether removal of a node from the parse tree of a test case is semantically valid or not. If the model predicts that removing the node is semantically valid, then the oracle will be called

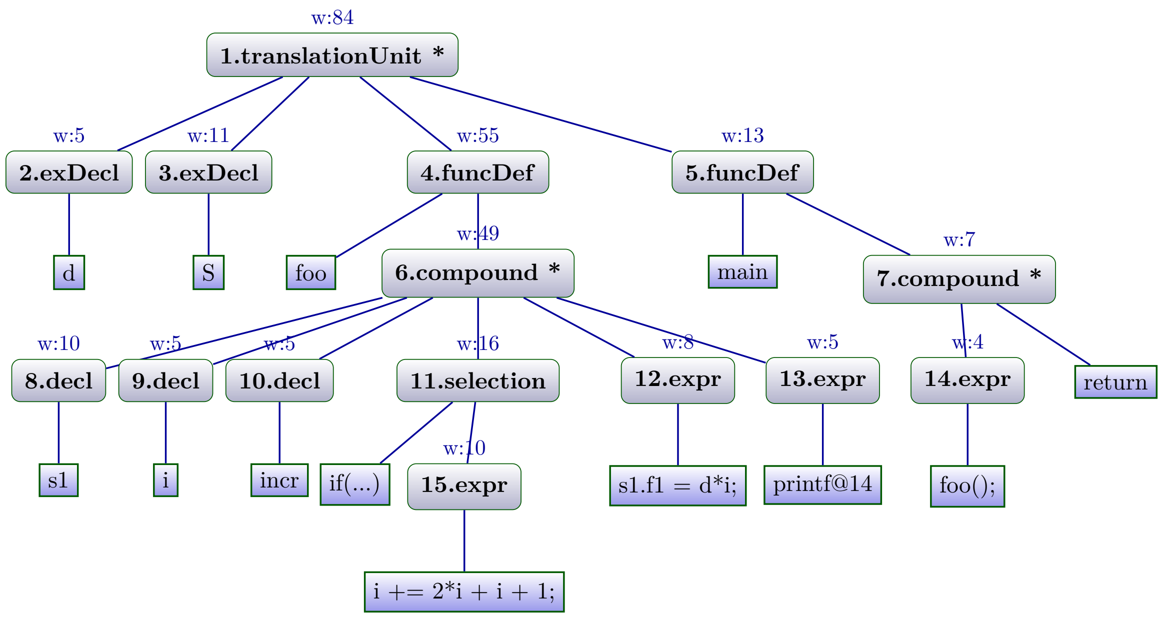


Figure 5.2: The simplified parse tree of program in Listing 5.1. Each internal node is annotated with an ID and its grammar rule type. * denotes a quantified Kleene-Star node.

to determine the test outcome. Otherwise, the oracle will be skipped and no test will be performed on that node. Our models are trained by extracting syntactic properties of a large corpus of programs described in Section 5.1.1. Given a correct grammar of any input domain, leveraging predictions by machine learning models allows for constructing a search space for Model Guided Pardis that consists of only candidate test cases that are syntactically valid and are likely to be semantically valid too.

```
1  int d = 10;
2  struct S {
3      int f1;
4      int f2;
5  };
6  void foo() {
7     struct S s1 = {1, 2};
8     int i = 0;
9     bool increment = true;
10    if (increment) {
11        i += 2*i + i + 1;
12    }
13    s1.f1 = d*i;
14    printf("Hello World!\n");
15 }
16 int main() {
17     foo();
18     return 0;
19 }
```

Listing 5.1: A C program with a statement to preserve on line 14.

Table 5.1: List of disallowed warnings used in oracle scripts of Perses to impose validity constraints during reduction.

Warning message	Warning message
incompatible redeclaration	more conversions than data arguments (printf)
eliding middle term	ordered comparison between pointer and integer
invalid in C99	format specifies type different from argument's type
return type of `main` is not `int`	incompatible integer to pointer conversion and vice versa
type specifier missing	assignment making integer from pointer without a cast
division by zero	missing return type: return type defaults to a type
cast from pointer to integer	expected semicolon at end of expression
missing declaration type	useless type name in empty declaration
too few arguments for format	comparison between pointer and integer
format expects type T (printf)	declaration does not declare anything
unknown type name	non-void function should return a value
use of undeclared identifier	excess elements in struct initializer

To this end, we need to train models that can correctly skip semantically invalid tests to improve the performance of the reduction. However, they should also have a high *precision* and be able to consider as many tests that are semantically valid as possible without skipping them in order to achieve reduced test cases with size similar to state of the art techniques.

We define a test case variant generated by removing a node as semantically invalid if it causes at least one of the following:

1. Compilation errors: Any test case variant that cannot get compiled successfully and causes compiler to return failure exit status is considered as semantically invalid.

2. Disallowed warnings: Any test case variant that generates a warning message that is in the list of warnings extracted from the oracle scripts of Perses [43] is semantically invalid. These warnings are presented in Table 5.1. The list includes a range of warnings from *missing type specifier* and *empty declaration* to *incompatible conversions*.

The focus of this study is to train models with respect to semantic invalidity caused by *compile time* issues since they comprise a larger portion of tests compared to tests with *run time* issues as briefly seen in Figure 5.1. In the following, we describe the process of collecting data and extracting features for training these predictive models.

5.1.1 Training Data Collection

To train models capable of predicting semantic validity, we need to collect training data in form of (source, target) pairs such that source contains the features representing the node and target is a Boolean value that shows a valid or invalid removal. To this end, we collect three different sets of training data for the three domains present in our benchmark: C, Rust and Go. We exclude the XML file because we have shown in the earlier chapters that the existing and proposed reducers can reduce this file in less than just two seconds which does not leave room for improvement. Moreover, the purpose of including this XML file in our benchmark was to demonstrate that our techniques are capable of reducing test cases beyond just programs. As a result, we can see this test case more as an indicator of feasibility rather than trying to improve it and will exclude it from the evaluations in the rest of the dissertation.

Our initial collected training sets consist of the following programs:

1. **C:** 40 large non-buggy fuzzer-generated C programs. These programs with an average number of more than 28,000 tokens are generated by Csmith [4], a tool for randomly generating C programs that statically and dynamically conform to the C99 standard [44].

2. **Rust:** 40 non-buggy Rust programs collected from Github trending Rust projects [50].

3. **Go:** 40 non-buggy Go programs collected from Github trending Go projects [51].

For each program, we build its parse tree and pass it to Pardis, its original version with *one node at at time removal*. Since Pardis places only syntactically removable nodes in a priority queue and removes them one at a time from the parse tree, it enables us to capture the removal success or failure of nodes with respect to their semantic validity by eliminating the possibility of removing syntactically invalid nodes. We define a validity checker oracle function and pass it to Pardis as a script along with the parse tree of the test case to reduce.

This oracle returns $True$ when both of the following criteria are satisfied:

- Removing the node from the parse tree does not yield semantic invalidity as determined by the compilation errors and warnings described above.

- Removing the node from the parse tree does not remove a specific randomly selected token from the test case.

The second criterion checks for a *simulated* property of interest which enables us to train our models without real-world bugs so that they can immediately be used on the next bug found. This is akin to a lightweight transfer learning process [52]. Moreover, it prevents early termination of a reduction phase and enables us to collect more data.

Figure 5.3 depicts the process of collecting and logging parse trees from test cases. By trying to remove each node, we capture a parse tree in which the node considered for removal, referred to as the *query node* henceforth, is tagged as the source along with a success (s) or failure (f) target value that is the outcome of the validity checker oracle on the candidate test case when the query node is removed.

If the query node *cannot be removed*, we record the parse tree, the node and the failure outcome and continue with trying to remove the next node in the queue.

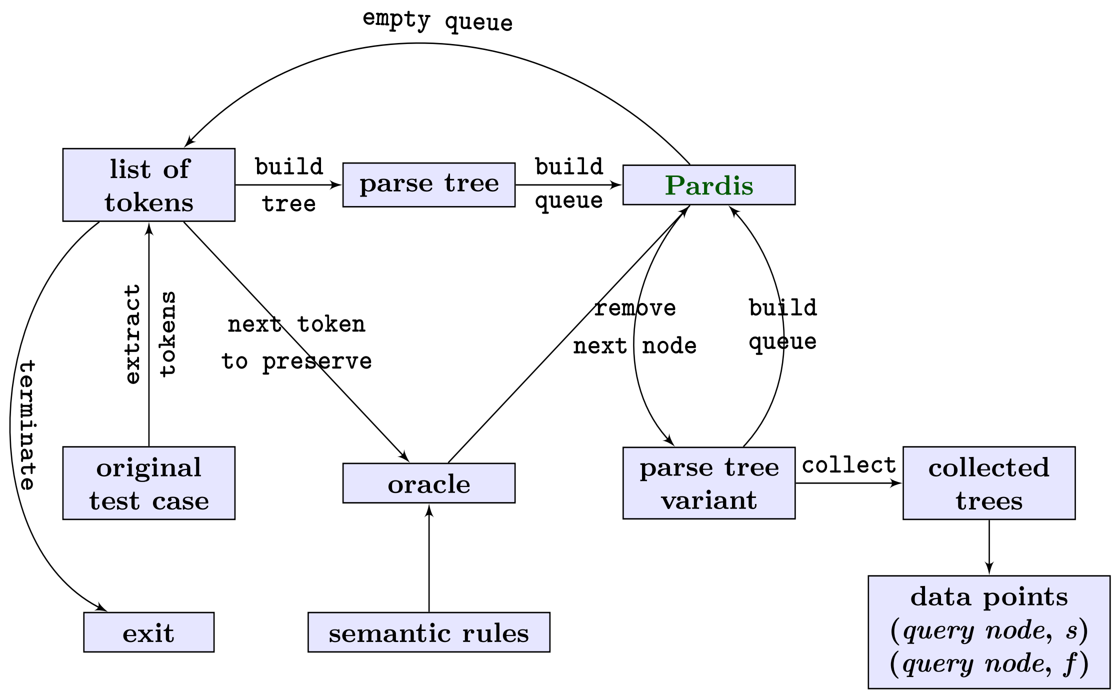


Figure 5.3: The training data collection process. For each test case, the process terminates when either every single token of the test case is considered to be preserved in the oracle or PARDIS hits a 30 minute timeout.

If the query node *can be removed* successfully, we record the parse tree, the node and the success outcome. Then we remove the node from the parse tree and continue with trying to remove the next node in the queue from the updated tree.

Hence, for each test, we record a parse tree that can be considered as one single data point in form of (source, target) where source is the query node with its features to be extracted and target is the success or failure of our oracle that is the validity checker script.

In total, we collect 286,842 parse trees for the C domain from which we use 215,131 trees (i.e., 75% of the total number of parse trees) to train our models and leave aside 71,711 trees as a test set. Our data set for C is close to balanced with 58% success vs. 42% failure labels. For the Rust domain, we initially collect 472,209 data points with 21% success and 79% failure labels. To balance this data set, we perform random undersampling to decrease the number of data points with failure labels. Our final balanced data set for Rust has 185,672 data points with 50% success and 50% failure labels from which we use 139,254 points to train our models. The initial data set collected for Go contains 174,371 points in which the failure labels comprise the majority of the collected labels. Similar to Rust, we perform random undersampling on the initial Go data set. From 28,940 data points in our final balanced Go set, we use 21,705 data points to train the Go models.

5.1.2 Features Extraction

Similar to a prior work on program mutation [53], we leverage simple syntactic properties of grammars to define our feature sets. In particular, we define the following sets of features for query nodes of parse trees collected in Section 5.1.1:

1. The grammar rule type of the query node itself.

2. The grammar rule types of the immediate children of the query node.

3. The grammar rule types of the nodes on the path from the query node to the root of the parse tree.

As an example, recall the parse tree in Figure 5.2. Suppose that we want to remove node 11.selection from the tree. This node will be our query node with the following features:

1. node's type: selection statement.

2. Children's types: terminal if and the type of node ⑮, expression statement.

3. Path to root's types: selection statement, compound statement, function definition, translation unit.

5.1.3 Features Representation

We make use of data constants and structures to represent the extracted features.

For the first feature, the node's grammar rule type, we use unique IDs to represent types of query nodes. Each unique ID corresponds to a unique rule type in the grammar. The C, Rust and Go grammars available on Perses Github [43] and used in our studies have 266, 709, and 213 unique rule types, respectively. Hence, we define 266, 709, and 213 unique IDs corresponding to these rule types.

To represent the second and third features, the children's types and the path's types, we leverage a bag of words model [54]. For children's type in the C grammar, we define a bit set of size 267 that represents 266 internal grammar rule types and the keyword terminal used for the leaves of the parse tree. This bit set is of size 710 and 214 for Rust and Go grammars, respectively. If a rule type is present among the children's types, we set the corresponding element in the bit set to 1 and set it to 0 otherwise. Similarly, for the types of nodes on the path from the query node to the root, we define bit sets with 266, 709 and 213 elements for the C, Rust and Go grammars, respectively. Because a terminal cannot be on the path to the root[1], the bit set representing path's types has one element fewer than the bit set of

[1]The query node itself cannot be a terminal either due to not being syntactically removable based on the properties of quantified grammar rule types.

children's types for each grammar. Again, we set the elements of these bit sets to 1 if they are present on the path and set them to 0 otherwise.

We feed these features individually or in combination into our training algorithm. The next part explains the training process in more detail.

5.1.4 Training Models

With Boolean target values of our data points collected in Section 5.1.1, our problem is a classification problem in which we try to decide which class of removals (i.e., valid or invalid) a specific node removal belongs to.

To train our models, we use random forests [55], a well-known and efficient classifier that chooses the class with the most votes over all the trees in the forest. To avoid complexity, we consider more advanced training techniques, such as deep learning, as a potential future work, and instead exploit simple program properties to train intuitive models in order to better understand the potential benefits of using models in test case reduction [56].

For each data point in our training set, we feed the features described in Section 5.1.2 as source and the result of the node removal (i.e., success or failure) as target into our training algorithm.

We build the following five models:

- $M_{rf.node}$[2]: model trained using the grammar rule type of the query node as a feature.
- $M_{rf.children}$: model trained using the types of immediate children of the query node as features.
- $M_{rf.path}$: model trained using the types of nodes on the path from the query node to the root of the parse tree as features.
- $M_{rf.node.children}$: model trained using the types of the query node and its immediate children in combination as features.
- $M_{rf.node.children.path}$: model trained using the types of the query node, its children and nodes on the path from the query node to the root all in combination as features.

Note that the process of collecting data, extracting features, and training our models is fairly quick. For instance, it took us 18 hours in total to collect the data set, extract node types as features, and train the models for the C domain. This is a one-time overhead, and the majority is consumed by the data collection phase. Once data has been collected, multiple models can be trained easily using the same data set.

In the next section, we explain how we use these models in the process of test case reduction.

[2] rf is abbreviation for random forests.

Algorithm 5: Model Guided Pardis [29].

```
Input: $\tau_{✗}$ – The test case to reduce as a parse tree
Input: $\psi : \mathbb{S} \rightarrow \mathbb{B}$ – Oracle for the property to preserve where $\mathbb{S}$ is the search space and
        $\psi(\tau_{✗}) = True$
Input: $M : \mathbb{F} \rightarrow \mathbb{B}$ – Model to predict whether a node removal is semantically valid or not.
        $\mathbb{F}$ represents the set of input features.
Input: $Q = \{n_1, n_2, ..., n_N\}$ – Priority queue of syntactically removable parse tree nodes
Result: A minimum test case $\tau_{✗}' \subseteq \tau_{✗}$ s.t. $\psi(\tau_{✗}') = True$
1 $\tau_{✗}' \leftarrow \tau_{✗}$;
2 while !Q.empty() do
3     node ← Q.front();
4     features ← extractFeatures(node) ;
5     if M(features) && $\psi(\tau_{✗}' - node)$ then
6         $\tau_{✗}' \leftarrow \tau_{✗}'$ - node;
7     else
8         work.insert(getRemovableFrontier(node.children));
9 return $\tau_{✗}'$;
```

5.1.5 Model Guided Pardis: The Algorithm

We present Model Guided Pardis in algorithm 5. Given a test case $\tau_{✗}$ and an oracle function ψ, this approach leverages a predictive model M that can be easily integrated into the reduction process. Model Guided Pardis traverses a priority queue of syntactically removable nodes similar to Pardis. However, for each query node, rather than immediately querying the oracle, it collects its features using extractFeatures() on line 4 and queries the model instead on line 5. The return value of extractFeatures() depends on the type of the model chosen. For instance, it returns the unique ID of the node's grammar rule type for $M_{rf.node}$, while a bit set is returned for $M_{rf.children}$ and $M_{rf.path}$. A combination of features is returned for $M_{rf.node.children}$ and $M_{rf.node.children.path}$. Each of these models is a Boolean function that receives features of the query node as input and returns a predicted Boolean value.

By querying the model, Model Guided Pardis can decide whether to continue considering a node. If the model suggests that removing a node will be valid, the approach proceeds by running the oracle and, if the oracle also returns *True*, the reducer removes the query node and its descendants from the parse tree on line 6 because the smaller test case preserves the bug.

If the model predicts that removal will be invalid, Model Guided Pardis takes the other branch and adds the frontier of syntactically removable descendants to the priority queue (line 8). This frontier consists of those syntactically removable descendants that are the highest in the parse tree. In other words, a descendant node *will not* be added to the queue if it has an ancestor that is also a syntactically removable descendant of the query node.

Adding the frontier of syntactically removable descendant nodes is similar to adding the children of a node to the queue in algorithm 2. Here, we use frontier rather than children because the model is called *only* on syntactically removable nodes in algorithm 5. Moreover, we applied some syntactical removability pruning to optimize chains of nodes as presented in algorithm 4. The term frontier thus more accurately describes how we update the priority queue in algorithm 5.

The algorithm continues by attempting to remove the next node in the queue until the queue becomes empty or Model Guided Pardis times out. Note that the key difference between Model Guided Pardis and Pardis is that Model Guided Pardis avoids executing the oracle ψ when the model predicts that removing the query node is invalid. Algorithm 5 also runs to a fixed point similar to other tree based reduction techniques, including Perses and Pardis.

5.1.6 Evaluation

We evaluate the performance of our models by answering the following research questions:

- **RQ1.** How do different reducer models perform in terms of reduction time and efficiency, number of oracle queries and size of the reduced test case?
- **RQ2.** What are the precision and recall rates of our models?
- **RQ3.** What types of semantic issues (i.e., errors and warnings) are predicted correctly by our models and what types are not? What is the ratio of correct and incorrect predictions with respect to each error and warning type?

RQ1. Models' Performance

As described earlier in Section 5.1.4, we have five different types of models trained for each domain that we need to evaluate: $M_{rf.node}$, $M_{rf.children}$, $M_{rf.path}$, $M_{rf.node.children}$ and $M_{rf.node.children.path}$. Our benchmark and performance metrics are the same as in the previous studies and presented in Section 2.6.1 and Section 2.6.3.

Results of reduction on test cases of each domain using different types of models are shown in Table 5.2 and Table 5.3. Similar to the previous chapters, the first table shows the results of reduction by Model Guided Pardis where node replacements are disabled, while the second table shows the reduction results with enabled node replacements.

As can be seen in Table 5.2, for the C domain, $M_{rf.node.children.path}$, our model with the combined set of features has the best results for the majority of the cases with respect to the total number of tests, reduction time and efficiency. On average, Model Guided Pardis using this type of model reduces test cases of domain C in 55 seconds while Pardis has an average reduction time of 595 seconds. This is an improvement of more than 90%. However, a closer inspection of the table reveals that this improvement has come at a cost and that is the significantly larger outputs of Model Guided Pardis using $M_{rf.node.children.path}$ compared to Pardis. The outputs generated by Model Guided Pardis with this type of model are almost 10 times larger than outputs of Pardis on average. This might be the result of missing reduction opportunities caused by a large number of false negatives in $M_{rf.node.children.path}$, which we will fully discuss when answering our second research question in Section 5.1.6.

The next fastest model in our set of models for the C test cases in Table 5.2 is $M_{rf.path}$ with 98 seconds of reduction time on average. This model also generates outputs of relatively large size with 5,035 tokens on average. There is an interesting observation in the results of this model for one of the test cases, `gcc-61452`. Model Guided Pardis does not perform any oracle queries on this test case which means that the $M_{rf.path}$ model predicts semantic invalidity for all the variants, generating a significant number of false negatives. As a result, the reducer using this model skips tests that could be successful otherwise. Model $M_{rf.children}$ also suffers from the problem of false negatives when reducing the majority of the test cases. We thoroughly examine the accuracy of our models in Section 5.1.6. For now, we exclude those test cases with no reduction when computing the geomean and median values of each model and ignore them when highlighting the best values for each performance metric in our tables. From the remaining models, we have $M_{rf.node}$ and $M_{rf.node.children}$ that not only do they outperform Pardis by an average reduction time of 460 and 465 seconds, respectively but they also generate outputs of comparable size (648 and 766 tokens on average vs. 587 tokens of Pardis).

Similarly for the Rust domain, there are models such as $M_{rf.children}$, $M_{rf.path}$ and $M_{rf.node.children.path}$ that do not perform any reduction on all or some of the test cases in Table 5.2. Our best model that performs reduction on all the Rust test cases is $M_{rf.node}$ with an average reduction time of 471 seconds compared to the average reduction time of Pardis that is 701 seconds. This 33% improvement is achieved by generating outputs that are larger by only 52 tokens on average.

For the Go test cases in Table 5.2, $M_{rf.node.children.path}$ is the best model for reducing test case `go-29220`. This model decreases the overall number of tests performed on this test case and improves the reduction efficiency by more than two times while generating outputs of the same size compared to Pardis. The other useful model for the Go domain is $M_{rf.children}$ that decreases the total number of tests from 421 in Pardis to 18 in Model Guided Pardis when reducing `go-30606`, while maintaining a similar reduction power.

These results suggest that using models can indeed help with accelerating test case reduction. However, the type of the selected model can depend on the domain and the metric one is trying to improve. For instance, if generating larger outputs is negligible, $M_{rf.node.children.path}$ can be used for the C and Rust domains. Otherwise, $M_{rf.node}$ will be a more suitable option since, despite offering less improvement on reduction time, it generates outputs that are comparable in size to those generated by the state of the art techniques.

Table 5.2: The performance of Model Guided Pardis with different types of models. Node replacements disabled.

Test Case	$O_{(\#)}$	$\boldsymbol{M_{rf.node}}$				$\boldsymbol{M_{rf.children}}$				$\boldsymbol{M_{rf.path}}$				$\boldsymbol{M_{rf.node.children}}$				$\boldsymbol{M_{rf.node.children.path}}$				**Pardis**			
		$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$
clang-22382	21,069	1,113	1,196	3,720	5.36	21,069	0	1	0	4,871	655	1,477	10.97	1,425	1,222	3,746	5.24	12,759	36	72	115.42	867	1,977	3,856	5.24
clang-22704	184,445	389	4,142	5,234	35.17	326	4,055	4,073	45.20	73,694	202	854	129.69	326	5,046	6,653	27.67	70,267	179	724	157.70	326	4,055	4,243	43.39
clang-23309	38,648	2,372	1,316	565	64.21	38,648	0	1	0	10,750	102	90	309.98	2,358	1,461	596	60.89	10,750	95	79	353.14	2,129	2,897	998	36.59
clang-25900	78,961	1,056	1,053	496	157.07	78,961	0	2	0	8,809	58	38	1,846	1,033	1,120	511	152.50	8,795	70	52	1,349	883	1,582	577	135.32
clang-27747	173,841	595	901	596	290.68	498	1,271	778	222.81	7,252	193	165	1,010	908	877	507	341.09	7,431	165	126	1,321	498	1,271	656	264.24
clang-31259	48,800	699	729	388	123.97	608	1,474	725	66.47	4,856	193	113	388.88	700	1,101	531	90.58	9,288	59	47	840.68	608	1,474	783	61.55
gcc-59903	57,582	2,109	1,503	689	80.51	57,582	0	2	0	7,152	332	173	291.50	2,004	2,126	1,362	40.81	10,636	125	101	464.81	1,901	3,618	1,429	38.97
gcc-60116	75,225	1,558	1,722	751	98.09	75,225	0	3	0	6,547	767	380	180.73	1,895	1,908	778	94.25	11,707	223	183	347.09	1,361	3,718	1,907	38.73
gcc-61383	32,450	1,180	1,511	838	37.32	32,450	0	2	0	9,833	391	253	89.40	1,717	1,187	754	40.76	15,816	32	22	756.09	1,031	2,103	799	39.32
gcc-61452	26,733	1,356	1,177	2,449	10.36	26,733	0	1	0	26,733	0	2	0	1,178	1,762	2,764	9.25	10,091	62	93	178.95	1,023	1,685	2,444	10.52
gcc-61917	85,360	1,788	1,633	754	110.84	85,360	0	2	0	16,424	87	53	1,301	2,055	1,663	777	107.21	16,424	80	47	1,467	1,421	3,243	951	88.26
gcc-64990	148,932	1,377	1,241	708	208.41	148,932	0	4	0	9,439	373	207	673.88	1,819	1,450	742	198.27	13,363	128	83	1,633	1,127	2,768	944	156.57
gcc-65383	43,943	267	2,274	1,335	32.72	43,943	0	1	0	9,685	91	57	601.02	1,669	1,015	597	70.81	9,017	88	64	545.72	1,110	2,381	748	57.26
gcc-66186	47,482	1,432	1,078	442	104.19	47,482	0	1	0	9,280	71	37	1,032	1,461	1,282	475	96.89	9,722	43	28	1,349	1,288	1,877	544	84.92
gcc-71626	6,134	85	208	30	201.63	61	229	32	189.78	244	134	29	203.10	86	198	30	201.60	127	148	24	250.29	61	229	28	216.89
gcc-71632	141	107	29	18	1.89	82	114	61	0.97	112	13	6	4.83	109	26	11	2.91	112	14	6	4.83	82	114	55	1.07
gcc-77624	1,306	31	78	8	159.38	1,306	0	0	0	812	189	14	35.29	23	88	9	142.56	54	67	8	156.50	23	108	10	128.30
geomean	29,769	648	837	460	58.17	218*	724*	339*	41.51*	5,035*	161*	98*	219.17*	766	882	465	57.13	5,245	77	55	379.46	587	1,416	595	46.58
median	47,482	1,113	1,196	689	98.09	326*	1,271*	725*	66.47*	8,031*	191*	102*	300.74*	1,425	1,222	597	90.58	10,091	80	64	464.81	1,023	1,977	799	57.26
rust-44800	802	480	354	2,453	0.13	802	0	0	0	802	0	0	0	472	1,107	6,066	0.05	687	16	100	1.15	472	1,107	6,576	0.05
rust-63791	8,144	5,053	3,711	T/O	0.21	8,144	0	0	0	8,144	0	0	0	4,948	4,131	T/O	0.22	8,144	16	55	0	4,948	3,737	T/O	0.22
rust-65934	107	100	26	18	0.39	107	0	0	0	107	0	0	0	100	72	34	0.21	107	1	0	0	100	72	38	0.18
rust-69039	191	174	59	189	0.09	191	0	0	0	191	1	1	0	128	252	908	0.07	191	0	0	0	128	252	1,217	0.05
rust-77002	348	302	268	394	0.12	348	0	0	0	348	0	0	0	302	457	698	0.07	338	94	122	0.08	302	457	1,536	0.03
rust-77993	4,989	182	130	819	5.87	4,989	0	0	0	4,989	0	0	0	50	166	785	6.29	4,262	6	28	25.96	50	166	175	28.22
rust-78336	980	35	56	319	2.96	980	0	0	0	980	0	0	0	18	79	358	2.69	61	43	159	5.78	18	79	313	3.07
rust-78622	157	32	29	197	0.63	157	0	0	0	157	0	0	0	29	39	212	0.60	59	9	54	1.81	29	39	159	0.81
geomean	659	200	136	471	0.43	-*	-*	-*	-*	-*	-*	-*	-*	148	258	761	0.32	324*	20*	78*	1.90*	148	254	701	0.35
median	575	178	95	357	0.30	-*	-*	-*	-*	-*	-*	-*	-*	114	209	742	0.22	338*	16*	100*	1.81*	114	209	765	0.20
go-28390	146	145	57	10	0.10	146	0	0	0	146	11	2	0	146	0	0	0	146	0	0	0	84	127	18	3.44
go-29220	127	74	33	5	10.60	74	51	4	13.25	74	60	5	10.60	74	36	5	10.60	74	28	3	17.67	74	69	6	8.83
go-30606	449	423	259	70	0.37	432	18	5	3.40	423	265	69	0.38	423	271	69	0.38	449	0	0	0	423	421	100	0.26
geomean	203	166	79	15	0.73	179*	30*	4*	6.71*	177*	126*	19*	2.01*	177*	99*	19*	2.01*	74*	28*	3*	17.67*	138	155	22	1.99
median	146	145	57	10	0.37	253*	35*	5*	8.33*	249*	163*	37*	5.49*	249*	154*	37*	5.49*	74*	28*	3*	17.67*	84	127	18	3.44

O, R and Q denote number of tokens in the original test case, reduced one and total number of oracle queries performed by the reduction technique, respectively. T is the reduction time in seconds and E is the efficiency in terms of the number of tokens removed per second. Timeout (T/O) is set to 4 hours.

* Test cases with no reduction are excluded when computing geomean and median values. They are also excluded when highlighting the best values of performance metrics for each test case.

Table 5.3: The performance of Model Guided Pardis with different types of models. Node replacements enabled.

Test Case	$O_{(\#)}$	$M_{rf.node}$				$M_{rf.children}$				$M_{rf.path}$				$M_{rf.node.children}$				$M_{rf.node.children.path}$				Pardis			
		$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$
clang-22382	21,069	369	2,394	4,248	4.87	4,584	1,886	5,958	2.77	2,204	2,980	6,525	2.89	373	2,376	4,290	4.82	3,141	2,461	5,626	3.19	343	2,847	4,273	4.85
clang-22704	184,445	177	4,150	3,184	57.87	10,077	7,300	9,408	18.53	9,137	8,167	9,619	18.23	173	4,345	3,265	56.44	8,573	8,162	7,659	22.96	173	4,506	3,296	55.91
clang-23309	38,648	649	5,350	2,308	16.46	4,064	1,435	827	41.82	3,570	5,755	3,272	10.72	261	5,652	2,859	13.43	3,914	5,199	2,649	13.11	60	4,741	2,109	18.30
clang-25900	78,961	311	2,090	945	83.23	3,696	1,670	1,138	66.14	1,737	1,892	942	81.98	327	1,748	826	95.20	2,561	1,902	939	81.36	307	2,383	987	79.69
clang-27747	173,841	231	1,950	1,250	138.89	4,228	2,234	1,542	110.00	955	2,813	1,786	96.80	240	2,075	1,174	147.87	926	2,751	2,113	81.83	230	2,059	1,069	162.41
clang-31259	48,800	374	2,375	1,239	39.08	4,081	1,643	973	45.96	1,753	2,728	1,476	31.87	374	2,362	1,183	40.93	1,738	2,768	1,486	31.67	376	2,847	1,495	32.39
gcc-59903	57,582	416	4,564	8,363	6.84	392	5,479	8,990	6.36	2,558	4,365	4,309	12.77	419	4,066	6,199	9.22	2,418	4,731	5,871	9.40	392	5,479	8,947	6.39
gcc-60116	75,225	543	4,584	2,802	26.65	4,378	2,494	1,568	45.18	2,978	5,131	3,313	21.81	543	4,723	2,755	27.11	3,032	5,119	3,294	21.92	528	5,397	3,107	24.04
gcc-61383	32,450	297	2,414	1,444	22.27	7,763	1,424	955	25.85	5,771	2,279	1,393	19.15	603	1,687	1,079	29.52	5,731	2,158	1,313	20.35	288	2,532	1,284	25.05
gcc-61452	26,733	340	2,113	2,957	8.93	3,182	1,380	3,671	6.42	2,874	2,026	3,754	6.36	339	2,365	3,051	8.65	2,310	2,020	3,944	6.19	342	2,539	2,884	9.15
gcc-61917	85,360	333	3,096	1,397	60.86	8,778	1,943	982	77.99	5,341	3,152	1,530	52.30	348	3,120	1,579	53.84	6,258	3,095	1,438	55.01	329	3,580	1,265	67.22
gcc-64990	148,932	359	2,917	1,910	77.79	4,318	1,877	1,453	99.53	3,064	2,580	1,956	74.57	383	2,928	1,783	83.31	2,921	2,792	2,148	67.98	354	2,696	1,248	119.05
gcc-65383	43,943	244	2,264	1,536	28.45	4,041	1,614	1,400	28.50	1,810	2,722	2,199	19.16	235	2,549	1,690	25.86	1,269	2,929	2,141	19.93	279	2,130	1,151	37.94
gcc-66186	47,482	398	2,235	1,300	36.22	3,819	1,525	1,054	41.43	1,173	3,141	2,134	21.70	404	2,008	1,224	38.46	2,577	2,050	1,261	35.61	391	2,085	914	51.52
gcc-71626	6,134	84	278	34	177.94	60	299	32	189.81	215	200	27	219.22	85	268	32	189.03	126	347	35	171.66	60	299	32	189.81
gcc-71632	141	88	198	41	1.29	75	173	54	1.22	81	120	40	1.50	89	196	43	1.21	91	188	39	1.28	75	173	54	1.22
gcc-77624	1,306	30	175	16	79.75	904	119	8	50.25	812	189	14	35.29	30	172	13	98.15	39	172	16	79.19	22	205	15	85.60
geomean	29,769	252	1,811	965	28.89	2,279	1,357	880	27.85	1,746	1,921	1,131	22.42	252	1,768	926	30.07	1,512	1,960	1,113	23.75	207	1,933	912	31.01
median	47,482	333	2,375	1,444	36.22	4,064	1,643	1,138	41.82	2,204	2,728	1,956	21.70	339	2,365	1,579	38.46	2,561	2,751	2,113	22.96	307	2,539	1,265	37.94
rust-44800	802	472	2,796	14,305	0.02	716	707	3,953	0.02	716	707	3,948	0.02	464	3,525	12,630	0.03	648	1,384	2,848	0.05	464	2,437	T/O	0.02
rust-63791	8,144	6,301	3,270	T/O	0.13	6,704	5,688	T/O	0.10	6,604	8,425	T/O	0.11	5,859	8,534	T/O	0.16	6,604	8,518	T/O	0.11	7,923	1,339	T/O	0.02
rust-65934	107	100	120	59	0.12	107	47	24	0	107	47	26	0	100	166	79	0.09	107	47	25	0	100	166	368	0.02
rust-69039	191	169	420	2,712	0.01	191	186	447	0	191	186	194	0	120	689	1,043	0.07	191	186	200	0	120	689	3,528	0.02
rust-77002	348	286	3,469	2,428	0.03	324	1,078	2,083	0.01	324	1,114	3,849	0.01	286	3,646	11,707	0.01	324	1,159	3,661	0.01	286	3,646	9,657	0.01
rust-77993	4,989	132	288	1,602	3.03	1,860	276	1,277	2.45	1,860	276	1,246	2.51	16	169	966	5.15	91	170	950	5.16	16	169	1,079	4.61
rust-78336	980	32	56	284	3.34	691	88	547	0.53	691	88	457	0.63	15	73	346	2.79	628	92	508	0.69	15	73	352	2.74
rust-78622	157	32	30	128	0.98	79	7	56	1.39	79	7	16	4.88	29	40	202	0.63	50	9	38	2.82	29	40	178	0.72
geomean	659	193	408	1,212	0.18	735*	301*	1,291*	0.18*	733*	323*	1,122*	0.24*	126	530	1,362	0.20	398*	353*	1,184*	0.29*	131	401	1,908	0.10
median	575	151	354	2,015	0.13	704*	492*	1,680*	0.32*	704*	492*	2,548*	0.37*	110	429	1,005	0.13	476*	665*	1,899*	0.40*	110	429	2,304	0.02
go-28390	146	140	107	19	0.32	141	27	4	1.25	141	38	5	1.00	141	27	4	1.25	141	38	5	1.00	84	163	24	2.58
go-29220	127	63	134	18	3.56	60	145	15	4.47	60	149	17	3.94	60	130	13	5.15	65	131	15	4.13	60	158	16	4.19
go-30606	449	213	700	161	1.47	222	611	141	1.61	219	690	152	1.51	222	611	136	1.67	229	305	67	3.28	213	794	179	1.32
geomean	203	123	216	38	1.19	123	134	20	2.08	123	157	23	1.81	123	129	19	2.21	128	115	17	2.38	102	273	41	2.43
median	146	140	134	19	1.47	141	145	15	1.61	141	149	17	1.51	141	130	13	1.67	141	131	15	3.28	84	163	24	2.58

O, R and Q denote number of tokens in the original test case, reduced one and total number of oracle queries performed by the reduction technique, respectively. T is the reduction time in seconds and E is the efficiency in terms of the number of tokens removed per second. Timeout (T/O) is set to 4 hours.

* Test cases with no reduction are excluded when computing geomean and median values. They are also excluded when highlighting the best values of performance metrics for each test case.

Table 5.4: The speed up achieved by Model Guided Pardis using each type of model with respect to Pardis.

Test Case	Replacements disabled					Replacements enabled				
	$M_{rf.node}$	$M_{rf.children}$	$M_{rf.path}$	$M_{rf.node}$ $_{.children}$	$M_{rf.node.}$ $_{children.path}$	$M_{rf.node}$	$M_{rf.children}$	$M_{rf.path}$	$M_{rf.node}$ $_{.children}$	$M_{rf.node.}$ $_{children.path}$
clang-22382	1.04	N/A*	2.61	1.03	53.56	1.01	0.72	0.65	1.00	0.76
clang-22704	0.81	1.04	4.97	0.64	5.86	1.04	0.35	0.34	1.01	0.43
clang-23309	1.77	N/A*	11.09	1.67	12.63	0.91	2.55	0.64	0.74	0.80
clang-25900	1.16	N/A*	15.18	1.13	11.10	1.04	0.87	1.05	1.19	1.05
clang-27747	1.10	0.84	3.98	1.29	5.21	0.86	0.69	0.60	0.91	0.51
clang-31259	2.02	1.08	6.93	1.47	16.66	1.21	1.54	1.01	1.26	1.01
gcc-59903	2.07	N/A*	8.26	1.05	14.15	1.07	1.00	2.08	1.44	1.52
gcc-60116	2.54	N/A*	5.02	2.45	10.42	1.11	1.98	0.94	1.13	0.94
gcc-61383	0.95	N/A*	3.16	1.06	36.32	0.89	1.34	0.92	1.19	0.98
gcc-61452	1.00	N/A*	N/A*	0.88	26.28	0.98	0.79	0.77	0.95	0.73
gcc-61917	1.26	N/A*	17.94	1.22	20.23	0.91	1.29	0.83	0.80	0.88
gcc-64900	1.33	N/A*	4.56	1.27	11.37	0.65	0.86	0.64	0.70	0.58
gcc-65383	0.56	N/A*	13.12	1.25	11.69	0.75	0.82	0.52	0.68	0.54
gcc-66186	1.23	N/A*	14.70	1.15	19.43	0.70	0.87	0.43	0.75	0.72
gcc-71626	0.93	0.88	0.97	0.93	1.17	0.94	1.00	1.19	1.00	0.91
gcc-71632	3.06	0.90	9.17	5.00	9.17	1.32	1.00	1.35	1.26	1.38
gcc-77624	1.25	N/A*	0.71	1.11	1.25	0.94	1.88	1.07	1.15	0.94
geomean	1.29	0.94**	5.54**	1.28	10.81	0.95	1.04	0.81	0.99	0.82
median	1.23	0.90**	5.98**	1.15	11.69	0.94	1.00	0.83	1.00	0.88
rust-44800	2.68	N/A*	N/A*	1.08	65.76	1.01	3.64	3.65	1.14	5.06
rust-63791	1.00	N/A*	N/A*	1.00	N/A*	1.00	1.00	1.00	1.00	1.00
rust-65934	2.11	N/A*	N/A*	1.12	N/A*	6.24	N/A*	N/A*	4.66	N/A*
rust-69039	6.44	N/A*	N/A*	1.34	N/A*	1.30	N/A*	N/A*	3.38	N/A*
rust-77002	3.90	N/A*	N/A*	2.20	12.59	3.98	4.64	2.51	0.82	2.64
rust-77993	0.21	N/A*	N/A*	0.22	6.25	0.67	0.84	0.87	1.12	1.14
rust-78336	0.98	N/A*	N/A*	0.87	1.97	1.24	0.64	0.77	1.02	0.69
rust-78622	0.81	N/A*	N/A*	0.75	2.94	1.39	3.18	11.13	0.88	4.68
geomean	1.49	-**	-**	0.92	7.86**	1.57	1.75**	2.02**	1.40	1.91**
median	1.56	-**	-**	1.04	6.25**	1.27	2.09**	1.76**	1.07	1.89**
go-28390	1.80	N/A*	N/A*	N/A*	N/A*	1.26	6.00	4.80	6.00	4.80
go-29220	1.20	1.50	1.20	1.20	2.00	0.89	1.07	0.94	1.23	1.07
go-30606	1.43	20.00	1.45	1.45	N/A*	1.11	1.27	1.18	1.32	2.67
geomean	1.46	5.48**	1.32**	1.32**	2.00**	1.08	2.01	1.75	2.14	2.39
median	1.43	10.75**	1.33**	1.33**	2.00**	1.11	1.27	1.18	1.32	2.67

* No speed up is calculated for models and test cases with no reduction.

** Test cases with no reduction are excluded when computing geomean and median values.

Table 5.4 also depicts the speed up achieved by using each model compared to Pardis by dividing the reduction time of Pardis by the reduction time of Model Guided Pardis using each model. Although leveraging models in reduction when node replacements are enabled can outperform Pardis in some cases, the improvement achieved by our best model for each domain is more significant when node replacements are disabled. One explanation could be that our models are trained with respect to removal and they do not reason about valid or invalid replacements. Moreover, as we will show in Section 5.1.6, our models have false negatives which can generate larger test case variants throughout the reduction process by missing reduction opportunities. As a result, performing node replacements on a larger variant with more number of nodes may take longer. Training models with higher accuracy that are also capable of predicting the semantic validity of a node replacement can help in this regard. We will further discuss this in Section 6.3.2.

Finally, we provide the average execution time of each type of model compared to the average execution time of the oracle for each test case in our sample benchmark. As can

Table 5.5: The execution time of different types of models compared to the execution time of the oracle. The execution times are in milliseconds.

Test Case	$M_{rf.node}$	$M_{rf.children}$	$M_{rf.path}$	$M_{rf.node.children}$	$M_{rf.node.children.path}$	Oracle	Speed Up (×)
clang-25900	2	2	2	2	4	365	91
clang-31259	2	1	2	1	1	531	266
gcc-60116	2	2	2	2	3	513	171
gcc-61383	2	2	2	2	2	380	190
gcc-77624	0.26	0.25	0.28	0.25	0.30	93	310
geomean	1.33	1.15	1.35	1.15	1.48	323	189
median	2	2	2	2	2	380	190
rust-63791	3	4	4	3	4	3,853	963
rust-65934	0.3	0.3	0.3	0.3	0.3	528	1,760
rust-77993	1	3	2	1	3	1,054	351
geomean	0.97	1.53	1.34	0.97	1.53	1,290	841
median	1	3	2	1	3	1,054	963
go-30606	0.6	1	0.9	0.6	0.7	238	238

be seen in Table 5.5, the execution of every individual model is substantially faster than executing an oracle. The speed up metric is computed by dividing the execution time of the oracle by the execution time of the model with the longest execution time. The significant speed up achieved for each test case further motivates the use of inexpensive models to avoid costly oracle checks when it is possible.

The results presented in this section suggest that by leveraging simple grammar properties, we can select an appropriate set of features such as the rule types of nodes in the parse tree of the test case to train models that are capable of speeding up syntax guided reduction by trying to avoid performing tests that are likely to be semantically invalid.

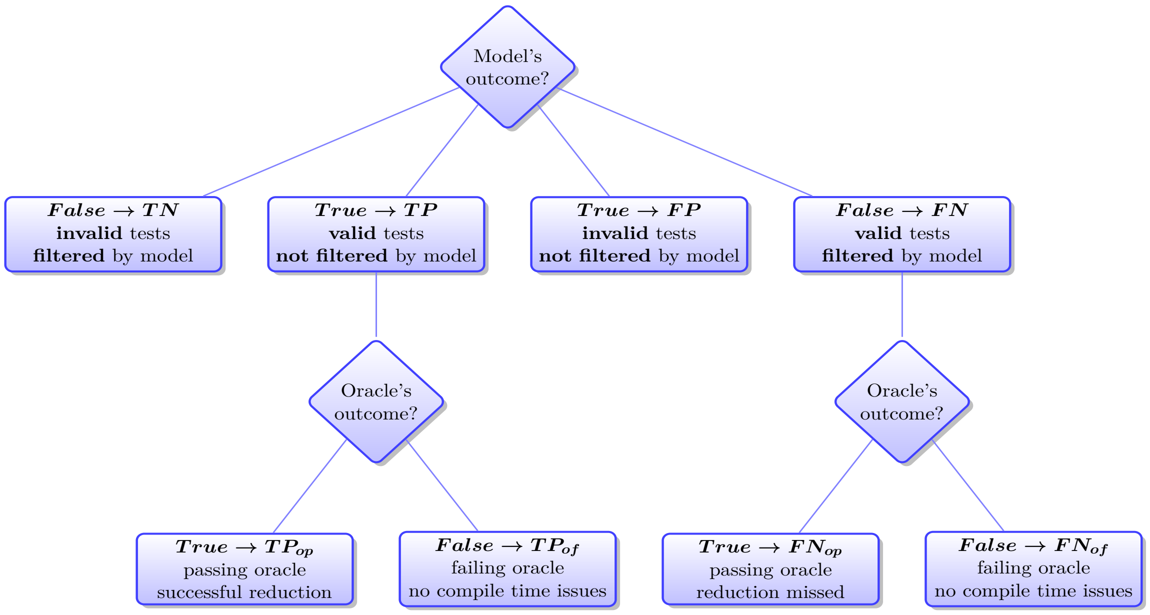
Next, we measure the accuracy metrics for our models on two types of test sets, synthetic and real. More details regarding these measures and the two test sets are provided in the following section.

RQ2. Precision and Recall Rates

In this section, we discuss the possible categories of our models' outcomes (the predicted values) with respect to the oracles' outcomes (the actual values) and compute the precision and recall rates of our models.

Models' outcomes. Figure 5.4 depicts the categories of our models' outcomes with respect to the outcome of an oracle query. As can be seen, the predicted outcomes of our models belong to one of the following categories:

- True positive (TP): Model *correctly* predicts that removal of the query node is *semantically valid.* After passing the semantic validity checks, these tests will be subdivided into two subcategories:
  - Oracle pass (TP_{op}): The return value of the oracle query is $True$ for these tests, leading to a successful removal and reducing the test case.
  - Oracle failure (TP_{of}): These tests fail to remove the node from the parse tree of the test case. However, their failure is caused by reasons other than compile time


Model's outcome?
False → *TN* **invalid** tests **filtered** by model
True → *TP* **valid** tests **not filtered** by model
True → *FP* **invalid** tests **not filtered** by model
False → *FN* **valid** tests **filtered** by model
Oracle's outcome?
Oracle's outcome?
True → TP_{op} passing oracle successful reduction
False → TP_{of} failing oracle no compile time issues
True → FN_{op} passing oracle reduction missed
False → FN_{of} failing oracle no compile time issues
Figure 5.4: A breakdown of models' outcomes.

semantic invalidity. They may fail due to either run time issues such as memory leaks in the test case variant or not preserving the property of interest.

- True negative (TN): Model *correctly* predicts that removal of the node is *semantically invalid*.
- False positive (FP): Model *falsely* predicts that removal of the query node is *semantically valid*.
- False negative (FN): Model *falsely* predicts that removal of the query node is *semantically invalid*. These tests *will not* be executed by MODEL GUIDED PARDIS due to a false prediction. If these tests were to be executed, they would have the subcategories of oracle pass (FN_{op}) and oracle failure (FN_{of}) similar to the category of true positives above.

While the false positives may decrease the improvement brought by MODEL GUIDED PARDIS by allowing for execution of tests on semantically invalid variants, false negatives can cause MODEL GUIDED PARDIS to generate larger outputs by missing potentially successful tests on semantically valid variants.

Distribution of models' outcome categories. To measure the accuracy of our models, we compute the distribution of true positives, true negatives, false positives and false negatives for each model on two types of data sets:

1. Synthetic test sets: As described earlier in Section 5.1.1, for training models in each domain, we use 75% of our collected data points while keeping the remaining 25% as

Table 5.6: The distribution of models' outcomes for our synthetic test sets.

Domain	Model	***TP***		***TN***		***FP***		***FN***		**Precision** (%)	**Recall** (%)
		#	%	#	%	#	%	#	%		
C	$M_{rf.node}$	45,908	64	13,301	19	9,474	13	3,028	4	83	94
	$M_{rf.children}$	44,709	62	14,592	20	8,183	12	4,227	6	85	91
	$M_{rf.path}$	47,351	66	14,197	20	8,578	12	1,585	2	85	97
	$M_{rf.node.children}$	44,709	62	14,592	20	8,183	12	4,227	6	85	91
	$M_{rf.node.children.path}$	46,185	64	15,541	22	7,234	10	2,751	4	86	94
Rust	$M_{rf.node}$	20,366	44	17,515	38	5,707	12	2,830	6	78	88
	$M_{rf.children}$	20,582	44	17,012	37	6,150	13	2,674	6	77	89
	$M_{rf.path}$	20,792	45	18,199	39	5,034	11	2,393	5	81	90
	$M_{rf.node.children}$	20,608	45	17,344	37	5,666	12	2,800	6	78	88
	$M_{rf.node.children.path}$	21,064	45	18,148	39	5,105	11	2,101	5	80	91
Go	$M_{rf.node}$	3,419	47	2,720	38	938	13	158	2	78	96
	$M_{rf.children}$	3,497	48	2,601	36	976	14	161	2	78	96
	$M_{rf.path}$	3,463	48	2,917	40	669	9	186	3	84	95
	$M_{rf.node.children}$	3,433	48	2,694	37	938	13	170	2	79	95
	$M_{rf.node.children.path}$	3,390	47	2,949	41	703	9	193	3	83	95

a test set. We refer to these test sets as *synthetic* because they have been collected during reduction of *non-buggy* test cases.

2. Real test sets: These test sets are collected when reducing test cases of our sample benchmark. Since the values in these sets have been collected when reducing test cases with *real bugs*, we refer to these sets as *real* test sets.

Additionally, we compute the precision and recall rates of each model on each test set.

Table 5.6 depicts the results of accuracy metrics described above for our synthetic test sets. For the C domain, all our models have precision and recall rates above 80% and 90%, respectively. The precision rates of our Rust models are between 77% and 81% with recall rates between 88% and 91%. The Go models also show a high accuracy by precision rates above 78% and recall rates above 95% on our Go synthetic test set.

To measure the accuracy metrics on our real test sets, we perform reduction by Model Guided Pardis, using different types of models on our sample benchmark with a slight modification to record the model's prediction along with the actual outcome of the oracle for each test.

To this end, we define three types of actual outcome for an oracle:

- Passed: Test was successful. Oracle passed and node got removed.
- Compile time semantic checks failed: Test was unsuccessful. Oracle failed due to the semantic invalidity of the test case variant during compilation and node did not get removed.
- Run time or property checks failed: Test was unsuccessful. Compile time semantic checks passed but oracle failed due to either failing the run time constraints or not preserving the bug.

Table 5.7: The distribution of models' outcomes for our real test sets.

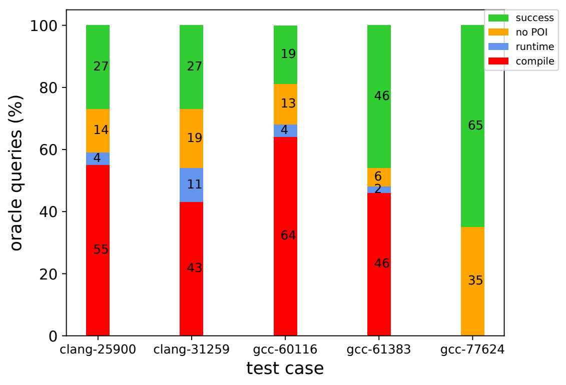
Model	Test Case	$\boldsymbol{TP_{op}}$		$\boldsymbol{TP_{of}}$		$\boldsymbol{TN}$		$\boldsymbol{FP}$		$\boldsymbol{FN_{op}}$		$\boldsymbol{FN_{of}}$		**Precision** (%)	**Recall** (%)
		#	%	#	%	#	%	#	%	#	%	#	%		
$M_{rf.node}$	clang-25900	396	25	188	12	543	34	333	21	25	2	97	6	64	83
	clang-31259	373	25	169	11	444	30	185	13	27	2	276	19	75	64
	gcc-60116	664	18	317	8	1,768	48	617	17	59	1	293	8	61	74
	gcc-61383	937	45	111	5	598	28	374	18	22	1	61	3	74	93
	gcc-77624	65	60	13	12	0	0	0	0	5	5	25	23	100	72
	rust-63791	123	5	269	11	1,179	45	826	33	21	1	122	5	32	73
	rust-65934	1	1	0	0	45	63	26	36	0	0	0	0	4	100
	rust-77993	80	23	47	13	6	2	3	1	122	35	91	26	98	37
	go-30606	17	4	242	58	0	0	0	0	0	0	162	38	100	62
$M_{rf.children}$	clang-25900	0	0	0	0	876	55	0	0	421	27	285	18	N/A*	0
	clang-31259	400	27	446	30	0	0	628	43	0	0	0	0	57	100
	gcc-60116	0	0	0	0	2,391	64	0	0	715	19	612	17	N/A*	0
	gcc-61383	0	0	0	0	949	46	0	0	959	47	154	7	N/A*	0
	gcc-77624	0	0	0	0	0	0	0	0	69	83	14	17	N/A*	0
	rust-63791	0	0	0	0	2,070	70	0	0	516	18	355	12	N/A*	0
	rust-65934	0	0	0	0	36	97	0	0	1	3	0	0	N/A*	0
	rust-77993	0	0	0	0	26	1	0	0	2,501	94	129	5	N/A*	0
	go-30606	17	4	1	0	0	0	0	0	2	0	411	96	100	4
$M_{rf.path}$	clang-25900	112	7	90	6	802	50	74	5	309	20	195	12	73	29
	clang-31259	7	0	66	4	600	41	29	2	393	27	379	26	72	9
	gcc-60116	79	2	226	6	1,985	54	411	11	634	17	383	10	43	23
	gcc-61383	120	6	59	3	894	42	81	4	836	40	113	5	69	16
	gcc-77624	24	22	14	13	0	0	0	0	46	43	24	22	100	35
	rust-63791	0	0	0	0	2,032	73	0	0	428	15	325	12	N/A*	0
	rust-65934	0	0	0	0	36	97	0	0	1	3	0	0	N/A*	0
	rust-77993	0	0	0	0	26	1	0	0	2,656	94	133	5	N/A*	0
	go-30606	1	0	242	56	0	0	0	0	32	7	162	37	100	56
$M_{rf.node.children}$	clang-25900	387	24	154	10	501	32	375	24	34	2	131	8	59	77
	clang-31259	376	25	189	13	362	25	259	18	32	2	256	17	69	66
	gcc-60116	650	18	277	7	1,603	43	792	21	63	2	333	9	54	70
	gcc-61383	890	42	27	1	830	40	142	7	69	3	145	7	87	81
	gcc-77624	65	60	10	9	0	0	0	0	5	5	28	26	100	69
	rust-63791	199	7	404	13	0	0	2,427	80	0	0	0	0	20	100
	rust-65934	1	1	0	0	0	0	70	98	0	0	1	1	1	50
	rust-77993	83	50	73	44	0	0	10	6	0	0	0	0	94	100
	go-30606	17	4	1	0	0	0	0	0	2	0	411	96	100	4
$M_{rf.node.children.path}$	clang-25900	15	1	26	2	856	54	18	1	406	26	261	16	69	6
	clang-31259	11	1	40	3	615	42	14	1	389	26	405	27	78	6
	gcc-60116	25	1	82	2	2,364	63	31	1	688	19	528	14	78	8
	gcc-61383	3	0	18	1	966	46	9	0	953	46	154	7	70	2
	gcc-77624	62	57	14	13	0	0	0	0	8	8	24	22	100	70
	rust-63791	0	0	0	0	2,081	69	0	0	564	19	370	12	N/A*	0
	rust-65934	0	0	0	0	35	97	0	0	1	3	0	0	N/A*	0
	rust-77993	2	0	0	0	31	1	0	0	2,606	93	157	6	100	0
	go-30606	0	0	0	0	0	0	0	0	18	8	206	92	N/A*	0

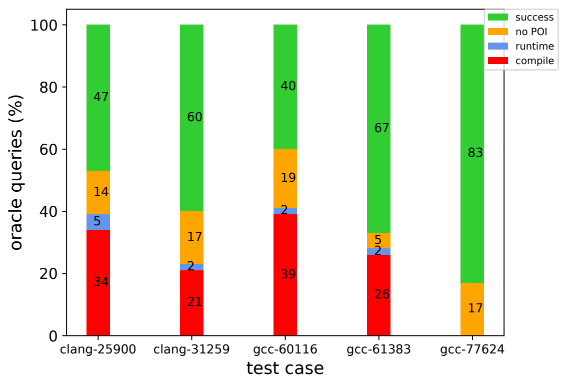
* N/A represents precision rates for those models and test cases that have no true and false positives in their reduction.

Note that although we guide the reduction process by our models' suggestions, we also query the oracle every time we attempt to remove a node, regardless of the models' predictions. This is how we can have pairs of values with predicted and actual outcomes used to compute the accuracy metrics for each model.

Table 5.7 shows the precision and recall rates of our models for an actual reduction of test cases in our sample benchmark. As can be seen, $M_{rf.node}$ and $M_{rf.node.children}$ have precision and recall rates above 60% in the majority of the cases. However, despite performing well on the synthetic test set, $M_{rf.children}$, $M_{rf.path}$ and $M_{rf.node.children.path}$ perform poorly on the real test set. They have low recall rates that indicate a large number of false negatives for these models. This means that these models may miss reduction opportunities. As a result, we observed that no reduction was performed by these models on some of the test cases in Table 5.2 in contrast to the other two models, $M_{rf.node}$ and $M_{rf.node.children}$ that have higher recall rates and more reduction power.

We can infer from these observed results that the grammar rule type of a node in the parse tree can be a useful piece of information for training models with high precision and recall rates.


(a) Pardis no model


(b) Model Guided Pardis $M_{rf.node}$

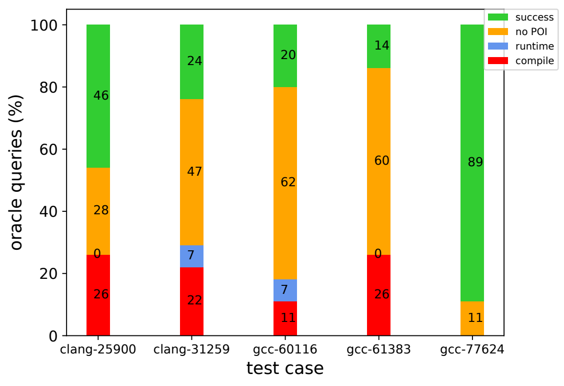

(c) Model Guided Pardis $M_{rf.node.children.path}$

Figure 5.5: A breakdown of the outcomes of the oracle queries performed by Pardis and Model Guided Pardis.

RQ3. Prediction of semantic invalidity types

In this section, we measure the performance of our models in terms of the total number of tests performed on variants with compile time issues. Moreover, we identify the correct and incorrect predictions of our models with respect to different types of semantic issues. More precisely, we examine which error and disallowed warning types are predicted correctly by our models and which ones are not.

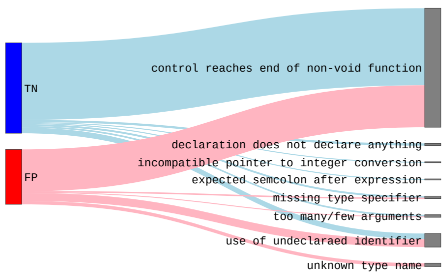
Recall the distribution of the oracle outcomes generated by Pardis in Figure 5.1 at the beginning of this chapter. Oracle queries with compile time issues comprised the plurality of the tests for 4 out of 5 test cases in this figure. The large portion of these queries can adversely affect the reduction speed as previously noted. To understand the impact of leveraging models on the number of this type of oracle outcomes, we select our two best models for the C domain, identified in the previous sections, and perform reduction by Model Guided Pardis using these models to collect its types of oracle outcomes. The two models selected are $M_{rf.node}$, our most effective model that generates outputs of similar size compared to Pardis and $M_{rf.node.children.path}$ that is our most efficient model, yielding the highest speed up in reduction.

Figure 5.5 depicts the distributions of oracle outcomes for Pardis and Model Guided Pardis with $M_{rf.node}$ and $M_{rf.node.children.path}$. As can be seen, Model Guided Pardis using these two types of models has distributions where compile time issues do not comprise the majority of the tests. Interestingly, the majority of the tests become successful when integrating $M_{rf.node}$ into Pardis to perform a model guided reduction. In the distribution of $M_{rf.node.children.path}$, the percentage of oracle outcomes with compile time issues also decreases and tests that do not preserve the property of interest (no POI) comprise the majority of the oracle outcomes instead. These results suggest that leveraging these models can indeed mitigate the problem of compile time issues generated during reduction. Generating fewer invalid oracle outcomes of this type can increase the efficiency of test case reduction as also shown in our results in Table 5.2. However, as Figure 5.5 also suggests, there is still room for improvement. Leveraging models in other forms with other purposes can be interesting directions. We investigate some of these models in Section 5.2.

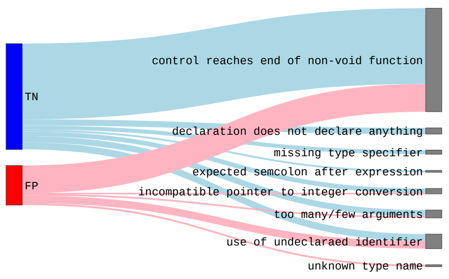
Finally, to understand what types of semantic issues are filtered or missed by Model Guided Pardis, we select $M_{rf.node}$ that is our model with a larger number of tests and investigate two sets of its outcomes: true negatives that are the frequent errors and warnings correctly filtered by our model and false positives that are the common errors and warnings falsely missed by our model.

Figure 5.6 depicts the breakdown of true negatives (TN) and false positives (FP) for $M_{rf.node}$ on the four C test cases in Figure 5.5. First, we can see that the number of true negatives for these cases is indeed larger than the number of false positives. Moreover, $M_{rf.node}$ is able to filter most of the frequent errors and disallowed warnings in the majority of the cases. In particular, control reaches end of non-void function is filtered correctly in most of the cases. Use of undeclared identifier is another common error that is partially handled by our model. Other errors and warnings such as declaration does not declare anything, expected semicolon after expression and missing type specifier are also mainly filtered. Unknown type name seems to be the only error in our list of common compile time issues that has a larger number of false positives compared to true negatives. However, as can be seen in the diagrams, this error comprises a small portion of semantic issues.

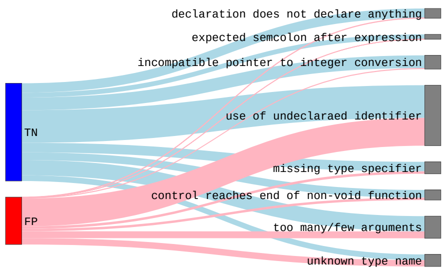
The data presented in this section demonstrate promising results for leveraging models in test case reduction. However, these models are still in the initial stage and steps to improve them could indeed increase the benefits of using them. The next section describes some of our measures taken for this purpose.



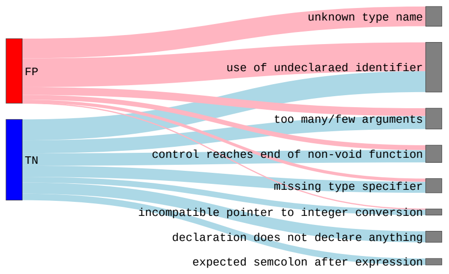
(a) clang-25900



(b) clang-31259



(c) gcc-60116



(d) gcc-61383

Figure 5.6: Semantic validity issues filtered (TN) and missed (FP) by $M_{rf.node}$.

5.2 Guiding Towards Semantic Validity

We originally introduced and published the ideas presented in this section at the 32nd ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023) [30].

So far in this chapter, we proposed and assessed Model Guided Pardis, a reducer tool based on Pardis that leverages trained models to predict and avoid performing tests on semantically invalid test case variants. Although Pardis enhanced by models shows promising results with respect to speeding up the reduction process, it still has limitations.

In this section, we propose a new technique called Type Batched Probabilistic Joint Reduction [30] to address the following drawbacks:

1. Model Guided Pardis traverses the parse tree in a strict top down fashion to prioritize nodes with a larger token weight. This can postpone visiting nodes located at lower levels of the tree that have a high likelihood of removal success.

2. Model Guided Pardis simply avoids querying an oracle on a test case variant that is predicted as semantically invalid by its models. For example, Model Guided Pardis may predict that removing a declaration node is always semantically invalid. This prediction may be useful when the use of that declaration is still present in the tree. However, once the use is removed, the declaration becomes removable and a better reducer should then be able to try removing the declaration. Model Guided Pardis does not have this dynamic reasoning about the reduction progress.

3. Model Guided Pardis removes nodes one at a time and lacks an efficient mechanism to enable successful group removals.

More specifically, we first propose a technique called Type Batched Reducer to directly address the first two limitations mentioned above. We further extend our Type Batched Reducer by introducing a Probabilistic Joint Reduction technique to address the third limitation. Moreover, we empirically demonstrate that Type Batched Reducer and Probabilistic Joint Reduction work synergistically to accelerate test case reduction.

5.2.1 Type Batched Test Case Reduction

The idea of Type Batched Test Case Reduction is to select an appropriate ordering among tests such that the following behaviors become possible:

1. The ordering allows for visiting nodes at lower levels of the tree with high likelihood of removal success early in the reduction process.

2. The ordering enables performing tests with higher likelihood of success *before* tests with lower likelihood of success. The likelihood of success for tests may change during reduction to better reflect an appropriate ordering. A good ordering, for instance, suggests removing uses before declarations. In other words, an appropriate ordering can *guide* reduction towards removing portions that are more likely to be successfully removed at a given point in time during reduction.

To better understand the differences between the Type Batched Reducer and the traversal based techniques such as Perses, Pardis and its variants introduced so far, consider the program in Listing 5.2 with its parse tree shown in Figure 5.7. This example is the same as the motivating example presented throughout the dissertation with a small change on line 13 where $d * i$ is replaced with d/i. Again, the property of interest to preserve when reducing this program is printing Hello World!.

As mentioned earlier in this chapter, the problem of semantic dependencies between elements within this program is evident. For instance, the use of type S depends on its definition. Similarly, calling function foo requires the definition of foo to be present. To mitigate this problem, as previously stated, there is the option of manually constructing these dependencies and using them in the search process. However, this would require detailed knowledge about the semantics of the input format. A reducer hand tailored based on domain knowledge would no longer be domain agnostic. It would require significant labor to be customized to the specific language and domain.

```
1   int d = 10;
2   struct S {
3       int f1;
4       int f2;
5   };
6   void foo() {
7       struct S s1 = {1, 2};
8       int i = 0;
9       bool increment = true;
10      if (increment) {
11          i += 2*i + i + 1;
12      }
13      s1.f1 = d/i;
14      printf("Hello World!\n");
15  }
16  int main() {
17      foo();
18      return 0;
19  }
```

Listing 5.2: A C program with a statement to preserve on line 14.

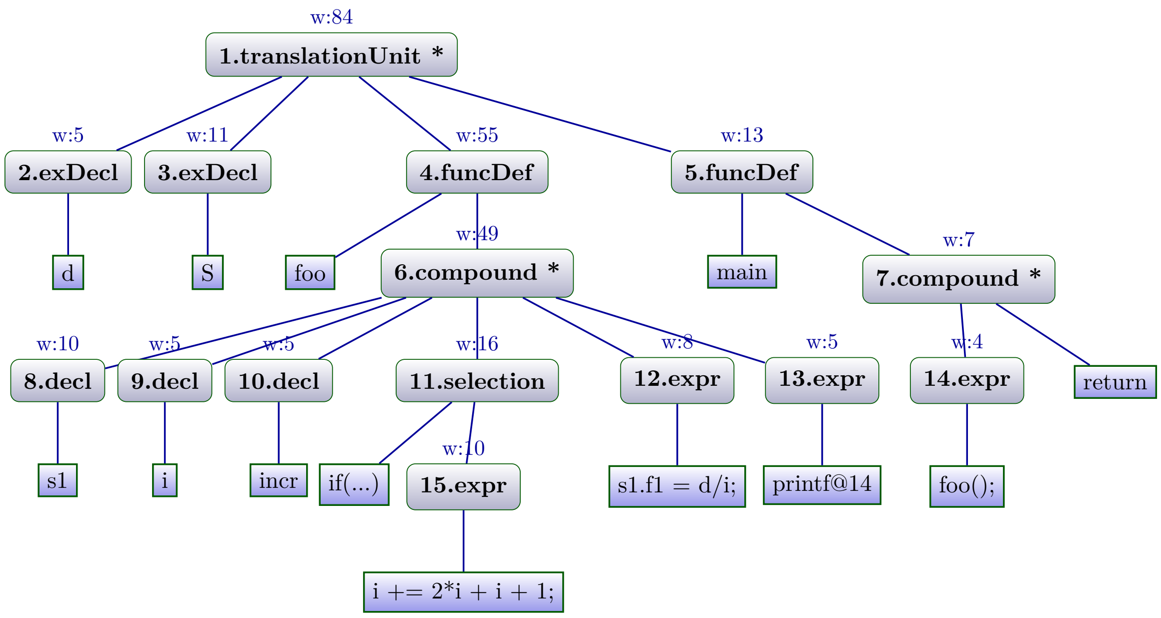


Figure 5.7: The simplified parse tree of program in Listing 5.2. Each internal node is annotated with an ID and its grammar rule type. * denotes a quantified Kleene-Star node.

In contrast, the key insight of domain agnostic Type Batched Reducer is that dependencies can be approximately inferred and modeled in a generalized fashion such that not only can a node removal causing semantic invalidity be predicted and avoided similar to what Model Guided Pardis does but also those portions of the test case that are more likely to be semantically valid at a given point during reduction are identified and reduction is guided to reduce them first.

To this end, grammar rule types can be used again to give us information about the possible dependencies. For example in Figure 5.7, the grammar rules for parsing expression statements differ from those used to parse function and type declarations. This time, instead of simply avoiding tests on declarations, we can first try removing expression statement nodes and only afterward try removing declarations. This *preserves* the dependencies that a naive traversal may violate, but it only directly uses information from the grammar. The general pattern is that it can be advantageous to consider reduction on some set of nodes before others, and the types of internal nodes provide leverage in making that decision.

We discuss how to choose an ordering over different grammar types in Section 5.2.2. For now, suppose that an appropriate ordering among types of nodes is given. Instead of traversing the parse tree and attempting to perform expensive reduction operations at *every* node, Type Batched Reducer selects one type at a time and only operates on nodes of that selected type during the traversal. Once all nodes of the selected type have been considered, it selects one of the remaining unselected types and traverses the tree again. After all types have been selected *or* it looks unprofitable to continue type batching, a traversal based syntax guided approach such as Pardis is used to finish reduction.

Consider our running example again. Recall that a traversal based reduction technique performs a tree traversal guided by the number of token descendants below each node to reduce this program. This means that a node with a larger number of tokens is visited before a node with a smaller number of tokens. Suppose that the traversal based reducer used for reducing this program is Pardis without loss of generality. Starting with the node with the largest number of token descendants (i.e., the root node ①), Pardis queries an oracle to see whether removal of ① succeeds or fails. It fails since removing ① yields an empty file. Pardis then tries to remove the node with the next largest number of tokens, ④. Removing this node is also unsuccessful since it removes the definition of function `foo` while its use is still within the program (call site on line 17). The reducer then proceeds with node ⑥ but cannot remove it since the property of interest (i.e., printing `Hello World!` on line 14) will not be preserved by removing ⑥. Removing node ⑪ will also cause the oracle to fail because it is required to increment variable i to avoid a run time issue on line 13. Removing node ⑤ will remove the `main` function, generating an invalid C program. Removing node ③ fails because of the *def-use* dependencies of struct `S`. Removing node ⑮ also fails due to the same reason for node ⑪. The next node considered by Pardis is ⑧ that is the declaration of `s1` and cannot get removed before its use. Next, node ⑫ is visited and PARDIS is able to remove it successfully. Removing nodes ⑦, ⑨, ⑩, ⑬ and ⑭ also fails due to either removing elements required by other elements in the program or removing the property of interest.

MODEL GUIDED PARDIS has a similar behavior to PARDIS when reducing this program. In particular, it traverses the tree in the same order as PARDIS. If a proper model is used, MODEL GUIDED PARDIS is however capable of avoiding oracle queries on some of the nodes with failed removal.

Now, let us see how a simple version of type batching might work on this example. Given a selected type, this version will merely traverse the parse tree and try to remove each node of that type individually. We develop a more nuanced version of the technique called PROBABILISTIC JOINT REDUCTION later in Section 5.2.3. For simplicity, we consider removing nodes of a given type one by one for now. Suppose that we select node types in the following order: expression statement (expr) $\rightarrow$ selection statement (selection) $\rightarrow$ declaration (decl) $\rightarrow$ compound statement (compound) $\rightarrow$ function definition (funcDef) $\rightarrow$ external declaration (exDecl) $\rightarrow$ translation unit, such that the type on the left is selected before the type on the right.

Instead of starting with the root node, the TYPE BATCHED REDUCER starts by selecting the first type, expression statement from the list of our type ordering and adds only nodes of that type to its search space in a decreasing order of the number of their token descendants, such that an expression statement node with a larger number of tokens as its descendants is visited before an expression statement node with fewer number of tokens. The first node to consider for removal is ⑮ since it has a grammar rule type that is the same as the selected type, expression statement, and it has the largest number of tokens (i.e., 10) among nodes with this type. Node ⑮ cannot get removed because it contains line 11 that is necessary to avoid a run time issue on line 13. The next node to consider is ⑫ and is successfully removed. Note that it took 9 oracle queries for the traversal based approach to remove ⑫ while TYPE BATCHED REDUCER could remove this node by performing only 2 queries. The

Table 5.8: Visiting order of nodes, number of total and successful oracle queries and number of removed tokens using traversal based PARDIS, TYPE BATCHED and TYPE BATCHED JOINT reduction techniques.

Reducer	**Order of removal trials (oracle queries) →**	**# total queries**	**# successful queries**	**# removed tokens**	**reduced output**
Traversal based PARDIS	(1, ×) (4, ×) (6, ×) (11, ×) (5, ×) (3, ×) (15, ×) (8, ×) **(12, ✓)** (7, ×) (13, ×) (10, ×) (9, ×) **(2, ✓)** (14, ×)	15	2	13	Listing 5.4
TYPE BATCHED	(15, ×) **(12, ✓)** (13, ×) (14, ×) **(11, ✓)** **(10, ✓)** **(9, ✓)** **(8, ✓)** (6, ×) (7, ×) (4, ×) (5, ×) **(3, ✓)** **(2, ✓)** (1, ×)	15	7	60	Listing 5.5
TYPE BATCHED JOINT	**({15, 12}, ✓)** (13, ×) (14, ×) **(11, ✓)** **({10, 9, 8}, ✓)** (6, ×) (7, ×) (4, ×) (5, ×) **({3, 2}, ✓)** (1, ×)	11	4	60	Listing 5.5

✓and × represent successful and unsuccessful removals, respectively.

next nodes with type expression statement are ⑬ and ⑭ but their removal is unsuccessful due to not preserving the property of interest. After all nodes of type expression statement are visited, the next type in our ordering, selection statement is selected. There is only one node of this type, node ⑪, and is successfully removed. The next type in our ordering is declaration with nodes ⑩, ⑨ and ⑧. By performing three oracle queries, the Type Batched Reducer can remove all the three nodes one by one. The next types to consider are compound statement and function definition. Nodes with type compound statement are ⑥ and ⑦ and cannot get removed because their removal makes the property of interest disappear. Removing nodes of type function definition, ④ and ⑤, also fails because of the same reasons explained above. The next type to select from our ordering is external declaration with nodes ③ and ② that successfully get removed. The last node to consider is the root node with an unsuccessful removal.

The first two rows of Table 5.8 contrast the traversal based Pardis with Type Batched Test Case Reduction approach for one round of tree reduction. Their resulting reduced outputs appear in Listing 5.4 and Listing 5.5, respectively. Observe that the traversal based reducer uses 15 oracle queries to remove only 13 tokens, leading to the majority of the tokens being tried again in subsequent fixed point rounds. In contrast, Type Batched Reducer removes 60 tokens using the same number of queries in a single tree reduction round, leading to a faster convergence towards the final reduced program.

```
int d = 10;
struct S {
  int f1;
  int f2;
};
void foo() {
  struct S s1 = {1, 2};
  int i = 0;
  bool increment = true;
  if (increment) {
      i += 2*i + i + 1;
  }
  s1.f1 = d/i;
  printf("Hello World!\n");
}
int main() {
  foo();
  return 0;
}
```

Listing 5.3: Original

```
 [redacted]
struct S {
  int f1;
  int f2;
};
void foo() {
  struct S s1 = {1, 2};
  int i = 0;
  bool increment = true;
  if (increment) {
      i += 2*i + i + 1;
  }
  [redacted]
  printf("Hello World!\n");
}
int main() {
  foo();
  return 0;
}
```

Listing 5.4: Traversal based

```
[redacted]
void foo() {
  [redacted]
  printf("Hello World!\n");
}
int main() {
  foo();
  return 0;
}
```

Listing 5.5: Type batched

Figure 5.8: Original program of Listing 5.2 and its reduced versions generated by traversal based Pardis and Type Batched Reducer after one round of reduction.

Additional Challenges and Advantages

Type batching shows strong potential benefits, but using it effectively requires addressing some additional challenges. We identify and address two key challenges in making type batching work: (1) *type scheduling*, and (2) *stopping criteria*. We also observe that type batching can significantly increase the likelihood of individual nodes being successfully removed, which enables further new techniques like Type Batched Probabilistic Joint Reduction.

Type scheduling refers to how we decide which types to run batches of at any point in time. It is the key mechanism that allows type batching to preserve dependencies. If, for instance, we chose to schedule declarations before statements, then attempting to remove declarations would fail if they are used by a statement. In contrast, removing statements first does not violate this dependency. The challenge lies in determining which orderings or schedules are better than others. In Section 5.2.2, we show that it is possible to train simple time-varying models of test case reduction that allow us to identify such orderings. Similar to the models of Model Guided Pardis in Section 5.1.1, these models can be trained using non-buggy code with a simulated property of interest and still yield appropriate orderings for improving reduction on real-world test data.

The stopping criteria determine when to switch from performing type batched operations to cleaning up the reduction with a traversal based technique. Not all types of nodes are as likely to be removable as others. At some point it may be more beneficial to stop considering any further type batches over such unsuccessful types when they are deemed to be expensive and unlikely to be beneficial. We explore this further also in Section 5.2.2.

Finally, we note that Pardis and Model Guided Pardis perform oracle queries to remove nodes *one by one*. We introduce Type Batched Probabilistic Joint Reduction that leverages the increased likelihood of successful removal for each node to attempt to remove *multiple* nodes at the same time where it is expected to be beneficial. In our example, a Probabilistic Joint Reduction technique enables a simultaneous removal of nodes {15, 12}, {10, 9, 8} and {3, 2} as shown in the last row of Table 5.8. Our Type Batched Probabilistic Joint Reduction builds upon the insights of recent works like Probabilistic Delta Debugging (PDD) [32], combining them with the benefits of type batching to accelerate reduction further for structured inputs such as programs. Unlike most approaches, including PDD, type batching enables the simultaneous removal of nodes across different levels and locations within the tree. We explore this reduction technique more in Section 5.2.3.

The next section elaborates on the core design of Type Batched Reducer while also presenting our solutions to the above mentioned challenges.

5.2.2 Type Batched Reducer: The Algorithm

The fundamental idea underlying Type Batched Test Case Reduction approach is that test case reduction can be faster when some types of nodes in a parse tree are considered before others. We partition the nodes into *batches* and perform reduction on one batch of nodes at a time. Because our goal is to improve the overall speed of reduction, we want to order the batches to create a list $\overline{batch}$ that maximizes the expected rate of reduction, ER, for a failing test case $\tau_{✗}$ as follows:

$$\mathrm{ER}(\overline{batch}, \tau_{✗}) = \mathbb{E}\left[\frac{\#\text{ tokens removed}}{\#\text{ oracle queries}}\right] \tag{5.1}$$

In other words, for a failing test case $\tau_{✗}$ to reduce, this rate of reduction expresses the number of tokens that we expect each oracle query to remove on average for a particular sequence of batches. If we can choose an ordering with a higher expected rate, then this can speed up the overall reduction process.

As stated earlier in the beginning of this section, *grammar rule types* or the labels applied by the parser to nodes of the parse tree can be used to partition nodes for a test case. Each type t captures both the structure of the test case as well as the aspects of the meaning, like the differences between declarations and expressions.

Algorithm 6: Type batched test case reduction [30].

```
Input: $\tau_{✗}$ – The test case to reduce as a parse tree
Input: $\psi : \mathbb{S} \to \mathbb{B}$ – Oracle for the property to preserve where $\mathbb{S}$ is the search space and
        $\psi(\tau_{✗}) = True$
Result: A minimum test case $\tau_{✗}' \subseteq \tau_{✗}$ s.t. $\psi(\tau_{✗}') = True$
1  $\tau_{✗}' \leftarrow \tau_{✗}$;
2  Function type_batched_reduction($\tau_{✗}'$, $\psi$):
3      types ← extract ($\tau_{✗}'$);
4      while types $\neq \varnothing$ do
5          best ← take_best(types, $\tau_{✗}'$);
6          types ← types - best;
7          if stop_early($\tau_{✗}'$, best) then
8              break
9          $\tau_{✗}' \leftarrow$ typed_traversal_reduction($\tau_{✗}'$, best, $\psi$)
10     $\tau_{✗}' \leftarrow$ full_traversal_reduction($\tau_{✗}'$, $\psi$) ;
11     return $\tau_{✗}'$
```

The core of our technique is shown in algorithm 6. We build upon this design throughout the remainder of this section. Line 3 starts by extracting the observed rule types from the test case being reduced. The loop starting on line 4 performs reduction on one batch of nodes at a time in the chosen order. Line 5 selects the best batch based on the current contents of the reduced test case so far and the remaining partitions. Lines 7-8 allow the process to stop early if that looks more fruitful than processing further partitions. Line 9 performs the reduction while only considering nodes in the chosen partition. For simplicity, we assume that reduction in the loop only removes nodes from the parse tree as opposed to both removing and replacing them. Finally, line 10 performs a clean-up pass using a normal traversal based test case reducer. This clean-up allows the technique to gain better efficiency by ordering the partitions well in the loop, while still having the full reduction power of existing reducers. In other words, Type Batched Reducer can be seen as a higher efficiency first phase to complement existing reducers.

Algorithm 6 highlights some of the challenges that must be addressed in order for the technique to work and ways that the technique can be extended or evolved. The most critical of these challenges is choosing which batches to reduce. We call this selection process *scheduling*. As we shall explore later, scheduling well can *boost* efficiency, but scheduling poorly can adversarially *harm* efficiency. For this reason, having effective and automated techniques is critical.

The next challenge arises because it can be advantageous to stop batch based reduction early. The conditions for stopping early are called the *stopping criteria*. It can be possible, for instance, that all easy to remove batches have been processed, and the only batches left to consider are for nodes that are very unlikely to be removed. In such cases, early transitioning to the clean-up phase is desirable.

Finally, even if we are able to find a good schedule, the efficiency is limited by the number of tokens that can be removed at once. We show how some of the effects of type batched reduction can overcome this burden with Type Batched Probabilistic Joint Reduction. We explore each of these challenges in the following.

Scheduling

The goal of scheduling is to produce an automated ordering for batches that maximizes $\mathrm{ER}(\overline{batch}, \tau_{\boldsymbol{\chi}})$ for the sequence of batches in the schedule. For illustration, consider a naive solution that uses some simplifying assumptions. Suppose that each individual type batch has a fixed rate $\mathrm{ER}_t(\tau_{\boldsymbol{\chi}}')$ known *a priori*. From our previous example, this is like assuming that declaration nodes are always removable (1) at the same probability of success and with (2) the same number of tokens beneath them. This assumption leads to a straightforward greedy approach to scheduling: sort the batches by descending $\mathrm{ER}_t(\tau_{\boldsymbol{\chi}}')$. This would allow algorithm 6 to reduce the most efficient batches earlier, leaving less remaining work for the following batches to perform. While this makes some impractical assumptions about $\mathrm{ER}_t(\tau_{\boldsymbol{\chi}}')$, this greedy approach lays the foundation of Type Batched Reducer. Specifically, we propose to use machine learning to automate a *time-varying* approximation of $\mathrm{ER}_t(\tau_{\boldsymbol{\chi}}')$ that we can use to guide scheduling.

However, as we shall see, this can be challenging for several reasons. First, it can be challenging to even compute $\mathrm{ER}_t(\tau_{\boldsymbol{\chi}}')$ for a *single* batch within a schedule. Oracle outcomes during reduction are not i.i.d. data, meaning that they are not independent and may change based on the outcomes of the earlier oracle calls. From the example in Figure 5.7, we expect $\mathrm{ER}_{\mathrm{declaration}}(\tau_{\text{✗}}')$ in a program to be *lower* if declarations are ordered *before* expression statements compared to when they are ordered *after* expression statements. Hence, $\mathrm{ER}_t(\tau_{\text{✗}}')$ of a single batch can vary both over time and with the batches that run before it. Second, training such a model of expectation requires training data, but we do not want to require data from, e.g., real-world bugs before building a model. Instead, we want to have the model and the technique trainable without real-world bugs so that it can immediately be used on the next bug found. This is similar to our training mechanism for Model Guided Pardis in Section 5.1.

Simple approximations for $\mathbf{ER}_t(\tau_{\text{✗}}')$

We start by discussing how to approximate $\mathrm{ER}_t(\tau_{\text{✗}}')$. Suppose for the moment that training data exist from previous test case reduction sessions using an existing traversal based reducer. In fact, our training data for Type Batched Reducer will be similar to the training data for Model Guided Pardis already presented in Section 5.1.1.

As the reduction traverses a sequence of nodes $n_1, n_2, n_3, \ldots$, traversing each node n_i records a tuple $(\tau_{\text{✗}}.\text{size}, n_i.\text{type}, \psi(c))$ for a smaller candidate test case c produced by removing n_i. The simplest strategy we might consider uses a naive model where the probability of successful reduction for a type is constant. For each node type t, we would compute P_t, the probability that a candidate is successful when visiting a node of the given type:

$$\begin{aligned}
&\text{Let } \mathit{of\_type}(c,t) = \exists n \text{ s.t. } c = \tau_{\text{✗}}' - n \wedge n.\text{type} = t \\
&\text{where } \tau_{\text{✗}}' - n \text{ is a subtree of } \tau_{\text{✗}}' \text{ rooted at node } n. \\
&P_t = P(\psi(c) = \mathit{True} \mid \mathit{of\_type}(c,t)) \\
&\quad\; = \frac{1}{|\{c \mid \mathit{of\_type}(c,t)\}|} \sum_{\{c \mid \mathit{of\_type}(c,t)\} \wedge \psi(c) = \mathit{True}\}} 1
\end{aligned} \tag{5.2}$$

That is, P_t is the number of *successful* candidates traversing a node of type t divided by the total number of candidates when traversing a node of type t. Given P_t for type t, we can compute $\mathbb{E}[\#\text{ tokens removed}]$ and $\mathbb{E}[\#\text{ queries}]$, which allow us to approximate $\mathrm{ER}_t(\tau_{\text{✗}}')$ for a particular input $\tau_{\text{✗}}$ using the *estimated rate* for a type, $\widehat{\mathrm{ER}}_t(\tau_{\text{✗}}')$:

$$\begin{aligned}
\mathrm{ER}_t(\tau_{\text{✗}}') &= \mathbb{E}\left[\frac{\#\text{ tokens removed}}{\#\text{ queries}}\right] \\
&\approx \frac{\mathbb{E}[\#\text{ tokens removed}]}{\mathbb{E}[\#\text{ queries}]} = \widehat{\mathrm{ER}}_t(\tau_{\text{✗}}')
\end{aligned} \tag{5.3}$$

Per Jensen's inequality, this only approximates the expected rate [57]. Given a reducer that tries to remove syntactically removable nodes while traversing them such as Perses or Pardis, these expectations can be computed by traversing the parse tree as shown below. For instance, $\mathit{tokens}(n,t)$ considers removing all the tokens below a node n with probability

P_t. When it is unsuccessful with probability $(1 - P_t)$, the expectations from the children of n are combined instead.

$$\begin{aligned}
&\mathbb{E}\left[\#\text{ tokens removed}\right] = \mathit{tokens}(\tau_{\text{✗}}.\text{root}, t) && (5.4)\\
&\mathit{tokens}(n, t) = P_t * 1^{\{n.\text{type}=t\}} * n.\text{tokens}\\
&\qquad\qquad + (1 - P_t * 1^{\{n.\text{type}=t\}}) \sum_{c \in n.\text{children}} \text{tokens}(c, t)\\
&\mathbb{E}\left[\#\text{ queries}\right] = \mathit{queries}(\tau_{\text{✗}}.\text{root}, t) && (5.5)\\
&\mathit{queries}(n, t) = 1^{\{n.\text{type}=t \wedge \text{is\_syntactically\_removable}(t)\}}\\
&\qquad\qquad + (1 - P_t * 1^{\{n.\text{type}=t\}}) \sum_{c \in n.\text{children}} \text{queries}(c, t)
\end{aligned}$$

where $1^{\{B\}}$ denotes 1 when B is true and 0 when B is false.

Thus, if we assume that P_t is constant per type t, we can use $\widehat{\text{ER}}_t(\tau_{\text{✗}}')$ for greedy scheduling. However, as previously mentioned, these probabilities are not i.i.d. in practice, so we may ask how much that affects the results and whether a richer model is required. Figure 5.9 explores this question for three types of nodes in our set of C programs.

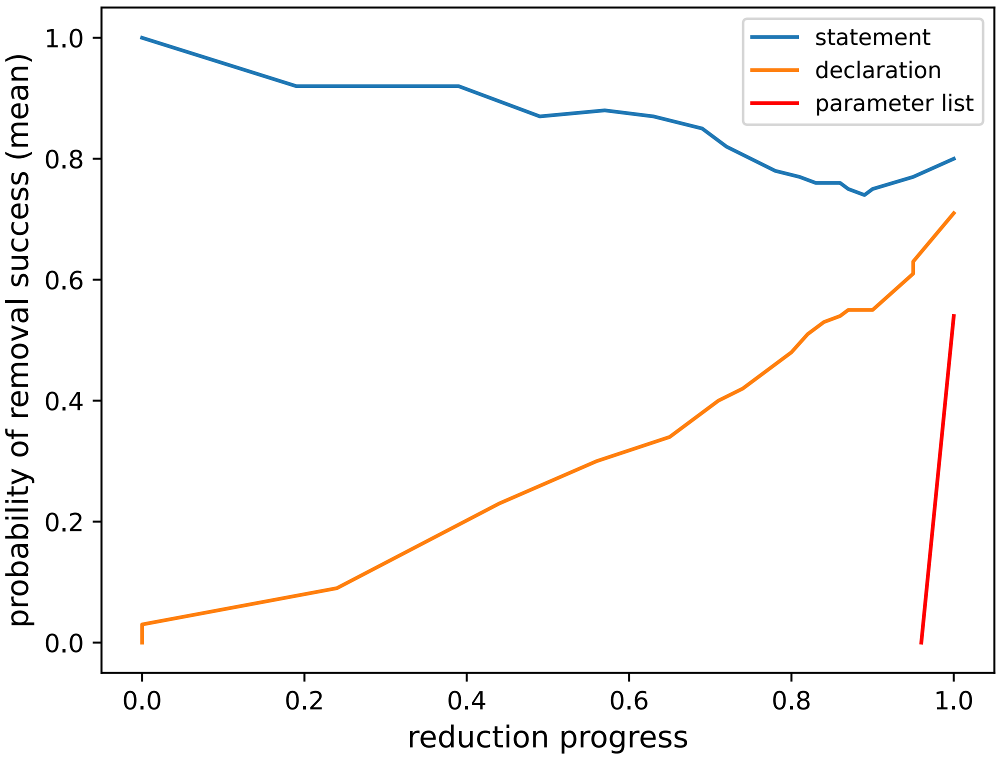


Figure 5.9: The probability of removal success for each type changes during reduction.

The Y axis captures the probability of successful removal, while the X axis represents reduction progress measured by the fraction of nodes in the original parse tree that have been removed in the current parse tree. Observe that the probabilities per type change significantly during reduction. For instance, declarations are unlikely to be removable at the beginning of the reduction process, but toward the end of the process they are quite likely to be removable. Statements, in contrast, are consistently likely to be removable. Some types, like parameter lists, do not even appear until close to the end of the reduction process because modern reducers such as Perses and PARDIS try to remove larger program chunks before smaller ones. We explore this further in our empirical results in the rest of the chapter, but this tells us that desirable models for P_t should at least account for the temporal effects of reduction.

To produce these time-varying models of P_t, we train a separate logistic regression model [58] for each rule type. The only feature of the model is the normalized reduction progress: $\frac{\tau_{\boldsymbol{\chi}}'.\text{size}}{\tau_{\boldsymbol{\chi}}.\text{size}}$. That is, the number of nodes in the parse tree of the partially reduced test case $\tau_{\boldsymbol{\chi}}'$, before a node is removed, out of the total number of nodes in the original test case $\tau_{\boldsymbol{\chi}}$. At the beginning of reduction, this has the value 1, and it approaches 0 as more of the tree is removed. The outcome labels are 1 for successful oracle queries and 0 for unsuccessful ones. With this, we estimate P_t by extracting the estimated probabilities from the model using the current test case size relative to the original. This is the version of P_t used in our computation of $\widehat{\text{ER}}_t(\tau_{\boldsymbol{\chi}}')$ in our greedy formulation of type scheduling in the following:

$$take\_best(types, \tau_{\boldsymbol{\chi}}') = argmax_{t \in types}\widehat{\text{ER}}_t(\tau_{\boldsymbol{\chi}}') \tag{5.6}$$

Note, this *does not* directly account for the impact of ordering between different types. Rather, it attempts to infer which orderings are more likely to be beneficial only independently considering the observed success rates as the combined reduction progresses. The idea is that the influence of one type on another is still captured in the individual success rates for each type alone. While we discuss further potential improvements in Section 6.3, we show in the evaluation of TYPE BATCHED REDUCER that this algorithm is able to infer effective orderings in practice.

Using synthetic training data.

We train the logistic regression models of our TYPE BATCHED REDUCER using the full sets of normally compiling source files used for training MODEL GUIDED PARDIS in Section 5.1. Similarly, we run *one node at at time removal* PARDIS on each file to minimize the source file while preserving a randomly selected token and compiling. The outcome of each oracle query is recorded along with the type of the node being visited and the current size of the test case. As a result, Type Batched Reducer preserves the generality of its models similar to Model Guided Pardis by not being restricted to bug specific information.

Stopping Criteria

The time-varying approach to $\widehat{\mathrm{ER}}_t(\tau_{\text{✗}}')$ allows us to choose the next type with the highest estimated rate at any moment. Eventually all types with high rates will have been chosen already, and all remaining rates will be slow. Recall from algorithm 6 that there is a final clean-up phase using a traversal based reducer, too. It is thus desirable to have some way of saying that all high-efficiency types have been chosen and that the algorithm should move on to the clean-up phase. The conditions for making this change are the *stopping criteria.*

Just as with take_best() in Equation 5.6, we use the estimated rates to make this decision. Instead of asking which type has the best $\widehat{\mathrm{ER}}_t(\tau_{\text{✗}}')$, we want to know whether $\widehat{\mathrm{ER}}_t(\tau_{\text{✗}}')$ for the best type is greater than the estimated rate from the clean-up phase itself. If the rate from the batch is higher, it makes sense to continue using the batch. If the rate from the clean-up phase is at least equal to the best next batch, it is preferable to exit to the clean-up phase. This is determined and made possible by using the Boolean function stop_early() defined in the following:

$$stop\_early(\tau_{\text{✗}}', best) = \left\{ \begin{array}{ll} True & \text{if } \widehat{\mathrm{ER}}_{\text{best}}(\tau_{\text{✗}}') \leq \widehat{\mathrm{ER}}_{\text{clean-up}}(\tau_{\text{✗}}') \\ False & \text{else} \end{array} \right\} \tag{5.7}$$

Note that computing the rate for the finishing phase is almost the same as for a single type. $tokens(n)$ and $queries(n)$ simply remove the type check on n ($1^{\{n.type=t\}}$) and use the probability $P_{n.\text{type}}$ for each individual node n in order to consider the probability of each different node type where appropriate.

5.2.3 Probabilistic Joint Reduction: The Algorithm

Type batching and training models based on the reduction progress can address the first two limitations of Model Guided Pardis that we described at beginning of the section: 1) Using type batching, a strict top down traversal order of the tree by Model Guided Pardis or any other traversal based reducer will be replaced with a traversal order that is capable of visiting nodes at different locations and levels within the tree, and 2) using the reduction progress to train models enables the dynamic reasoning of the reduction technique.

In addition, type batching increases the probability that any one node is removable on its own. This also means that the probability of removing multiple nodes at the same time (jointly) increases, giving another opportunity to improve the reduction rate by changing typed_traversal_reduction() in algorithm 6. This can be a further improvement over the limited single node removal operations in Pardis original version and Model Guided Pardis.

Algorithm 7: Probabilistic Joint Reduction over types [30].

```
Input: $t$ – The type of the parse tree node to reduce over
Input: $\tau_{✗}$ – The partially reduced test case so far
Input: $\psi : \mathbb{S} \to \mathbb{B}$ – Oracle for the property to preserve where $\mathbb{S}$ is the search space and
       $\psi(\tau_{✗}) = True$
Result: A test case $\tau_{✗}' \subseteq \tau_{✗}$ s.t. $\psi(\tau_{✗}') = True$
$\tau_{✗}' \leftarrow \tau_{✗}$;
Function probabilistic_joint_reduction($\tau_{✗}'$, $t$, $\psi$):
    initialize_probabilities($\tau_{✗}'$);
    $Q_{t,prio} \leftarrow$ collect_type_frontier($\tau_{✗}'.root$, $t$);
    while $Q_{t,prio} \neq \varnothing$ do
        chunk $\leftarrow$ pop_chunk($Q_{t,prio}$);
        if $\psi(\tau_{✗}'$ - chunk$)$ then
            $\tau_{✗}' \leftarrow \tau_{✗}'$ - chunk;
        else if let $\{n\}$ = chunk then
            frontier$_t \leftarrow$ collect_type_frontier($n$, $t$);
            $Q_{t,prio}$.insert(frontier$_t$);
        else
            update_probabilities(chunk);
            $Q_{t,prio}$.insert(chunk);
    return $\tau_{✗}'$;
Function collect_type_frontier(n, t)
    frontier $\leftarrow \varnothing$;
    for child of $n$ do
        if $child.type = t$ then
            frontier $\leftarrow$ frontier $\cup$ { child };
        else
            frontier $\leftarrow$ frontier $\cup$ collect_type_frontier(child, $t$);
    return frontier;
Function prio(n)
    return $n.probability$ * $n.tokens$;
Function pop_chunk(Q)
    chunk $\leftarrow \varnothing$ ;
    while $Q \neq \varnothing \wedge EG(chunk) < EG(chunk + Q.top())$ do
        chunk $\leftarrow$ chunk + $Q$.pop();
    return chunk
```

To propose an effective joint reduction mechanism, we leverage recent approaches from Probabilistic Delta Debugging (PDD) [32]. The original idea of PDD is that every failed trial when removing a subset from a list of elements lowers the *belief* about whether each element in the subset can be removed. If removing subset S from the list fails, for each element e_i present is S, PDD increases the probability of keeping (not removing) that element, $prob(e_i)$, as follows:

$$prob(e_i) = prob(e_i)/(1 - \prod_{e_j \in S} (1 - prob(e_j))) \tag{5.8}$$

The belief (or probability) of removing the element then becomes $1 - prob(e_i)$. The reduction process then leverages this belief in order to estimate the expected number of elements that can be removed (the *expected gain*, *EG*) and to prioritize which elements of the list to try removing next. This yields an $O(n)$ reduction process for a list, but unlike other recent O(n) reducers, including our ONE PASS DELTA DEBUGGING introduced in Chapter 3, it is guided by the belief in removability. Moreover, prioritizing elements based on the belief of removability updated by observed behavior of prior tests provides a joint reduction mechanism that is more efficient than other joint reduction techniques, including the original Delta Debugging algorithm [32].

This careful selection of how to remove multiple elements at the same time is precisely what we would like to leverage within a type batch. Note that our scenario is also somewhat different from PDD. Rather than considering removal of elements in a *list*, we are interested in removing all nodes of a given type from across the *entire parse tree*. The gain that we are most interested in is also not +1 for each node considered, but rather the gain is the actual number of tokens removed from the entire tree. These differences lead to a different problem formulation as presented in Algorithm 7.

The core function of the algorithm starts on line 2, providing a refined implementation for typed_traversal_reduction() in algorithm 6. As with other priority aware reducers discussed in this dissertation, it maintains a priority queue of nodes to consider, $Q_{t,\text{prio}}$. However, this queue maintains the invariant that it only holds nodes of the correct type for the current batch. Each iteration of the loop on line 5 is similar to a trial from PDD. It (1) builds a chunk of nodes from the queue based on the expected gain, (2) tries removing those nodes from the tree, (3) updates the tree, the belief in removability, and/or the priority queue based on the outcome of the trial for the chunk. The process repeats until no nodes of the given type are left to explore.

Note that the probability in algorithm 7 is different than the probability previously used to compute $\widehat{\text{ER}}_t(\tau_{\text{✗}}')$. The former represents the belief of removability for each *node* while the latter shows the probability of removal success for each *grammar rule type*.

The belief initialization (line 3) and belief updates (line 13) are the same as in PDD. However, it is worth mentioning that in contrast to PDD, we are running a chunk removal trial in a search space where syntactic validity is already preserved. As a result, our tests may fail (line 9 or line 12) because of only two reasons: Violating semantic constraints or removing the property of interest.

At the beginning phase of reducing elements without any observed behavior of tests, the probability of removing each element (the belief in removability of a node) is set to a high value (0.99 in PDD and this dissertation). The probability of removal for each element then starts to decrease after each failing trial of removing that element using Equation 5.8.

The main differences between original PDD and our Probabilistic Joint Reduction lie in our priority queue and how we measure expected gain. Note, unlike PDD, our priority queue holds nodes from across the entire parse tree rather than a single list. These nodes capture a $frontier_t$ of the tree similar to Model Guided Pardis in algorithm 5. They are the highest nodes of type t that have not been ruled out by the PDD reduction. When a node is ruled out by PDD, the t frontier inside the subtree of that node is added to the exploration queue (lines 10-11).

Our changes to expected gain affect both how the queue is prioritized as well as how we construct chunks. In PDD, the benefit of removing an element from a list was simply 1. The expected gain for removing a chunk was then $(\prod_{n \in \text{chunk}} b_n)|\text{chunk}|$ where b_n measures the belief that a node is removable. In our case, however, the gain from removing each node may differ based on the number of tokens in the subtree below a node. Thus, we have:

$$EG(\text{chunk}) = (\prod_{n \in \text{chunk}} b_n) * \sum_{n \in \text{chunk}} |\text{tokens(n)}| \tag{5.9}$$

That is, the expected gain is the belief that all nodes in a chunk are removable multiplied by the total number of tokens removed by a chunk. Nodes in the $Q_{t,\text{prio}}$ are ordered by expected gain, which guides chunk construction by this new metric in pop_chunk.

In the next section, we shall see that this provides a significant benefit, enabling the approach to remove more nodes at once with each oracle query inside a type batch.

5.2.4 Evaluation

In this section, we empirically assess the main strategies we take in our new approach. Moreover, we perform an overall evaluation of Type Batched Probabilistic Joint Reduction technique to compare it against Perses, Pardis and its variants.

To this end, we explore the following research questions:

- **RQ1.** How do different batch schedules impact the performance of reduction? In particular:
  - How do the schedules generated by our models compare against random schedules?
  - How do our time-varying schedules compare against time-invariant schedules?
- **RQ2.** How do type batching and probabilistic joint reduction interplay? In particular:
  - What is the impact of type batching on the performance of joint reduction?
  - What is the impact of joint reduction on the performance of type batched reduction?

- **RQ3.** How does Type Batched Probabilistic Joint Reduction compare to the state of the art reducers in terms of reduction time, number of oracle queries and final reduced size?

The experimental design with respect to benchmark, performance metrics and execution environment is based on the information available in Section 2.6 and is the same as other evaluation studies in this dissertation. Our training data sets consist of the full sets of programs collected in Section 5.1.1 for training Model Guided Pardis.

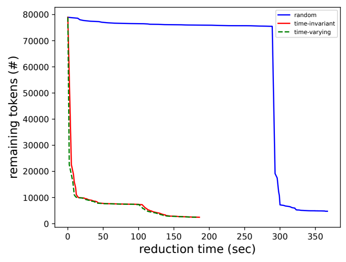
RQ1: The impact of different batch schedules on reduction

To understand the effectiveness of our model-generated schedules, we compare them against two groups of baseline schedules on our sample benchmark set. The first group, referred to as *random*, consists of schedules with the same types as our schedules but ordered in a random fashion. The second group called *time-invariant* consists of greedy schedules that use constant probabilities for each type of node in the parse tree. In the following, we explain how we construct each group in more detail.

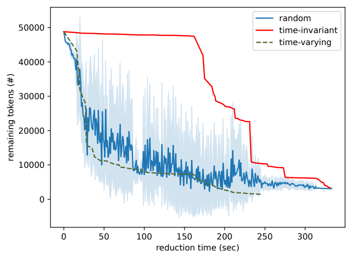
Random schedules. Comparing against random schedules of types allows us to see how our schedules perform relative to the space of possible schedules. Note that this space is still limited to the same types selected by our schedules but with different orderings. It does not contain schedules constructed by any possible syntactically removable type in the grammar. We believe that schedules created by arbitrary types from across the grammar are far from an appropriate baseline to compare our schedules against and limiting random schedules to the list of best types selected by our algorithm enables us to have a more accurate comparison.

To create random schedules for comparison, we take the same first types selected by Type Batched Reducer (up to five types for efficiency) and randomly sample 20 from their set of possible orderings (without replacement and excluding our schedule). If the number of possible orderings is less than 20 for a schedule and a test case, we select all the available orderings. We compare these orderings against our schedules that are also truncated to five types where applicable. Our full schedules are short in practice. On average, our schedules have five types for our C programs and consist of only two types for Rust and Go programs. As a result, it is unlikely to lose substantial scheduling data by truncating longer schedules to five types.

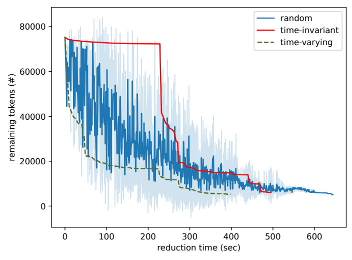
Time-invariant schedules. Time-invariant schedules use the greedy approach from algorithm 6, Equation 5.6 and Equation 5.7, but the probabilities for each node type are time-invariant, or constant. These probabilities are computed *a priori* as in Equation 5.2. Comparing against these schedules allows us to assess the impact of using the time-varying models of type probabilities P_t. It is informative for understanding the importance of accounting for how these probabilities change over time. For comparison, the number of types included in these schedules is not larger than that of random and time-varying schedules.



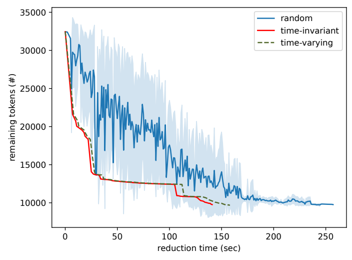
(a) clang-25900



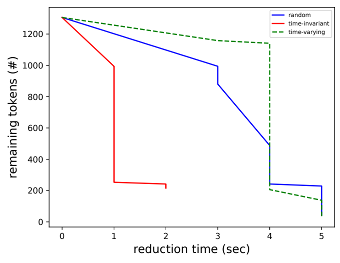
(b) clang-31259



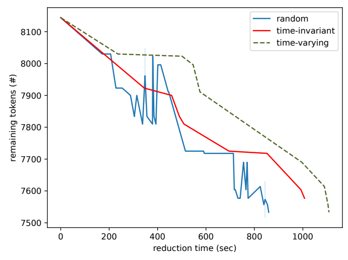
(c) gcc-60116



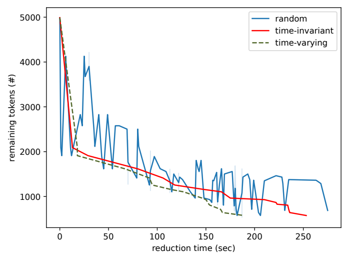
(d) gcc-61383



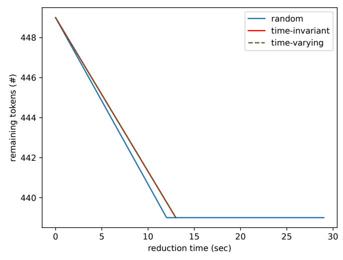
(e) gcc-77624



(f) rust-63791



(g) rust-77993



(h) go-30606

Figure 5.10: Performance of Type Batched Probabilistic Joint Reduction using random, time-invariant and time-varying schedules.

Results contrasting the different schedules are shown in Figure 5.10. We exclude one of our test cases, `rust-65934` from our set because its schedule has only one type and no nodes from that type are removed from the test case. The blue line depicts the average performance of random schedules, augmented by a shaded range for the standard deviation of random schedules. The green dashed line shows the performance of reduction using the time-varying schedules. The red line shows the performance of time-invariant schedules. The X axis is the reduction time in seconds, and the Y axis is the number of remaining tokens in the test case as the reduction proceeds.

As shown, for three test cases, `clang-31259`, `gcc-60116` and `rust-77993`, the time-varying schedules yield a faster convergence towards a reduced test case compared to both the average behavior of random schedules and the time-invariant schedules. The time-varying schedules for test cases `clang-25900`, `gcc-61383` and `go-30606` are better than random schedules and perform on par with time-invariant schedules. For the remaining two test cases, `gcc-77624` and `rust-63791`, either random or time-invariant schedule has the best performance. For `gcc-77624`, the shorter reduction time of the time-invariant schedule comes at a price though. This schedule has only one type while random and time-varying schedules consist of two types. As a result, the time-invariant schedule generates a larger output due to its less reduction power caused by having one fewer type. In Section 6.3, we further discuss the results of this study, especially to explain the observation that our time-varying schedules fall within the standard deviation range of random schedules for some cases. Overall, the results of this study suggest that using our time-varying models to generate schedules can provide a more effective framework to guide reduction.

To understand the behavior of time-invariant schedules, we examined the orders of their scheduled types and observed that a large number of declarations (e.g., function definitions) are always scheduled before their uses (e.g., function call sites). This can explain the plateau without reduction progress at the beginning of time-invariant schedules for `clang-31259` and `gcc-60116` in Figure 5.10. We find two reasons to explain why time-invariant schedules prioritize declarations over uses:

1. These declarations contain many tokens, so removing them can yield a large gain for the reducer.
2. The *overall* probability of removal success for these declarations is high. It means that although declarations are not removable at the beginning of reduction, most of them can successfully be removed as reduction proceeds and removes their uses. This increases their *overall* probability of removal, which is used in the time-invariant schedules.

In contrast, time-varying schedules account for changes in P_t. They prioritize uses over declarations by assigning a low probability to declarations at the beginning of reduction and a higher probability later as reduction proceeds. Uses and declarations are a specific example of one program element depending on another. Our time-varying scheduling tries to learn these relationships automatically and prioritize accordingly.

RQ2: Interplay between Type Batched Reducer and Probabilistic Joint Reduction

This research question studies the interplay between type batching and joint reduction by examining whether each of them can improve the other with respect to the performance of reduction. In particular, we answer the following questions:

- Does type batching improve the performance of joint reduction? To answer, we compare the performance of joint reduction *without* type schedules against joint reduction *with* type schedules.

- Does joint reduction improve the performance of type batching? To answer, we need to compare the performance of batched reduction using joint reduction vs. batched reduction using a traversal based reducer.

To measure the impact of type batching on joint reduction, we compare two different versions of algorithm 6. In the *type agnostic* version, we remove the schedule loop of line 4 and make the clean-up phase use joint reduction over all types simultaneously at line 10. In the *type batched* version, we use the algorithm as written with joint reduction inside the schedule loop at line 9. We use PARDIS at line 10 for the clean-up phase in this version.

To measure the impact of joint reduction on type batching, we also consider two variants of algorithm 6. In the *joint* reduction version, we again use joint reduction inside the schedule loop at line 9. In the *traversal based* reduction version, we replace line 9 with PARDIS. The clean-up phase at line 10 is again PARDIS for both versions.

Results are shown in Figure 5.11. By comparing the green and blue lines, we can see that type batching can improve the performance of joint reduction with respect to either time or size in 7 out of 9 test cases. Similarly, by comparing the green and purple lines, we can see that in 6 out of 9 cases, using joint reducer on nodes of the same type can improve the performance of type batched reduction compared to using PARDIS.

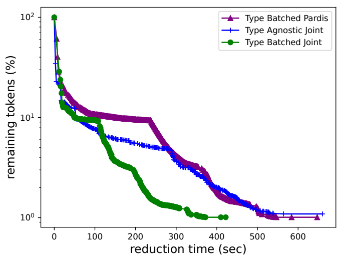
Results of these studies suggest that TYPE BATCHED REDUCER and PROBABILISTIC JOINT REDUCTION are synergistic and leveraging both of them together can help us to achieve a better overall performance of reduction.

To further investigate the impact of our joint reducer on type batching, we replace line 9 of algorithm 6 with Probabilistic Delta Debugging (PDD) [32]. As explained earlier in Section 5.2.3, we make use of insights from PDD to propose PROBABILISTIC JOINT REDUCTION that is suitable for structured inputs. To measure how our PROBABILISTIC JOINT REDUCTION compares against PDD, we define four different variants of joint reduction to replace line 9 of algorithm 6 with and evaluate the performance of the following reducers:

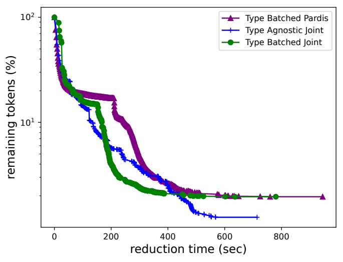
1. Type Batched PDD: This is the variant that is the closest to the original PDD proposed in the literature. Although type batched PDD reduces an entire parse tree rather than a list, it uses removal success probabilities of nodes to construct chunks. In addition, it considers the value gained by removing each node to be the same and equal to one. These are the same strategies taken by PDD.

2. Type Batched PDD^{+w}: Similar to type batched PDD, Type Batched PDD^{+w} leverages probabilities of nodes to construct chunks but instead of considering the gain of all nodes as one, it defines the number of token descendants or token weight of each node as its gain.

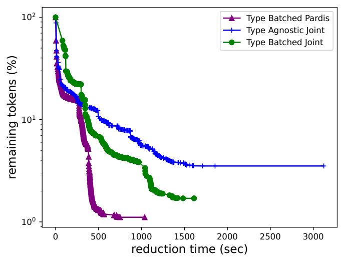
3. Type Batched Joint: This is our TYPE BATCHED PROBABILISTIC JOINT REDUCTION that uses expected gain of each node that is the product of its removal probability



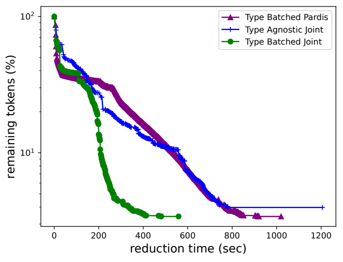
(a) clang-25900



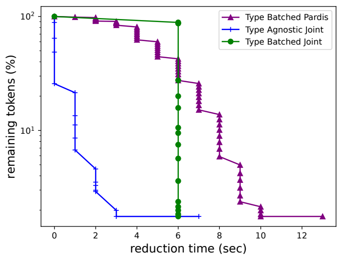
(b) clang-31259



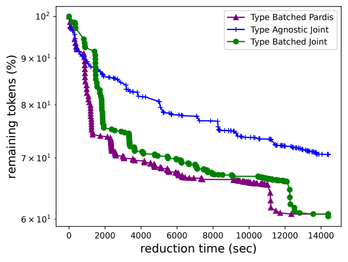
(c) gcc-60116



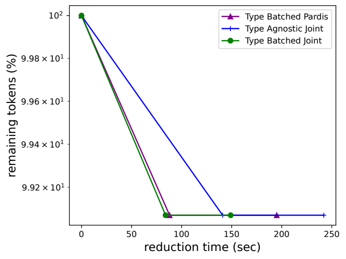
(d) gcc-61383



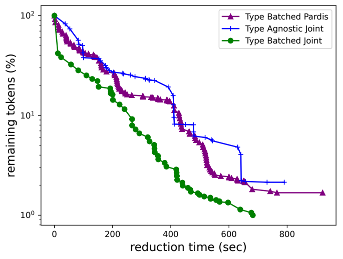
(e) gcc-77624



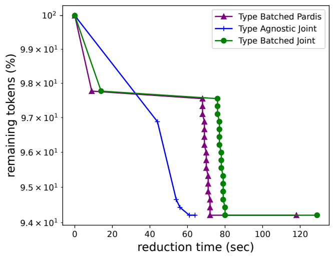
(f) rust-63791



(g) rust-65934



(h) rust-77993



(i) go-30606

Figure 5.11: Converging to a reduced test case in our sample benchmark using TYPE BATCHED JOINT, Type Agnostic Joint and Type Batched PARDIS.

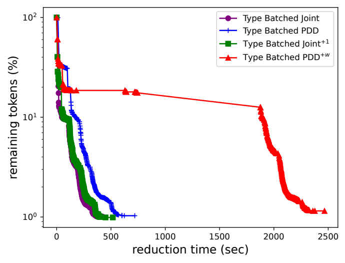
and token weight to construct chunks and considers the token weight of each node as its gain.

4. Type Batched Joint^{+1}: This variant constructs chunks of nodes by expected gain but considers the value of each node to be one.

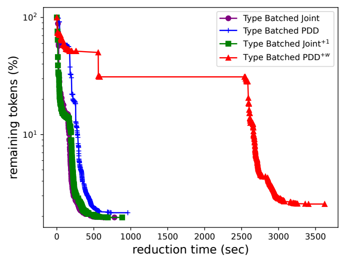
Table 5.9 summarizes the above variants based on their prioritization in chunk construction and gained value computation mechanisms. Results of reduction on our sample benchmark using these joint variants are shown in Figure 5.12 with X axis to be the reduction time in seconds and Y axis to be the log scaled percentage of remaining tokens. As can be seen, for the majority of the cases, either Joint or Joint^{+1} shows the best results among variants which highlights the significance of prioritizing elements by expected gain in structured domains.

Table 5.9: Different variants of joint reduction distinguished by their different prioritization mechanisms in chunk construction and computing gained values.

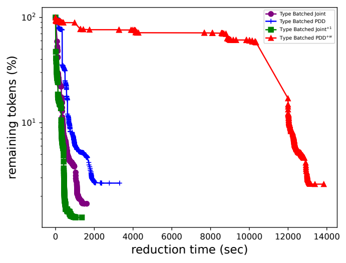
Variant	**Prioritized by**		**Gained value**	
	probability	expected gain	token weight	1
PDD	✓			✓
PDD^{+w}	✓		✓	
Joint		✓	✓	
Joint^{+1}		✓		✓



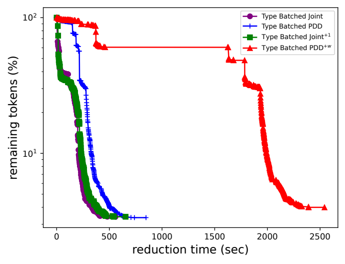
(a) clang-25900



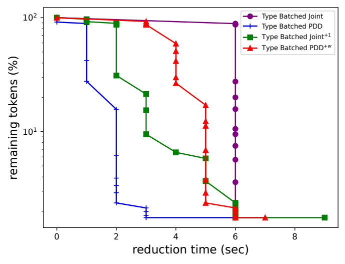
(b) clang-31259



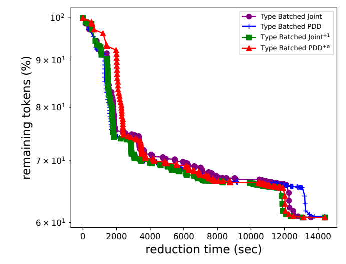
(c) gcc-60116



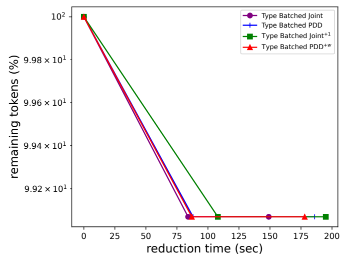
(d) gcc-61383



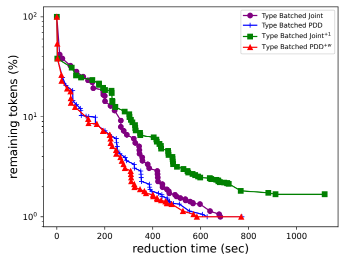
(e) gcc-77624



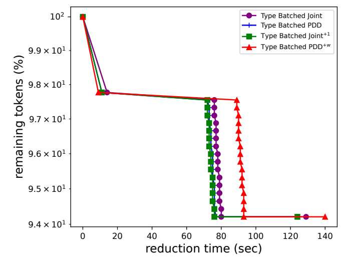
(f) rust-63791



(g) rust-65934



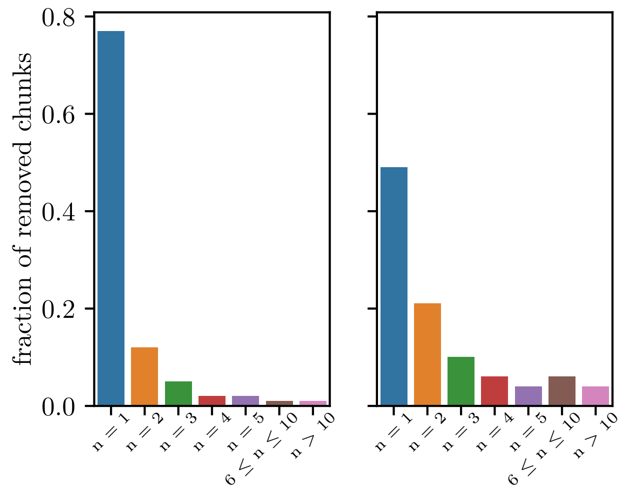
(h) rust-77993



(i) go-30606

Figure 5.12: Converging to a reduced test case in our sample benchmark using different variants of Joint reduction and PDD.

Size of removed chunks. Finally, to better understand why type batching improves the overall performance of joint reduction, we compute the distributions of sizes of chunks that



(a) type agnostic chunk size (b) type batched chunk size

Figure 5.13: Distribution of number of nodes (`n`) removed together using Type Batched Joint vs. type agnostic joint reduction.

were successfully removed using Type Batched Probabilistic Joint Reduction and its type agnostic version (using type schedules vs. not using them) for the C benchmarks.

Figure 5.13 compares the two distributions. As can be seen, the number of larger chunks that were successfully removed increases using type batching. In more detail, chunks of size one decrease from 77% in type agnostic approach to 49% of the total successful oracle queries when performing type batching while chunks of size two, three and four increase from 12%, 5% and 2% to 21%, 10% and 6% of the total successfully removed chunks. Chunks of size larger than 10 comprise 4% of successful oracle queries in the type batched approach while this number is 1% for the approach not performing type batching. In particular, we observe chunks of size up to 58 when using our type batched reducer which enables successful removal of 58 nodes together. Having larger chunks of nodes can explain the faster convergence of joint reduction when using type batching compared to not using it.

RQ3: Comparison with the state of the art techniques

Finally, we compare the overall performance of Type Batched Probabilistic Joint Reduction against Perses, Pardis and its variants on our full benchmark set. Again, we use Pardis for the clean-up phase of Type Batched Probabilistic Joint Reduction. Although any type of reducer can be used for this phase, we believe that using other reducers for clean-up is not likely to have a significant impact on the overall performance of Type Batched Probabilistic Joint Reduction. This can be supported by the empirical evidence shown in Figure 5.10 where most of the reduction occurs during type batching with few tokens left for the clean-up phase.

Results are shown in Table 5.10 and Table 5.11. Similar to other full evaluation studies in the dissertation, the first table shows results of reduction with removal only operations while the second table presents results of removal and replacement.

For C test cases, Type Batched Probabilistic Joint Reduction has the shortest average reduction time and the highest average efficiency in both tables. It improves the reduction time of Perses DD by 60% in Table 5.11 and has 41% faster reduction compared to the next best reducer for C programs in this table, Pardis Hybrid. For Rust and Go programs, Model Guided Pardis with $M_{rf.node}$ as its model has the best performance on average among all techniques.

Table 5.10: Comparing the performance of TYPE BATCHED PROBABILISTIC JOINT REDUCTION with different versions of Perses and PARDIS. Node replacements disabled.

Test Case	$O_{(\#)}$	Perses DD				Perses OPDD				Pardis				Pardis Hybrid				Model Guided Pardis				Type Batched Joint			
		$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$
clang-22382	21,069	1,144	5,114	3,220	6.19	1,144	4,674	3,165	6.30	867	1,977	3,856	5.24	867	2,373	2,567	7.87	1,113	1,196	3,720	5.36	1,090	2,129	1,760	11.35
clang-22704	184,445	746	4,104	1,645	111.67	746	3,728	1,423	129.09	326	4,055	4,243	43.39	326	1,886	1,367	134.69	389	4,142	5,234	35.17	326	4,330	4,344	42.38
clang-23309	38,648	2,321	9,173	2,882	12.60	2,321	8,659	2,667	13.62	2,129	2,897	998	36.59	2,129	3,543	1,045	34.95	2,372	1,316	565	64.21	389	941	232	164.91
clang-25900	78,961	798	4,205	1,393	56.11	798	3,777	1,217	64.23	883	1,582	577	135.32	883	1,948	550	141.96	1,056	1,053	496	157.07	797	1,654	422	185.22
clang-27747	173,841	612	3,743	1,613	107.40	612	3,411	1,374	126.08	498	1,271	656	264.24	498	1,470	624	277.79	595	901	596	290.68	751	1,446	520	332.87
clang-31259	48,800	920	4,189	1,605	29.83	920	3,805	1,524	31.42	608	1,474	783	61.55	608	1,644	751	64.17	699	729	388	123.97	957	2,117	780	61.34
gcc-59903	57,582	1,879	10,477	4,040	13.79	1,879	10,010	3,732	14.93	1,901	3,618	1,429	38.97	1,901	4,380	1,539	36.18	2,109	1,503	689	80.51	1,920	3,794	1,339	41.57
gcc-60116	75,225	1,281	9,066	4,706	15.71	1,281	8,392	4,496	16.45	1,361	3,718	1,907	38.73	1,331	4,631	1,964	37.62	1,558	1,722	751	98.09	1,281	3,717	1,612	45.87
gcc-61383	32,450	1,287	6,161	1,454	21.43	1,287	4,717	1,166	26.73	1,031	2,103	799	39.32	1,031	2,031	473	66.42	1,180	1,511	838	37.32	1,107	2,001	559	56.07
gcc-61452	26,733	1,023	5,548	3,563	7.22	1,023	5,165	3,515	7.31	1,023	1,685	2,444	10.52	1,023	2,084	1,887	13.62	1,356	1,177	2,449	10.36	1,164	2,040	1,624	15.74
gcc-61917	85,360	1,401	8,132	1,905	44.07	1,401	7,510	1,718	48.87	1,421	3,243	951	88.26	1,421	3,332	693	121.12	1,788	1,633	754	110.84	1,457	3,765	819	102.45
gcc-64990	148,932	1,120	6,350	1,932	76.51	1,120	5,859	1,735	85.19	1,127	2,768	944	156.57	1,127	3,385	879	168.15	1,377	1,241	708	208.41	941	2,503	706	209.62
gcc-65383	43,943	1,121	5,850	1,628	26.30	1,121	5,395	1,447	29.59	1,110	2,381	748	57.26	1,110	2,926	729	58.76	267	2,274	1,335	32.72	1,188	2,056	596	71.74
gcc-66186	47,482	1,299	4,889	1,244	37.12	1,299	4,539	1,129	40.91	1,288	1,877	544	84.92	1,288	2,231	537	86.02	1,432	1,078	442	104.19	1,283	2,210	592	78.04
gcc-71626	6,134	61	474	38	159.82	61	446	34	178.62	61	229	28	216.89	61	185	16	379.56	85	208	30	201.63	61	142	9	674.78
gcc-71632	141	82	190	62	0.95	82	190	60	0.98	82	114	55	1.07	82	121	56	1.05	107	29	18	1.89	82	121	57	1.04
gcc-77624	1,306	23	99	13	98.69	23	97	5	256.60	23	108	10	128.30	23	83	6	213.83	31	78	8	159.38	23	56	6	213.83
geomean	29,769	654	3,264	1,013	27.29	654	3,007	876	31.56	587	1,416	595	46.58	586	1,490	476	58.21	648	837	460	58.17	565	1,314	419	66.24
median	47,482	1,120	5,114	1,628	29.83	1,120	4,674	1,447	31.42	1,023	1,977	799	57.26	1,023	2,084	729	66.42	1,113	1,196	689	98.09	957	2,056	596	71.74
rust-44800	802	472	2,113	11,907	0.03	472	2,102	11,208	0.03	472	1,107	6,576	0.05	472	1,291	6,267	0.05	480	354	2,453	0.13	472	1,144	6,056	0.05
rust-63791	8,144	5,768	3,873	T/O	0.17	5,809	4,126	T/O	0.16	4,948	3,737	T/O	0.22	5,496	3,463	T/O	0.18	5,053	3,711	T/O	0.21	4,948	4,252	T/O	0.22
rust-65934	107	100	148	86	0.08	100	148	76	0.09	100	72	38	0.18	100	77	41	0.17	100	26	18	0.39	106	74	149	0.01
rust-69039	191	128	428	2,275	0.03	128	428	2,265	0.03	128	252	1,217	0.05	128	290	1,417	0.04	174	59	189	0.09	128	259	1,252	0.05
rust-77002	348	302	818	1,846	0.02	302	810	2,195	0.02	302	457	1,536	0.03	302	617	2,018	0.02	302	268	394	0.12	302	468	1,394	0.03
rust-77993	4,989	48	174	181	27.30	48	169	160	30.88	50	166	175	28.22	48	164	166	29.77	182	130	819	5.87	50	161	682	7.24
rust-78336	980	18	117	504	1.91	18	109	487	1.98	18	79	313	3.07	18	67	317	3.03	35	56	319	2.96	18	81	339	2.84
rust-78622	157	29	78	335	0.38	29	74	282	0.45	29	39	159	0.81	29	39	160	0.80	32	29	197	0.63	29	44	156	0.82
geomean	659	151	401	1,083	0.22	151	396	1,037	0.23	148	254	701	0.35	150	268	739	0.31	200	136	471	0.43	150	266	975	0.20
median	575	114	301	1,175	0.13	114	299	1,341	0.13	114	209	765	0.20	114	227	867	0.18	178	95	357	0.30	117	210	967	0.14
go-28390	146	84	212	33	1.88	84	211	30	2.07	84	127	18	3.44	84	134	18	3.44	145	57	10	0.10	84	135	19	3.26
go-29220	127	74	98	12	4.42	74	98	8	6.63	74	69	6	8.83	74	68	6	8.83	74	33	5	10.60	74	104	9	5.89
go-30606	449	423	629	155	0.17	423	629	149	0.17	423	421	100	0.26	423	429	101	0.26	423	259	70	0.37	423	522	129	0.20
geomean	203	138	236	39	1.12	138	235	33	1.33	138	155	22	1.99	138	158	22	1.99	166	79	15	0.73	138	194	28	1.57
median	146	84	212	33	1.88	84	211	30	2.07	84	127	18	3.44	84	134	18	3.44	145	57	10	0.37	84	135	19	3.26

O, R and Q denote number of tokens in the original program, reduced one and total number of oracle queries performed by the reduction technique, respectively. T is the reduction time in seconds and E is the efficiency in terms of number of tokens removed per second. Timeout (T/O) is set to 4 hours. $M_{rf.node}$ is the model selected to use for MODEL GUIDED PARDIS.

Table 5.11: Comparing the performance of Type Batched Probabilistic Joint Reduction with different versions of Perses and Pardis. Node replacements enabled.

Test Case	$O_{(\#)}$	**Perses DD**				**Perses OPDD**				**Pardis**				**Pardis Hybrid**				**Model Guided Pardis**				**Type Batched Joint**			
		$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$	$R_{(\#)}$	$Q_{(\#)}$	$T_{(s)}$	$E_{(\#/s)}$
clang-22382	21,069	334	5,030	3,535	5.87	334	4,656	3,474	5.97	343	2,847	4,273	4.85	343	3,081	3,076	6.74	369	2,394	4,248	4.87	331	1,882	2,261	9.17
clang-22704	184,445	266	4,289	1,788	103.01	266	3,950	1,489	123.69	173	4,506	3,296	55.91	173	3,265	1,680	109.69	177	4,150	3,184	57.87	233	4,755	3,542	52.01
clang-23309	38,648	237	6,747	2,229	17.23	237	6,365	2,095	18.33	60	4,741	2,109	18.30	874	7,092	2,630	14.36	649	5,350	2,308	16.46	154	780	241	159.73
clang-25900	78,961	304	4,578	1,660	47.38	304	4,205	1,493	52.68	307	2,383	987	79.69	307	2,508	919	85.59	311	2,090	945	83.23	315	1,889	568	138.46
clang-27747	173,841	220	4,311	1,624	106.91	220	3,974	1,495	116.13	230	2,059	1,069	162.41	220	2,467	1,125	154.33	231	1,950	1,250	138.89	401	1,730	631	274.87
clang-31259	48,800	374	5,312	2,424	19.98	374	4,920	2,161	22.41	376	2,847	1,495	32.39	376	2,964	1,369	35.37	374	2,375	1,239	39.08	362	1,623	698	69.40
gcc-59903	57,582	465	9,908	4,572	12.49	465	9,680	4,489	12.72	392	5,479	8,947	6.39	392	5,789	8,955	6.39	416	4,564	8,363	6.84	623	2,699	1,821	31.28
gcc-60116	75,225	480	7,514	9,088	8.22	480	7,129	8,905	8.39	528	5,397	3,107	24.04	532	5,835	3,143	23.76	543	4,584	2,802	26.65	608	4,032	1,830	40.77
gcc-61383	32,450	321	5,649	1,805	17.80	321	4,487	1,472	21.83	288	2,532	1,284	25.05	288	2,191	916	35.11	297	2,414	1,444	22.27	315	1,642	609	52.77
gcc-61452	26,733	370	5,552	3,993	6.60	370	5,238	3,876	6.80	342	2,539	2,884	9.15	342	2,696	2,224	11.87	340	2,113	2,957	8.93	360	1,867	2,067	12.76
gcc-61917	85,360	327	5,869	1,829	46.49	327	5,399	1,652	51.47	329	3,580	1,265	67.22	329	3,350	988	86.06	333	3,096	1,397	60.86	366	2,369	686	123.90
gcc-64990	148,932	354	5,873	2,288	64.94	354	5,489	2,112	70.35	354	2,696	1,248	119.05	354	2,851	1,161	127.97	359	2,917	1,910	77.79	332	1,834	687	216.30
gcc-65383	43,943	228	5,383	1,819	24.03	228	5,002	1,723	25.37	279	2,130	1,151	37.94	279	2,217	1,029	42.43	244	2,264	1,536	28.45	271	1,600	612	71.36
gcc-66186	47,482	441	4,692	1,563	30.10	441	4,392	1,403	33.53	391	2,085	914	51.52	391	2,219	863	54.57	398	2,235	1,300	36.22	394	1,720	621	75.83
gcc-71626	6,134	60	535	36	168.72	60	507	35	173.54	60	299	32	189.81	60	255	14	433.86	84	278	34	177.94	60	164	12	506.17
gcc-71632	141	75	217	57	1.16	75	217	57	1.16	75	173	54	1.22	75	176	52	1.27	88	198	41	1.29	75	134	49	1.35
gcc-77624	1,306	22	196	10	128.40	22	194	13	98.77	22	205	15	85.60	22	180	11	116.73	30	175	16	79.75	22	89	5	256.80
geomean	29,769	232	3,315	1,119	25.27	232	3,093	1,058	26.72	207	1,933	912	31.01	242	1,967	758	37.26	252	1,811	965	28.89	240	1,219	445	63.47
median	47,482	321	5,312	1,819	24.03	321	4,656	1,652	25.37	307	2,539	1,265	37.94	329	2,696	1,125	42.43	333	2,375	1,444	36.22	331	1,730	631	71.36
rust-44800	802	464	3,057	T/O	0.02	464	2,728	T/O	0.02	464	2,437	T/O	0.02	464	2,307	T/O	0.02	472	2,796	14,305	0.02	464	2,493	11,155	0.03
rust-63791	8,144	6,203	3,395	T/O	0.13	6,269	3,285	T/O	0.13	7,923	1,339	T/O	0.02	6,332	2,996	T/O	0.13	6,301	3,270	T/O	0.13	6,177	3,599	T/O	0.14
rust-65934	107	100	238	510	0.01	100	238	559	0.01	100	166	368	0.02	100	171	424	0.02	100	120	59	0.12	106	175	398	0.003
rust-69039	191	120	826	4,372	0.02	120	826	4,248	0.02	120	689	3,528	0.02	120	727	3,695	0.02	169	420	2,712	0.01	123	571	3,068	0.02
rust-77002	348	286	3,900	11,018	0.01	286	3,892	9,650	0.01	286	3,646	9,657	0.01	286	3,819	9,684	0.01	286	3,469	2,428	0.03	297	1,672	4,808	0.01
rust-77993	4,989	16	175	705	7.05	16	170	786	6.33	16	169	1,079	4.61	16	165	781	6.37	132	288	1,602	3.03	16	139	501	9.93
rust-78336	980	15	106	438	2.20	15	98	379	2.55	15	73	352	2.74	15	61	299	3.23	32	56	284	3.34	15	89	329	2.93
rust-78622	157	29	77	319	0.40	29	73	310	0.41	29	40	178	0.72	29	40	172	0.74	32	30	128	0.98	29	57	235	0.54
geomean	659	127	571	2,176	0.12	127	550	2,139	0.12	131	401	1,908	0.10	128	437	1,831	0.14	193	408	1,212	0.18	129	425	1,568	0.12
median	575	110	532	2,539	0.08	110	532	2,517	0.08	110	429	2,304	0.02	110	449	2,238	0.08	151	354	2,015	0.13	115	373	1,785	0.09
go-28390	146	84	245	35	1.77	84	244	35	1.77	84	163	24	2.58	84	170	23	2.70	140	107	19	0.32	84	231	32	1.94
go-29220	127	60	165	16	4.19	60	165	18	3.72	60	158	16	4.19	62	156	17	3.82	63	134	18	3.56	65	180	17	3.65
go-30606	449	233	912	204	1.06	233	912	203	1.06	213	794	179	1.32	213	802	187	1.26	213	700	161	1.47	233	788	189	1.14
geomean	203	106	333	49	1.99	106	332	50	1.91	102	273	41	2.43	104	277	42	2.35	123	216	38	1.19	108	320	47	2.01
median	146	84	245	35	1.77	84	244	35	1.77	84	163	24	2.58	84	170	23	2.70	140	134	19	1.47	84	231	32	1.94

O, R and Q denote number of tokens in the original program, reduced one and total number of oracle queries performed by the reduction technique, respectively. T is the reduction time in seconds and E is the efficiency in terms of number of tokens removed per second. Timeout (T/O) is set to 4 hours. $M_{rf.node}$ is the model selected to use for Model Guided Pardis.

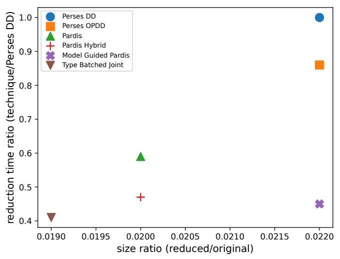
Similar to other chapters, we provide a speed up table by dividing the reduction time of Perses DD by the reduction time of Type Batched Probabilistic Joint Reduction. To make a thorough comparison, we also compute and provide the speed up metric for other reduction techniques proposed in this dissertation with respect to Perses DD. Table 5.12 summarizes the results for the speed up metric across all techniques. Based on these result, the two techniques Type Batched Reducer and Model Guided Pardis yield the highest speed up for test case reduction among all techniques.

Table 5.12: The speed up achieved by each test case reduction technique with respect to Perses DD.

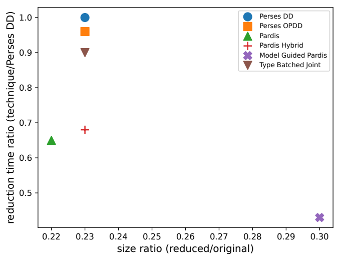
Test Case	**Replacements disabled**					**Replacements enabled**				
	Perses OPDD	Pardis	Pardis Hybrid	Model Guided Pardis	Type Batched Joint	Perses OPDD	Pardis	Pardis Hybrid	Model Guided Pardis	Type Batched Joint
clang-22382	1.02	0.84	1.25	0.87	1.83	1.02	0.83	1.15	0.83	1.56
clang-22704	1.16	0.39	1.20	0.31	0.38	1.20	0.54	1.06	0.56	0.50
clang-23309	1.08	2.89	2.76	5.10	12.42	1.06	1.06	0.85	0.97	9.25
clang-25900	1.14	2.41	2.53	2.81	3.30	1.11	1.68	1.81	1.76	2.92
clang-27747	1.17	2.46	2.58	2.71	3.10	1.09	1.52	1.44	1.30	2.57
clang-31259	1.05	2.05	2.14	4.14	2.06	1.12	1.62	1.77	1.96	3.47
gcc-59903	1.08	2.83	2.63	5.86	3.02	1.02	0.51	0.51	0.55	2.51
gcc-60116	1.05	2.47	2.40	6.27	2.92	1.02	2.93	2.89	3.24	4.97
gcc-61383	1.25	1.82	3.07	1.74	2.60	1.23	1.41	1.97	1.25	2.96
gcc-61452	1.01	1.46	1.89	1.45	2.19	1.03	1.38	1.80	1.35	1.93
gcc-61917	1.11	2.00	2.75	2.53	2.33	1.11	1.45	1.85	1.31	2.67
gcc-64900	1.11	2.05	2.20	2.73	2.74	1.08	1.83	1.97	1.20	3.33
gcc-65383	1.13	2.18	2.23	1.22	2.73	1.06	1.58	1.77	1.18	2.97
gcc-66186	1.10	2.29	2.32	2.81	2.10	1.11	1.71	1.81	1.20	2.52
gcc-71626	1.12	1.36	2.38	1.27	4.22	1.03	1.13	2.57	1.06	3.00
gcc-71632	1.03	1.13	1.11	3.44	1.09	1.00	1.06	1.10	1.39	1.16
gcc-77624	2.60	1.30	2.17	1.63	2.17	0.77	0.67	0.91	0.63	2.00
geomean	1.16	1.71	2.13	2.20	2.42	1.06	1.23	1.48	1.16	2.51
median	1.11	2.05	2.32	2.71	2.60	1.06	1.41	1.77	1.20	2.67
rust-44800	1.06	1.81	1.90	4.85	1.97	1.00	1.00	1.00	1.01	1.29
rust-63791	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
rust-65934	1.13	2.26	2.10	4.78	0.58	0.91	1.39	1.20	8.64	1.28
rust-69039	1.00	1.87	1.61	12.04	1.82	1.03	1.24	1.18	1.61	1.43
rust-77002	0.84	1.20	0.91	4.69	1.32	1.14	1.14	1.14	4.54	2.29
rust-77993	1.13	1.03	1.09	0.22	0.27	0.90	0.65	0.90	0.44	1.41
rust-78336	1.03	1.61	1.59	1.58	1.49	1.16	1.24	1.46	1.54	1.33
rust-78622	1.19	2.11	2.09	1.70	2.15	1.03	1.79	1.85	2.49	1.36
geomean	1.04	1.54	1.46	2.30	1.11	1.02	1.14	1.19	1.79	1.39
median	1.05	1.71	1.60	3.20	1.41	1.02	1.19	1.16	1.58	1.35
go-28390	1.10	1.83	1.83	3.30	1.74	1.00	1.46	1.52	1.84	1.09
go-29220	1.50	2.00	2.00	2.40	1.33	0.89	1.00	0.94	0.89	0.94
go-30606	1.04	1.55	1.53	2.21	1.20	1.00	1.14	1.09	1.27	1.08
geomean	1.20	1.78	1.78	2.60	1.41	0.96	1.19	1.16	1.28	1.03
median	1.10	1.83	1.83	2.40	1.33	1.00	1.14	1.09	1.27	1.08

The speed up metric allows for comparing reduction techniques based on their reduction time. To also illustrate the impact of reduced size on the performance of these techniques, we provide graphs in Figure 5.14 and Figure 5.15. For each domain, the X axis shows the average reduced size generated by each technique divided by the average original size of the test cases in the domain. The Y axis is the average reduction time of the technique divided by the average reduction time of Perses DD in that domain. Because X axis represents size and Y axis is reduction time, the technique on the bottom left of the graphs has the best performance.

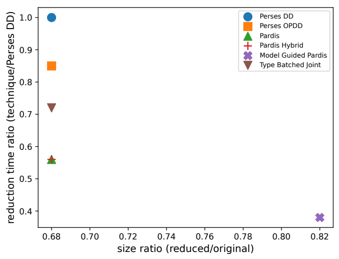
As a result, we can see that Type Batched Reducer with disabled node replacement is the best reducer for the C domain as shown in Figure 5.14 (a). Depending on the metric one is trying to improve, either Type Batched Reducer, Model Guided Pardis, Pardis or even Pardis Hybrid may be the best reduction option for the Rust and Go domains.



(a) C

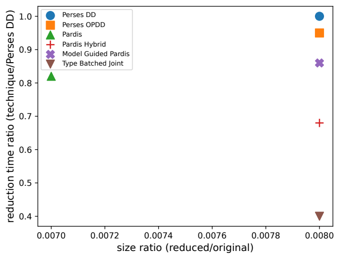


(b) Rust

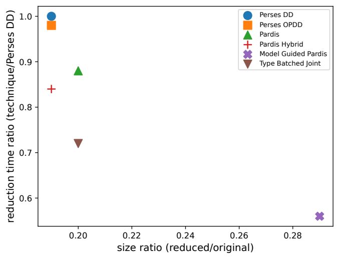


(c) Go

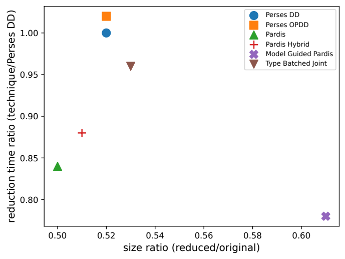
Figure 5.14: Reduced size and reduction time of each technique with respect to the original size and Perses DD's reduction time. Node replacements disabled. The technique on the bottom left of the graphs has the best performance.



(a) C



(b) Rust



(c) Go

Figure 5.15: Reduced size and reduction time of each technique with respect to the original size and Perses DD's reduction time. Node replacements enabled. The technique on the bottom left of the graphs has the best performance.

5.3 Summary

In this chapter, we trained and used machine learning models to further speed up test case reduction. In particular, we focused on addressing the problem of semantically invalid tests generated by domain agnostic reducers during reduction. These tests with no real impact on the reduced size could significantly waste the reduction time and effort.

To mitigate the adverse effect of these tests, we proposed Model Guided Pardis that improves the performance of Pardis by predicting and skipping tests on candidates that are likely to be semantically invalid. We leveraged multiple sets of features to train various models. Our results highlight the importance of grammar rule types in developing effective and efficient models that are capable of predicting semantic invalidity. In particular, Model Guided Pardis with $M_{rf.node}$ model trained by the grammar rule type of the node considered for removal, resulted in an average speed up of 1.29x, 1.49x, and 1.46x for C, Rust, and Go domains, respectively compared to Pardis. This speed up led to a slight increase in the average reduced size. While Pardis generated outputs with 587, 148, and 138 tokens on average for C, Rust, and Go domains, respectively, the reduced outputs produced by Model Guided Pardis had 648, 200, and 166 tokens on average.

Additionally, we proposed Type Batched Probabilistic Joint Reduction that leverages grammar rule types to prioritize and suggest portions of the test case that are the most advantageous to reduce. This suggestion is made through a schedule of types that is built dynamically during the reduction process. Moreover, we demonstrated that using type schedules can increase the probability of successful simultaneous removal of several nodes to further accelerate reduction.

Compared to the state of the art Perses, Type Batched Probabilistic Joint Reduction led to an average speed up of 2.42x, 1.11x, and 1.41x when reducing C, Rust, and Go programs, respectively. The outputs generated by Type Batched Probabilistic Joint Reduction were either smaller or equal in size to those produced by Perses with 565, 150, and 138 tokens on average for C, Rust, and Go, respectively. The average reduced size by Perses was 654, 151, and 138 tokens for these domains.

To summarize, both Model Guided Pardis and Type Batched Probabilistic Joint Reduction address the problem of semantic invalidity during reduction to increase efficiency. While the former achieves this by learning to avoid semantically invalid tests, the latter learns to guide reduction towards performing tests that are likely to be semantically valid. Additionally, by building chunks of nodes based on updating the belief of removability of each node with regard to both semantic validity and the property of interest, Type Batched Probabilistic Joint Reduction is capable of simultaneously removing multiple portions of the test case.

Chapter 6

Discussion

So far in the dissertation, we proposed and evaluated domain agnostic techniques to accelerate test case reduction for inputs from multiple domains. In this section, we examine how our techniques compare against C-Reduce [13], the most well-known domain specific program reducer for C. We also discuss the threats to validity of our techniques and the potential improvements as directions for further exploring domain agnostic test case reduction in the future.

6.1 C-Reduce: A Domain Specific C Reducer

As described earlier in Section 2.4.2, C-Reduce is a powerful tool for effectively reducing C programs. Its search space is hand tailored to adhere to the validity constraints of the domain. Due to a larger and more sophisticated set of reduction transformations, C-Reduce tends to generate outputs that are smaller than the domain agnostic reducers with generalized search spaces.

To better understand how our proposed techniques stand in relation to C-Reduce, we conduct the following two studies:

1. Running C-Reduce on the *original* C programs in our benchmark to measure the reduced size, reduction time, the number of queries and the efficiency of this domain specific reducer.

2. Running C-Reduce on the smaller versions of the C programs in our benchmark that are *already reduced* by one of our proposed domain agnostic reducers. In other words, we make use of C-Reduce as a post processing step for enabling potential further reduction on the outputs generated by our reducer.

Results of the first study are presented in Table 6.1. For direct comparison, reduction results of TYPE BATCHED PROBABILISTIC JOINT REDUCTION are also shown in the table. We select our TYPE BATCHED REDUCER to compare it against C-Reduce since it has the smallest number of oracle queries, the shortest reduction time and the highest efficiency

Table 6.1: A comparison between the performance of domain specific C-Reduce and domain agnostic Type Batched Probabilistic Joint Reduction on the set of C programs. Node replacements are enabled in Type Batched Reducer.

Test Case	$O(\#)$	**C-Reduce**				**Type Batched Joint**			
		$R(\#)$	$Q(\#)$	$T(s)$	$E(\#/s)$	$R(\#)$	$Q(\#)$	$T(s)$	$E(\#/s)$
clang-22382	21,069	70	16,104	4,659	4.51	331	1,882	2,261	9.17
clang-22704	184,445	51	16,953	3,656	50.44	233	4,755	3,542	52.01
clang-23309	38,648	43	30,569	3,348	11.53	154	780	241	159.73
clang-25900	78,961	101	21,982	2,354	33.50	315	1,889	568	138.46
clang-27747	173,841	69	15,038	1,638	106.09	401	1,730	631	274.87
clang-31259	48,800	127	25,741	3,034	16.04	362	1,623	698	69.40
gcc-59903	57,582	102	47,632	5,156	11.15	623	2,699	1,821	31.28
gcc-60116	75,225	84	33,960	4,328	17.36	608	4,032	1,830	40.77
gcc-61383	32,450	71	17,574	2,154	15.03	315	1,642	609	52.77
gcc-61452	26,733	130	32,825	13,293	2.00	360	1,867	2,067	12.76
gcc-61917	85,360	80	31,999	4,431	19.25	366	2,369	686	123.90
gcc-64990	148,932	110	31,258	6,270	23.74	332	1,834	687	216.30
gcc-65383	43,943	61	18,193	2,617	16.77	271	1,600	612	71.36
gcc-66186	47,482	142	27,907	4,358	10.86	394	1,720	621	75.83
gcc-71626	6,134	44	3,831	226	26.95	60	164	12	506.17
gcc-71632	141	73	8,686	953	0.07	75	134	49	1.35
gcc-77624	1,306	15	951	79	16.34	22	89	5	256.80
geomean	29,769	72	17,254	2,367	12.00	240	1,219	445	63.47
median	47,482	73	21,982	3,348	16.34	331	1,730	631	71.36

O, R and Q denote number of tokens in the original program, reduced one and total number of oracle queries performed by the reduction technique, respectively. T is the reduction time in seconds and E is the efficiency in terms of number of removed tokens per second.

among our domain agnostic reducers on average for the C domain. As can be seen, Type Batched Reducer is able to reduce these programs in 445 seconds on average compared to C-Reduce with an average reduction time of 2,367 seconds. In addition, Type Batched Reducer performs only 1,219 oracle queries while the average number of tests performed by C-Reduce is 17,254. The number of tokens removed per second is more than 63 by Type Batched Reducer compared to 12 tokens by C-Reduce. In other words, reduction time and number of oracle queries improve by 81% and 93%, respectively and the efficiency becomes more than 5 times larger using the Type Batched Reducer. In contrast, with respect to size, Type Batched Reducer generates outputs of size 240 tokens on average compared to 72 tokens by C-Reduce which means that C-Reduce has 70% more reduction power.

The above results are expected because as described earlier in this chapter, the more sophisticated search space of C-Reduce increases both the power and time of reduction. If a slight increase in size is negligible, Type Batched Reducer will be a more suitable choice because it offers a framework for faster reduction. C-Reduce, on the other hand, excels in situations where a smaller size is strictly required.

A question that arises here is that can we combine the benefits of both domain specific and domain agnostic reducers? Can we set up a framework to yield the higher reduction power of C-Reduce and the shorter reduction time of Type Batched Reducer together? The second study stated above can answer this question.

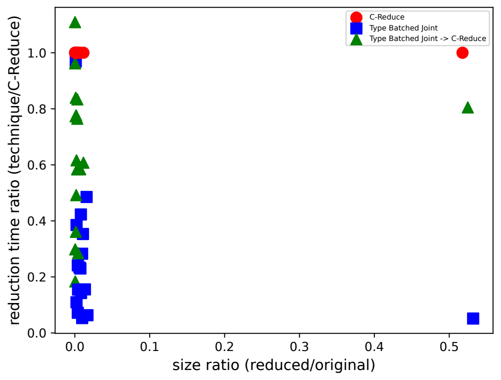
Table 6.2: The performance of reduction when using domain agnostic Type Batched Probabilistic Joint Reduction as a preprocessing step before C-Reduce.

Test Case	$O(\#)$	**C-Reduce**				**Type Batched Joint → C-Reduce**				**Speed Up** $(\times)$
		$R(\#)$	$Q(\#)$	$T(s)$	$E(\#/s)$	$R(\#)$	$Q(\#)$	$T(s)$	$E(\#/s)$	
clang-22382	21,069	70	16,104	4,659	4.51	70	11,309	3,562	5.90	1.31
clang-22704	184,445	51	16,953	3,656	50.44	50	11,304	4,055	45.47	0.90
clang-23309	38,648	43	30,569	3,348	11.53	22	5,732	615	62.81	5.44
clang-25900	78,961	101	21,982	2,354	33.50	83	15,869	1,820	43.34	1.29
clang-27747	173,841	69	15,038	1,638	106.09	71	12,513	1,576	110.26	1.04
clang-31259	48,800	127	25,741	3,034	16.04	166	23,354	2,529	19.23	1.20
gcc-59903	57,582	102	47,632	5,156	11.15	102	32,428	4,004	14.36	1.29
gcc-60116	75,225	84	33,960	4,328	17.36	93	22,665	3,633	20.68	1.19
gcc-61383	32,450	71	17,574	2,154	15.03	73	11,645	1,327	24.40	1.62
gcc-61452	26,733	130	32,825	13,293	2.00	117	17,584	3,782	7.04	3.51
gcc-61917	85,360	80	31,999	4,431	19.25	118	14,243	1,599	53.31	2.77
gcc-64990	148,932	110	31,258	6,270	23.74	75	14,501	1,870	79.60	3.35
gcc-65383	43,943	61	18,193	2,617	16.77	81	10,144	1,287	34.08	2.03
gcc-66186	47,482	142	27,907	4,358	10.86	142	19,011	2,544	18.61	1.71
gcc-71626	6,134	44	3,831	226	26.95	44	2,721	132	46.14	1.71
gcc-71632	141	73	8,686	953	0.07	74	2,385	767	0.09	1.24
gcc-77624	1,306	15	951	79	16.34	15	983	48	26.90	1.65
geomean	29,769	72	17,254	2,367	12.00	71	10,155	1,375	20.71	1.72
median	47,482	73	21,982	3,348	16.34	75	12,513	1,820	26.90	1.62

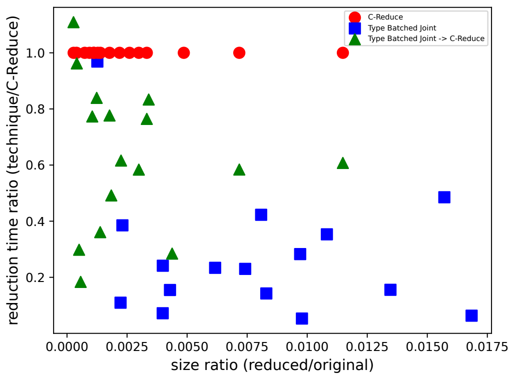
O, R and Q denote number of tokens in the original program, reduced one and total number of oracle queries performed by the reduction technique, respectively. T is the reduction time in seconds and E is the efficiency in terms of number of removed tokens per second.

To this end, we run Type Batched Reducer as a preprocessing step of C-Reduce to perform reduction as fast as possible using our domain agnostic reducer, and then run C-Reduce on the output files of Type Batched Reducer to perform a more rigorous reduction. Results of this combined study are shown in Table 6.2. Interestingly, using Type Batched Reducer before C-Reduce generates outputs that are the same or even smaller than those generated by C-Reduce in 10 out of 17 cases. In other cases, outputs generated by Type Batched Reducer → C-Reduce are only a few tokens larger. On average, running Type Batched Reducer before C-Reduce generates outputs of size 71 tokens in 1,375 seconds while C-Reduce generates outputs of size 72 tokens in 2,367 seconds on average. The combined technique increases the speed of C-Reduce by 1.72 on average while maintaining its reduction power. The results of this study suggest that Type Batched Reducer can be used complementary to C-Reduce to accelerate test case reduction even in cases where a smaller size is strictly required.

To further support the above claim, Figure 6.1 (a) depicts a comparison between C-Reduce and Type Batched Probabilistic Joint Reduction both as standalone and complementary techniques. The X axis is computed by dividing the reduced size generated by each technique by the original size for each test case. Y axis shows the reduction time of each technique divided by C-Reduce's reduction time. As a result, this value is always one for C-Reduce when it is used individually. Again because X axis represents size and Y axis is reduction time, the technique on the bottom left of the graphs has the best performance. For



(a) Full set of C programs



(b) Set of C programs excluding `gcc-71632` (outlier)

Figure 6.1: Reduced size and reduction time of C-Reduce, Type Batched Probabilistic Joint Reduction and their combined version with respect to the original size and C-Reduce's reduction time. The technique with more points on the bottom left of the graphs has a better performance.

better illustration, Figure 6.1 (b) shows the same graph with a focus on the majority of the data by excluding `gcc-71632` with the outlier points. As can be seen, both Type Batched Joint and Type Batched Joint → C-Reduce have shorter reduction time compared to C-Reduce, while Type Batched Joint → C-Reduce generates outputs similar in size to those produced by C-Reduce.

6.2 Threats to Validity

In this section, we discuss some of the threats to validity of the results presented in the dissertation.

6.2.1 Generalizability

To prevent bias as much as possible on our part, we used the benchmark provided by Perses and available on their Github repository [43]. This benchmark consists of multiple primarily large C, Rust and Go programs that reveal bugs in real-world compilers. We were able to replicate failures for the majority of these programs. To further generalize the benchmark, we augmented it by an XML file to demonstrate the capability of our techniques in reducing non-program inputs as well.

We compare our techniques with the state of the art on the full set of benchmark discussed above. For explanatory studies, we choose a random sample and use it consistently across the dissertation. We embed at least one program from each domain into this sample benchmark.

6.2.2 Non-deterministic Behavior

The majority of our test cases reveal bugs in compilers. As a result, the oracle scripts in our studies contain calls to failing compilers and other potential applications such as address sanitizers and undefined behavior checkers. Although compilers are deterministic in general, address space layout randomization (ASLR) may cause a non-deterministic behavior of a compiler by randomizing loading memory locations of executables. To mitigate this problem, in addition to performing multiple runs for each test case and each reduction technique, we disabled ASLR during our studies.

6.2.3 Oracle Verification Time

We previously discussed in Section 4.5.3 that the verification time of every single oracle run can directly impact the total reduction time of the technique. In particular, we demonstrated that a technique with a larger number of oracle calls may have a shorter overall reduction time if its oracle calls are less expensive. One of the factors that affects the oracle verification time is the input size. For instance, compared to Pardis, Pardis Hybrid removes multiple nodes together and generates typically smaller remaining configurations that need to be tested against an oracle. Smaller inputs tend to be compiled faster. As a result, there were cases in Table 4.1 where the reduction time of Pardis Hybrid was shorter but the number of tests were larger than those performed by Pardis.

However, it is also possible that multiple runs of an oracle on the same input differ in verification time as well. This is again due to the compilation phase that may differ in timing when compiling the same input multiple times. Thus, when comparing two techniques, it is possible that they both run the same oracle on the same input at some point during their reduction and obtain different results for verification time. This problem may be more noticeable when compiling the Rust test cases as the compilation in Rust requires multiple dependencies and usually takes longer than compilation in C. To mitigate this problem, we run several rounds of reduction for each test case and technique on the same execution environment to ensure that the results do not diverge significantly.

6.2.4 1-minimality

As described earlier in Chapter 2, finding the absolute global minimum of a test case with a property of interest is not feasible. Hence, each reducer has its own search space and may produce a different reduced output. This may cause some unclarity when comparing techniques. For example, it is possible that a technique that is theoretically worse than other techniques performs well on a specific test case in practice. The structure of the specific test case and the search space of the reducer can for instance allow for removing a statement with multiple uses early in the reduction phase and enable the removal of those uses' declarations once they are visited.

To mitigate this problem, in addition to the results of reduction on each individual test case, we provide the average behavior of each technique through geomean and median values for each domain.

6.3 Future Work

In this section, we discuss some potential future research directions that can further accelerate test case reduction and yield speeds that can minimize developer interruption. In particular, we focus on integrating machine learning into the reduction process.

6.3.1 Input Features

We used a specific set of features to train our models. In particular, we found that grammar rule types provide valuable information for the learning process. While effective, the features selected in this dissertation are not the only properties to leverage in order to reason about the validity of the tests. We may use more complex features in future to further improve precision of models. For example, we can define features that more precisely capture dependencies among elements in the test case. A path from a declaration to a use or some dependency pairs representing *def/use* elements may be appropriate choices to this end. However, a possible challenge is increased noise in the training data that may adversely affect the performance of the models.

6.3.2 Reduction Operations

We currently train our models with respect to the removal operations. Model Guided Pardis learns when to perform a removal trial on a node and when not to. Type Batched Reducer also learns how to select batches of nodes that are likely to have a higher expected rate of removal. As a future direction, it is possible to learn other reduction operations such as node replacement. Model Guided Pardis can be extended to learn when to perform a node replacement or what candidate node to choose for replacement in order to increase the likelihood of successful reduction. Type Batched Reducer can also be extended to approximate the expected rate of both removal and replacement.

6.3.3 Training Data and Algorithms

We used random forests and logistic regression to train our models. Our goal in this dissertation was not to propose models with the best absolute performance. Rather, we aimed to derive models that are easy to understand and that can convincingly show that the effect of machine learning on test case reduction is real. In fact, this dissertation opens the door to several future improvements for using machine learning to directly guide test case reduction. For instance, neural embeddings of the nodes [59] could be used to help guide the batch partitioning process. Moreover, longer term reduction reward computed by reinforcement learning techniques [60] could be used to improve the overall reduction process. For instance, our Type Batched Reducer currently focuses on greedy selection of batches, but it is possible that suboptimal greedy batches actually improve the reduction rates for other partitions afterward. Models of reduction that can account for ordering beyond time and impact on longer rates provide an avenue here. We expect this to improve our time-varying schedules that still fall in some regions of the standard deviation of random schedules in Figure 5.10.

Finally, the training data directly impact the behavior of the models. Thus, we made our best effort to collect representative data by using large fuzzer-generated programs and trending Github projects. However, different training data sets can be used to investigate their impact on the reduction process. In one of our prior works [29], we leveraged small test cases from the GCC torture test suite [61] to train models for Model Guided Pardis on the C domain that has the largest test cases. Those models had a similar behavior compared to the models trained by large fuzzer-generated programs in this dissertation. Future research on using different training data sets on various domains is possible.

6.3.4 Human Studies

Similar to prior works on test case reduction [1, 14, 15, 13, 36, 16, 31, 19], we evaluate the performance of our techniques in terms of the metrics mentioned in Section 2.6.3. The intuition is that a smaller test case obtained within a shorter reduction time indicates a higher performance of the reduction technique. Although this statement seems reasonable, it leaves out an important determining factor in evaluating a software technique: How can a technique help humans in practice? Although an interesting question, finding an answer to it has been out of the scope of our dissertation and can be pursued as an interesting direction for future work.

Chapter 7

Related Work

We describe research work related to this dissertation from multiple perspectives. First, We discuss research on test case reduction, its techniques and applications in general. Then, we focus on program reduction as a specific kind of test case reduction and present the most recent state of the art program reducers. In addition, we describe some works that introduce other applications and uses of program reduction. We explain how the methods suggested in this dissertation differ from the ones already in use for test case reduction. Finally, we review works that make use of machine learning algorithms in the context of programs.

7.1 Test Case Reduction

In this section, we discuss related research on different algorithms and applications of test case reduction.

7.1.1 Algorithms

Over the past few years, test case reduction has become a popular and interesting research direction. The Delta Debugging algorithm, *ddmin*, proposed by Zeller and Hildebrandt [1] has undoubtedly played an important role in this regard. This algorithm provides a general framework similar to binary search that is capable of reducing inputs from any domain. In spite of its generality, Delta Debugging is primarily unsuitable for minimizing test cases with structure. This is because Delta Debugging treats a test case as a flat input without reasoning about the relationships among its atomic elements.

To compensate for this shortcoming, a variant of Delta Debugging called Hierarchical Delta Debugging (HDD) was proposed by Misherghi and Su [14] that applies Delta Debugging on the levels of the parse tree generated for the structured input. Leveraging the structure of a test case in the form of a parse tree increases the likelihood of reducing related portions together, creating smaller test cases faster compared to the original Delta Debugging algorithm.

Although HDD improves the efficiency and effectiveness of Delta Debugging, it still suffers from drawbacks. More specifically, HDD does not preserve syntactic validity of a test case during reduction. This is due to the fact that HDD only uses the domain knowledge to build a parse tree at the beginning of the reduction and does not utilize it to further guide the reduction process. As a result, a large number of syntactically invalid test case variants can be generated during reduction. Performing tests on these variants is always unsuccessful, leading to an increase in the reduction time without any impact on the reduced size. To mitigate this problem, Misherghi proposed a syntax guided version of HDD in a dissertation format [23]. By computing the minimal syntactically valid string for each grammar rule and its corresponding node in the parse tree, the syntax guided HDD reduces the total number of invalid tests and the overall test case reduction time by only running tests that adhere to the syntactic constraints of the input domain.

Since HDD and its syntax guided version were first proposed, extensive research has been done to improve them. To this end, Hodován and Kiss proposed Modernized Hierarchical Delta Debugging [19] that uses an extended form of context-free grammars to replace recursive rules with rules that have quantifiers. In contrast to HDD, Modernized HDD generates parse trees that are more balanced, leading to potentially more reduction. Moreover, using quantifiers in context-free grammars makes syntactically valid reduction opportunities more explicit.

To enable reduction operations other than removing nodes from the parse tree, Morton and Bruno proposed performing node substitutions in their reducers, FlexMin [10] and SIMP [9] that were introduced for simplifying database queries. These reducers perform HDD's regular reduction operations, while they are also capable of restructuring trees by performing node replacements. These replacements can allow for more reduction, yielding potentially smaller final reduced test cases. To preserve syntactic validity when replacing nodes in the parse tree, FlexMin and SIMP leverage the grammar knowledge to identify syntactically compatible descendant nodes.

Replacing nodes is also proposed by Herfert et al. [22] in their approach called generalized tree reduction (GTR). Similar to FlexMin and SIMP, GTR tends to preserve syntactic validity in its reduction. To this end, GTR limits its search space to the children of a node and instead of using the grammar knowledge, it leverages a corpus of data that has been automatically collected from a large number of test cases in the same domain. Using this corpus of data, GTR filters out those candidate replacements that will restructure the tree such that the new structure has not been observed in the corpus. The GTR's filtering mechanism can be useful if the grammar of the test case is not available (e.g., due to the ownership rights). However, the possibility of the false negatives can make GTR miss reduction opportunities by skipping valid replacements that have not been observed in the corpus.

Finally, a recent work by Vince et al. [37] thoroughly investigates the node replacement operation in a version of HDD called HDD extended with hoisting and shows that node replacements can indeed lead to smaller reduced test cases. For each technique that we investigate in this dissertation, we provide results for both removal only and removal and replacement reduction. Our results also support the idea of applying node replacements when smaller outputs are desired.

As another measure to improve HDD, Hodován and Kiss [36] introduced a simple but effective algorithm to squeeze the parse tree vertically. This algorithm skips performing tests on nodes present in the middle part of a chain that have a minimal string identical to the last node of the chain. A chain is defined as a sequence of nodes with only one child. Using this algorithm, the efficiency of HDD improves by skipping tests that can not have any impact on the size of the reduced test case. Our syntactical removability pruning algorithm proposed in Section 4.4 is similar to the idea of Hodován and Kiss for chains.

Coarse Hierarchical Delta Debugging [16] is another improvement over HDD which can be used as a standalone reduction technique if 1-minimality is not a strict requirement. It can also be used as a preprocessing step for the original HDD. Coarse HDD skips performing tests on nodes that do not have an empty minimal string according to the grammar and only targets syntactically removable nodes. As a result, it has a smaller search space that can speed up reduction.

HDDr [17], a recursive variant of Hierarchical Delta Debugging is another proposed improvement that recursively applies HDD on the children of nodes starting from the root node. Hence, instead of applying the Delta Debugging algorithm on all the nodes present at a level, HDDr applies Delta Debugging on the siblings, increasing the likelihood of successful tests by considering interrelated nodes together.

In addition to the solutions presented to improve Hierarchical Delta Debugging, there are works that try to improve the original Delta Debugging algorithm, *ddmin*, itself.

A lack of dependence among tests performed by Delta Debugging has enabled parallel implementations of its algorithm [31]. Berkeley Delta [62] makes use of a line-based partitioning mechanism to generate subsets and complements that are more likely to adhere to the domain constraints. Generalizing the split factor of Delta Debugging is another improvement measure proposed in the literature to accelerate the progress of the algorithm [63].

Groce et al. [33] extend Delta Debugging to achieve test cases that are more suited for scenarios other than failures. Lithium [18] is another improvement over Delta Debugging which performs revisiting tests only at the finest granularity. The idea of our technique, One Pass Delta Debugging (OPDD) [2] is similar to Lithium. However, OPDD skips performing revisiting tests at all granularities, including the finest granularity. Moreover, OPDD is the first work to propose the notion of deferred removal and empirically exploiting it along with satisfaction of unambiguity described by Zeller [34] and common dependence order in practice to speed up test case reduction.

Recently, Wang et al. [32] proposed Probabilistic Delta Debugging (PDD), a variation of Delta Debugging that works by learning the probability of removal success for each element in the test case based on the collection of prior tests. The original idea of our PROBABILISTIC JOINT REDUCTION technique was inspired by the PDD algorithm. However, we apply the modifications discussed in this dissertation to make the reduction algorithm more suitable for structured domains. Additionally, we empirically demonstrate that running our TYPE BATCHED REDUCER in combination with PROBABILISTIC JOINT REDUCTION technique can further improve the performance of PDD-based joint reducer.

Using dynamic tainting to reduce inputs of programs [64], searching through file-based subsets of a test case exhaustively [65], isolating failure causes by comparing passing and failing runs [66, 67, 68, 69, 70], embedding internal reduction into test case generation tools [71], and optimizing test suites rather than simplifying test cases [72, 73] are among other remarkable reduction techniques in the literature.

7.1.2 Applications

Test case reduction techniques, particularly the Delta Debugging algorithm, *ddmin*, have been widely used in a variety of domains and applications in practice.

To mention a few, using the Delta Debugging algorithm, Orso et al. [74] automatically isolate the subset of the interactions between components of a large system and their environment, Hammoudi et al. [8] reduce recordings of events that lead to a failure of a web application, and Choi and Zeller [11] simplify the failure-inducing thread schedules of concurrent programs.

DEMi, a tool developed by Scott et al. [75], minimizes faulty executions of distributed systems by applying Delta Debugging as part of its minimization phase and Clapp et al. [12] propose a technique for minimizing GUI event traces generated for Android applications by utilizing a variant of Delta Debugging. Lei and Andrews [76] generate randomized unit tests and show that applying Delta Debugging is very effective for reducing sequences of method calls in the failing tests generated during the fuzz testing process [77]. Similarly, Pike introduces SmartCheck [78] to reduce counterexamples generated by QuickCheck [79], a tool for random testing of Haskell programs. Leitner et al. [80] propose that applying static slicing in conjunction with Delta Debugging could result in a more efficient test case minimization technique. Similarly, Gupta et al. [81] integrate Delta Debugging with the benefit of forward and backward dynamic program slicing. Brummayer and Biere [82, 83] devise a grammar-based black box fuzz testing technique for generating robust SMT solvers and integrate Delta Debugging into their technique to obtain minimized SMT formulas that are more useful for test case generation, debugging and verification.

7.2 Program Reduction

Program reduction is a specific kind of test case reduction in which reduction is performed on program source code typically used as an input to a compiler or interpreter. These programs in the form of complex and structured data are often generated randomly during the fuzz testing of the compiler or interpreter and may contain parts that are not relevant to the failure [4, 13]. Two different types of program reducers are proposed in the literature to simplify these programs: domain specific and domain agnostic tools.

Domain specific reducers are powerful tools capable of reducing programs of a single domain. Their development requires extensive domain specific knowledge and expertise. The most well-known domain specific program reducer is C-Reduce [13], an effective tool for reducing C programs. C-Reduce has inspired similar domain specific program reducers, each hand tailored for a specific programming language domain. Among them are Elm-Reduce [6], a reducer for the test cases written in the Elm programming language, JS Delta [84], developed for the JavaScript domain, spirv-fuzz [85], a tool specific to the SPIR-V domain, an intermediate representation used by the OpenCL programming models, J-Reduce for Java bytecode [20, 21], and ReduKtor for Kotlin [86]. Our domain agnostic techniques proposed in this dissertation can be used as both standalone or complementary to the domain specific reducers.

A recent domain agnostic program reducer is Perses [15]. It performs reduction in a syntax guided manner using a priority queue. Although Perses is introduced as a program reducer, it is capable of reducing inputs of any domain with a given grammar, including test cases that are not programs such as XML files.

This dissertation tackles the problem of speeding up test case reduction with a particular eye on improving Perses. More specifically, we address the shortcomings present in the priority mechanism of Perses, how it traverses the parse tree, and the semantically invalid tests generated during its reduction. We propose PARDIS and PARDIS HYBRID [28], two syntax guided priority aware reducers. In contrast to Perses, PARDIS and its hybrid version aim at spending the reduction effort on those portions of the test case that can have a larger impact on the reduction progress. We further propose a model guided version of PARDIS called MODEL GUIDED PARDIS [29] that can mitigate the problem of semantically invalid tests generated by Perses and PARDIS.

In addition to simplifying test cases, techniques for program reduction are helpful for removing unnecessary features, reducing attack surfaces, and decreasing security vulnerabilities. This process called program debloating has been extensively researched in the literature [38, 35, 40, 39, 87, 41].

Program reduction has also demonstrated recent utility across domains other than test case reduction and program debloating. Resource efficiency is one area in which eliminating features can shrink programs to increase their efficiency in IoT settings [25]. In addition, program reduction can benefit machine learning by helping to understand models of code [42, 26]. Trace simplification [88, 89], abstracting an executable slice from the program [90], fault localization [91], model checking software [92], and reducing execution paths [93] are among other use cases of program reduction.

7.3 Machine Learning and Programs

Models to perform tasks related to programs are continuously trained and used. Raychev et al. [94] make use of decision trees to propose probabilistic models for code completion and repair. Mesbah et al. [95] leverage neural machine translation in their tool called DeepDelta to repair the code that does not compile. Similarly, bug detectors based on deep learning have received attention in the literature [96, 97, 98, 99, 100]. There are works using machine learning to help facilitate static analysis [101, 102, 103, 104]. Grieco et al. [105] make use of static and dynamic features to predict if a test case is likely to contain a software vulnerability using machine learning techniques. Xue et al. [106] propose a machine learning based binary code analysis to facilitate malware detection and code refactoring tasks, and Brun and Ernst [107] identify a subset of properties that are most likely to reveal an error using a classifier.

Finally, several techniques have integrated learning into the reduction process in some form. Among traversal based reducers, GTR [22] learns from a corpus which tree transformations to consider at a node. Both Model Guided Pardis [29] and Type Batched Reducer proposed in this dissertation leverage models to accelerate reduction by skipping invalid tests and prioritizing candidates with higher probability of success. CHISEL [35] applies reinforcement learning to select which Delta Debugging action (test subset, test complement, refine granularity) to apply on a list. Probabilistic Delta Debugging (PDD) [32] updates its belief in each element of a list being removable based on failed oracle queries, attempting to remove subsets of a list that look most promising. Neither Chisel nor PDD leverages prior knowledge about the probability of success. They both solely rely on updating beliefs based on observed oracle failures during reduction.

Chapter 8

Conclusions

In this dissertation, we proposed a variety of techniques to accelerate test case reduction, which is a crucial step for efficiently testing and debugging software. Given a test case with a property of interest, such as exhibiting a bug in a software under test, our techniques build a parse tree or abstract syntax tree (AST) of the test case using a grammar and traverse the tree to apply generalized reduction transformations, such as node removals and node replacements, to generate a smaller variant of the test case that still preserves the property. Since our techniques require only a parse tree (or a grammar to build it), they are generalized and domain agnostic, meaning that they are capable of simplifying inputs from multiple domains such as test cases written in different programming languages.

In particular, we investigated Perses [15], the latest state of the art domain agnostic test case reducer that leverages a priority queue to order visiting nodes in the parse tree, and identified some of its shortcomings that can slow down the reduction process. We then proposed solutions to address these limitations.

To this end, we first proposed One Pass Delta Debugging (OPDD) [2] which improves the theoretical bounds of the classic Delta Debugging algorithm [1] by converting its $O(n^2)$ worst case time complexity into just $O(n)$. By skipping unnecessary reduction steps, OPDD achieves better empirical results. Because Perses makes use of Delta Debugging to reduce lists of nodes, it directly benefits from our One Pass Delta Debugging algorithm. Using OPDD led to an average speed up of 1.16x, 1.04x, and 1.20x when reducing C, Rust, and Go programs of our benchmark, respectively while generating outputs of the same size.

Next, we proposed a more efficient traversal order of the tree by devising two new techniques, Pardis and Pardis Hybrid [28]. These reducers further accelerate Perses' reduction by prioritizing nodes that may have a larger impact on the reduced size and visiting them early in the reduction process. Using either Pardis or Pardis Hybrid when reducing our benchmark led to an average speed up of 2.13x, 1.54x, and 1.78x for C, Rust, and Go domains, respectively while preserving a similar reduction power compared to the state of the art Perses.

Model Guided Pardis [29] was another reduction technique proposed in the dissertation to mitigate the problem of semantically invalid test case variants generated during reduction. To avoid generating these variants that are expensive and hinder successful reduction, Model Guided Pardis consults models that we have trained using a large corpus of data to predict and skip semantically invalid tests. By leveraging simple syntactic properties of grammars such as rule types, we trained models that could speed up reduction by 1.29x, 1.49x, and 1.46x on average for C, Rust, and Go domains, respectively compared to our Pardis technique. While this came at the price of a slight increase in the average reduced size, our best models exhibited precision and recall rates above 60% when reducing the majority of real-world test cases in our benchmark.

Finally, we proposed Type Batched Probabilistic Joint Reduction [30] which again uses machine learning to suggest portions of a test case that are most likely to be advantageous to reduce at a particular point in the reduction. Using our Type Batched Reducer, we can guide the reduction process with the goal of maximizing the expected rate of reduction. We further extended this to jointly reduce multiple portions of the test case at once. Compared to the state of the art Perses, Type Batched Probabilistic Joint Reduction led to an average speed up of 2.42x, 1.11x, and 1.41x when reducing C, Rust, and Go programs, respectively. The outputs generated by Type Batched Probabilistic Joint Reduction were either smaller or equal in size to those produced by Perses.

The performance results of our techniques are promising and show a significant improvement in the speed of test case reduction. Our reduction strategies can be applied both independently and as complementary to domain specific reducers to enhance their speed. We expect that by minimizing the disruptions caused by long test case reduction times, our more efficient reducers can assist developers in maintaining their productivity. We also believe that our proposed techniques can open up new possibilities for future research.

Bibliography

[1] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

[2] Golnaz Gharachorlu and Nick Sumner. Avoiding the familiar to speed up test case reduction. In *2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018*, pages 426–437. IEEE, 2018.

[3] Penny A. Grubb and Armstrong A. Takang. *Software maintenance - concepts and practice (2. ed.)*. World Scientific, 2003.

[4] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011.

[5] Marcel Böhme and Soumya Paul. On the efficiency of automated testing. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 632–642. ACM, 2014.

[6] Philipp Krüger. Elm-reduce: Delta debugging functional programs. Master's thesis, Karlsruhe Institute of Technology, Karlsruhe , Germany, 2019. (Approved).

[7] How to Minimize Test Cases for Bugs. `https://gcc.gnu.org/bugs/minimize.html`.

[8] Mouna Hammoudi, Brian Burg, Gigon Bae, and Gregg Rothermel. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 333–344. ACM, 2015.

[9] Nicolas Bruno. Minimizing database repros using language grammars. In Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan, editors, *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, volume 426 of *ACM International Conference Proceeding Series*, pages 382–393. ACM, 2010.

[10] Kristi Morton and Nicolas Bruno. Flexmin: a flexible tool for automatic bug isolation in DBMS software. In Goetz Graefe and Kenneth Salem, editors, *Proceedings of the Fourth International Workshop on Testing Database Systems, DBTest 2011, Athens, Greece, June 13, 2011*, page 1. ACM, 2011.

[11] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In Phyllis G. Frankl, editor, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, pages 210–220. ACM, 2002.

[12] Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. Minimizing GUI event traces. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 422–434. ACM, 2016.

[13] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 335–346. ACM, 2012.

[14] Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 142–151. ACM, 2006.

[15] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: syntax-guided program reduction. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 361–371. ACM, 2018.

[16] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Coarse hierarchical delta debugging. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 194–203. IEEE Computer Society, 2017.

[17] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. Hddr: a recursive variant of the hierarchical delta debugging algorithm. In Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir, editors, *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, pages 16–22. ACM, 2018.

[18] Jesse Ruderman. *Lithium*. `https://www.squarefree.com/lithium`.

[19] Renáta Hodován and Ákos Kiss. Modernizing hierarchical delta debugging. In Tanja E. J. Vos, Sigrid Eldh, and Wishnu Prasetya, editors, *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2016, Seattle, WA, USA, November 18, 2016*, pages 31–37. ACM, 2016.

[20] Christian Gram Kalhauge and Jens Palsberg. Binary reduction of dependency graphs. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 556–566. ACM, 2019.

[21] Christian Gram Kalhauge and Jens Palsberg. Logical bytecode reduction. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1003–1016. ACM, 2021.

[22] Satia Herfert, Jibesh Patra, and Michael Pradel. Automatically reducing tree-structured test inputs. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 861–871. IEEE Computer Society, 2017.

[23] Ghassan Shakib Misherghi. Hierarchical delta debugging. Master's thesis, Computer Science, University of California, Davis, 6 2007.

[24] Chris Parnin and Spencer Rugaber. Resumption strategies for interrupted programming tasks. *Softw. Qual. J.*, 19(1):5–34, 2011.

[25] Arpit Christi, Alex Groce, and Austin Wellman. Building resource adaptations via test-based software minimization: Application, challenges, and opportunities. In Katinka Wolter, Ina Schieferdecker, Barbara Gallina, Michel Cukier, Roberto Natella, Naghmeh Ramezani Ivaki, and Nuno Laranjeiro, editors, *IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2019, Berlin, Germany, October 27-30, 2019*, pages 73–78. IEEE, 2019.

[26] Md. Rafiqul Islam Rabin, Aftab Hussain, and Mohammad Amin Alipour. Syntax-guided program reduction for understanding neural code intelligence models. In Swarat Chaudhuri and Charles Sutton, editors, *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022*, pages 70–79. ACM, 2022.

[27] *Context switching is killing your productivity.* `https://www.software.com/devops-guides/context-switching`.

[28] Golnaz Gharachorlu and Nick Sumner. PARDIS: priority aware test case reduction. In Reiner Hähnle and Wil M. P. van der Aalst, editors, *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11424 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2019.

[29] Golnaz Gharachorlu and Nick Sumner. Leveraging models to reduce test cases in software repositories. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, pages 230–241. IEEE, 2021.

[30] Golnaz Gharachorlu and Nick Sumner. Type batched program reduction. In *ISSTA '23: 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle, USA, July 17 - 21, 2023,*. ACM, 2023.

[31] Renáta Hodován and Ákos Kiss. Practical improvements to the minimizing delta debugging algorithm. In Leszek A. Maciaszek, Jorge S. Cardoso, André Ludwig, Marten van Sinderen, and Enrique Cabello, editors, *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, July 24 - 26, 2016*, pages 241–248. SciTePress, 2016.

[32] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. Probabilistic delta debugging. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 881–892. ACM, 2021.

[33] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: delta debugging, even without bugs. *Softw. Test. Verification Reliab.*, 26(1):40–68, 2016.

[34] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1999.

[35] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 380–394. ACM, 2018.

[36] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Tree preprocessing and test outcome caching for efficient hierarchical delta debugging. In *12th IEEE/ACM International Workshop on Automation of Software Testing, AST@ICSE 2017, Buenos Aires, Argentina, May 20-21, 2017*, pages 23–29. IEEE Computer Society, 2017.

[37] Dániel Vince, Renáta Hodován, Daniella Bársony, and Ákos Kiss. Extending hierarchical delta debugging with hoisting. In *2nd IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2021, Madrid, Spain, May 20-21, 2021*, pages 60–69. IEEE, 2021.

[38] Zhongshu Gu, William N. Sumner, Zhui Deng, Xiangyu Zhang, and Dongyan Xu. DRIP: A framework for purifying trojaned kernel drivers. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, pages 1–12. IEEE Computer Society, 2013.

[39] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A framework for post-deployment software debloating.

In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1733–1750. USENIX Association, 2019.

[40] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. TRIMMER: application specialization for code debloating. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 329–339. ACM, 2018.

[41] Ryan Williams, Tongwei Ren, Lorenzo De Carli, Long Lu, and Gillian Smith. Guided feature identification and removal for resource-constrained firmware. *ACM Trans. Softw. Eng. Methodol.*, 31(2):28:1–28:25, 2022.

[42] Md. Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. Understanding neural code intelligence through program simplification. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 441–452. ACM, 2021.

[43] Perses: Syntax-Directed Program Reduction. `https://github.com/uw-pluverse/perses`.

[44] *C99 Standard*. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.

[45] XMLProc: An XML parser in Python. `https://github.com/mcdir/xmlproc`.

[46] random: Generate pseudo-random numbers. `https://docs.python.org/3/library/random.html`.

[47] Thesis Reducers. `https://github.com/golnazgh`.

[48] IBM Support. *Test Case Reduction Techniques*. `http://www-01.ibm.com/support/docview.wss?uid=swg21084174`.

[49] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman and Hall/CRC, 2011.

[50] Trending: See what the GitHub community is most excited about today., howpublished = "`https://github.com/trending/rust`".

[51] Trending: See what the GitHub community is most excited about today., howpublished = "`https://github.com/trending/go`".

[52] Qiang Yang, Yu Zhang, Wenyuan Dai, and Sinno Jialin Pan. *Transfer Learning*. Cambridge University Press, 2020.

[53] René Just, Bob Kurtz, and Paul Ammann. Customized program mutation: Inferring mutant utility from program context. Technical Report UM-CS-2017-004, University of Massachusetts, Amherst, Amherst, MA, USA, April 2017.

[54] Zellig Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[55] Yanli Liu, Yourong Wang, and Jian Zhang. New machine learning algorithm: Random forest. In Baoxiang Liu, Maode Ma, and Jincai Chang, editors, *Information Computing and Applications - Third International Conference, ICICA 2012, Chengde, China, September 14-16, 2012. Proceedings*, volume 7473 of *Lecture Notes in Computer Science*, pages 246–252. Springer, 2012.

[56] Vincent J. Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 763–773, New York, NY, USA, 2017. Association for Computing Machinery.

[57] Marek Kuczma. *An Introduction to the Theory of Functional Equations and Inequalities*. Birkhäuser Verlag AG, 2009.

[58] Jr Harrell, Frank E. *Regression Modeling Strategies*. Springer International Publishing AG, 2015.

[59] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[60] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2015.

[61] GCC Compiler. *GCC Torture Test Suite*.

[62] Scott McPeak, Daniel S. Wilkerson, and Simon Goldsmith. *Delta*, July 2003. `http://delta.stage.tigris.org/`.

[63] Ákos Kiss. Generalizing the split factor of the minimizing delta debugging algorithm. *IEEE Access*, 8:219837–219846, 2020.

[64] James A. Clause and Alessandro Orso. Penumbra: automatically identifying failure-relevant inputs using dynamic tainting. In Gregg Rothermel and Laura K. Dillon, editors, *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, pages 249–260. ACM, 2009.

[65] Jacqueline M. Caron and Peter A. Darnell. Bugfind: A tool for debugging optimizing compilers. *SIGPLAN Notices*, 25(1):17–22, 1990.

[66] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*, pages 1–10. ACM, 2002.

[67] Holger Cleve and Andreas Zeller. Locating causes of program failures. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 342–351. ACM, 2005.

[68] Cyrille Artho. Iterative delta debugging. *Int. J. Softw. Tools Technol. Transf.*, 13(3):223–246, 2011.

[69] Jeremias Rößler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. Isolating failure causes through test case generation. In Mats Per Erik Heimdahl and Zhendong Su, editors, *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 309–319. ACM, 2012.

[70] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. Causal testing: understanding defects' root causes. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 87–99. ACM, 2020.

[71] David Maciver and Alastair F. Donaldson. Test-case reduction via test-case generation: Insights from the hypothesis reducer (tool insights paper). In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 13:1–13:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[72] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[73] Raphael Noemmer and Roman Haas. An evaluation of test suite minimization techniques. In Dietmar Winkler, Stefan Biffl, Daniel Méndez, and Johannes Bergsmann, editors, *Software Quality: Quality Intelligence in Software and Systems Engineering - 12th International Conference, SWQD 2020, Vienna, Austria, January 14-17, 2020, Proceedings*, volume 371 of *Lecture Notes in Business Information Processing*, pages 51–66. Springer, 2020.

[74] Alessandro Orso, Shrinivas Joshi, Martin Burger, and Andreas Zeller. Isolating relevant component interactions with jinsi. In *Proceedings of the 2006 international workshop on Dynamic systems analysis, WODA 2006, Shanghai, China. May 23-23, 2006*, pages 3–10, 2006.

[75] Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George C. Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing faulty executions of distributed systems. In Katerina J. Argyraki and Rebecca Isaacs, editors, *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 291–309. USENIX Association, 2016.

[76] Yong Lei and James H. Andrews. Minimization of randomized unit test cases. In *16th International Symposium on Software Reliability Engineering (ISSRE 2005), 8-11 November 2005, Chicago, IL, USA*, pages 267–276, 2005.

[77] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[78] Lee Pike. Smartcheck: automatic and efficient counterexample reduction and generalization. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 53–64. ACM, 2014.

[79] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

[80] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 417–420, 2007.

[81] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 263–272. ACM, 2005.

[82] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging smt solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories, SMT 2009, Montreal, Canada. August 02-03, 2009*, pages 1–5, 2009.

[83] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010.

[84] *JS Delta.* `https://github.com/wala/jsdelta`.

[85] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpinski. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, pages 1017–1032. ACM, 2021.

[86] Daniil Stepanov, Marat Akhin, and Mikhail A. Belyaev. Reduktor: How we stopped worrying about bugs in kotlin compiler. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 317–326. IEEE, 2019.

[87] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. Program debloating via stochastic optimization. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020*, pages 65–68. ACM, 2020.

[88] Nicholas Jalbert and Koushik Sen. A trace simplification technique for effective debugging of concurrent programs. In Gruia-Catalin Roman and André van der Hoek, editors, *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 57–66. ACM, 2010.

[89] Jeff Huang and Charles Zhang. An efficient static trace simplification technique for debugging concurrent programs. In Eran Yahav, editor, *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2011.

[90] Mark Weiser. Program slicing. In Seymour Jeffrey and Leon G. Stucki, editors, *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981*, pages 439–449. IEEE Computer Society, 1981.

[91] Ezekiel O. Soremekun, Lukas Kirschner, Marcel Böhme, and Andreas Zeller. Locating faults with program slicing: an empirical analysis. *Empir. Softw. Eng.*, 26(3):51, 2021.

[92] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005.

[93] Kostas Ferles, Valentin Wüstholz, Maria Christakis, and Isil Dillig. Failure-directed program trimming (extended version). *CoRR*, abs/1706.04468, 2017.

[94] Veselin Raychev, Pavol Bielik, and Martin T. Vechev. Probabilistic model for code with decision trees. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 731–747. ACM, 2016.

[95] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: learning to repair compilation errors. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 925–936. ACM, 2019.

[96] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019.* OpenReview.net, 2019.

[97] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020.* OpenReview.net, 2020.

[98] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR, 2020.

[99] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 27865–27876, 2021.

[100] Zimin Chen, Vincent J. Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. PLUR: A unifying, graph-based view of program learning, understanding, and repair. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 23089–23101, 2021.

[101] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound static analysis. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 519–529. IEEE / ACM, 2017.

[102] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017*, pages 42–54. ACM, 2017.

[103] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. Continuously reasoning about programs using differential bayesian inference. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 561–575. ACM, 2019.

[104] Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. Automatically generating features for learning program analysis heuristics for c-like languages. *Proc. ACM Program. Lang.*, 1(OOPSLA):101:1–101:25, 2017.

[105] Gustavo Grieco, Guillermo Luis Grinblat, Lucas C. Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. Toward large-scale vulnerability discovery using machine learning. In Elisa Bertino, Ravi S. Sandhu, and Alexander Pretschner, editors, *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, LA, USA, March 9-11, 2016*, pages 85–96. ACM, 2016.

[106] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access*, 7:65889–65912, 2019.

[107] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 480–490. IEEE Computer Society, 2004.