BUILDING RESOURCE ADAPTATIONS VIA TEST-BASED SOFTWARE MINIMIZATION: APPLICATION, CHALLENGES, AND OPPORTUNITIES

By

David Weber

A thesis

Submitted to the faculty of the

MSCS Graduate Program of Weber State University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

December 16, 2022

Ogden, Utah

Approved: Date:

____________________ ____________________

Committee Chair, Arpit Christi, Ph.D.

____________________ ____________________

Committee member, Yong Zhang, Ph.D.

____________________ ____________________

Committee member, Nicole Anderson, PhD.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Arpit Christi, and my committee members, Dr. Yong Zhang, and Dr. Nicole Anderson, for their guidance and support throughout the course of this research. In addition, I would also like to thank Chelsea Johnson, Nathan Cummings, and additional friends and family for their support during the course of this research. .

ABSTRACT

Modern software systems can be complex, which can require more time dedicated to locating, isolating, and fixing faults. Even with a failing test, this process is tedious and time-consuming. Simplifying failing tests can significantly reduce the development effort by reducing the irrelevant program entities that developers observe. Delta Debugging (DD) algorithms automatically reduce the failing tests. Hierarchical Delta Debugging (HDD) algorithms improve DD for hierarchical tests like source code and HTML files. Many modern implementations of these algorithms work on a generic tree-like structure and fail to consider the complex structures and the interdependence of program elements of programming languages. We propose a tool, ReduSharptor, to simplify C# tests that uses language-specific features and the interdependence of C# program elements using the Roslyn compiler APIs. We evaluate the tool on a set of 30 failing C# tests to demonstrate its applicability and accuracy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The complexity of modern software makes debugging difficult and time consuming. To debug a failing program, the developer needs to locate and isolate the fault first; a slow and tedious process known as Fault Localization (FL). If the failing tests only execute a few faulty program elements, FL is trivial. The complexity arises from the fact that failing tests often execute a large set of non-faulty program elements. Hence, simplification of failing tests, while keeping the bug, reduces the complexity of fault localization by reducing the number of non-faulty program elements the developers need to search. It focuses developers' attention on a few faulty program elements, leading to faster debugging times. Simplified failing tests are not only helpful aid to developers, it can significantly improve the accuracy of automatic fault localization techniques [1, 2]. These automatic fault localization techniques have the goal of automatically finding the faulty program elements without the need of a developer to search through them. While a long, complex, failing test leads to longer execution times for these techniques, simplifying the unit tests before this step reduces the execution time significantly.

The most widely known and utilized automatic test simplification technique is the Delta Debugging (DD) algorithm, by Zeller and HildeBrandt. This algorithm works well on test inputs that can be considered array or list-like structures [3]. Additionally, it is not the most effective technique for tests that have tree-like structures as seen in HTML files, C or java programs, or XML files. This is due to the fact that it only works on a flat structure, or in other words, will not break apart smaller blocks in order to simplify those sections as well. However, Mishreghi and Su proposed the Hierarchical Delta Debugging (HDD) algorithm that utilizes the underlying Abstract Syntax Tree (AST) structure to effectively simplify these unit tests. [4].

Recently, a few researchers proposed modern implementations of HDD algorithms and their variants [5, 6, 7, 8]. Most of the implementations are language-agnostic and, hence, can reduce a variety of tests in languages such as HTML, XML, C, or java. Stepanov et al. noted the language-agnosticism of the HDD tools. This is a major limiting factor in employing the tools efficiently for real-world, large-scale usage as the tools fail to consider and utilize the language-specific features, complexities, and inter-dependencies [8]. These tools rely on a generic AST or grammar in the simplification process and produce many noncompilable intermediate variants before the convergence. Sun et al. noted the need of producing syntactically correct intermediate test variants while proposing the *Perses* algorithm [6]. Also, most of the tools rely on many libraries, components, and external tools that need to be up-to-date all the time to utilize the tools. Binkley et al. argue that the cost of development and maintenance is prohibitive for program slicing tools (DD/HDD produces a slice) due to the need of a large set of libraries and components [9]. Many of these tools require a certain preprocessing step before they can be utilized to simplify tests [5, 6].

Instead of focusing on varying sets of test inputs and test cases, the focus was on *developer-written C#* unit tests. As we focused our attention, observed, and studied unit tests implemented in C# by developers, we noticed that we can utilize new avenues to implement a test reduction tool that is applicable, accurate, and easy to use.

To this end, we propose a tool, ReduSharptor, that provides the following:

1. A tool specifically implemented for C# tests that utilizes language-specific features of C# programs and tests.
2. A tool that utilizes an empirical analysis to prune the search space.
3. A tool that exists as a stand-alone entity and does not require any further libraries or tool sets. This tool can be invoked using an executable file.
4. A tool that requires absolutely no preprocessing steps.

We evaluate ReduSharptor on a set of 30 failing tests on 5 open source C# projects to demonstrate that ReduSharptor is applicable and accurate. The tool can produce correct test simplifications with high precision (96.58%) and recall (96.45%). ReduSharptor is publicly available on GitHub.

CHAPTER 2

Related Work

2.1 DD and HDD

DD is an algorithm that simplifies failing tests while still keeping the bug by utilizing a variant of binary search to remove individual components that are unnecessary for triggering the bug [3]. To retrofit DD for hierarchical test inputs like XML, HTML, or programs, the top syntax tree is used as a flat structure. This means the elements would include blocks of nested elements for removal. This method is temporally efficient since it doesn't entail further nested elements.

While DD is useful alone, it is not effective for all scenarios. This led Misherghi and Su to propose that HDD works efficiently and effectively on tree-like inputs by exploiting the underlying AST [4]. This AST allows for the HDD algorithm to break down the blocks of code into smaller blocks of code, thus allowing for the algorithm to run recursively. While both DD and HDD are theoretically sound algorithms that guarantee convergence and minimalism, HDD is able to break down the statements further, allowing for more effective simplification.

One well-known program used to reduce, *C-Reduce*, is used to reduce programs written in the C language. Regehr et al. utilized DD/HDD to propose *C-Reduce* to minimize these programs for compiler testing [10]. This is a much more powerful program than others with a similar purpose. This is due to the program allowing it to perform all of the same abilities of both DD and HDD in one program, allowing the user to use one program for both scenarios.

2.2 Additional resources

Even though HDD was able to create a tree structure, it did not do anything about changing the structure of the syntax tree. This would lead the tree to be imbalanced at times, depending on the specific structure. This led Hodovan and Kiss to research further, and claim that Extended Context Free Grammar produces a more balanced tree than one produced by Context Free Grammar. They then utilized it in implementing a modernized HDD tool called picireny [5].

Herfert et al. proposed an additional algorithm known as the Generalized Tree Reduction (GTR) algorithm. The GTR algorithm relies on operations other than removal or deletion and replacing a tree node with a similar tree node [11]. This presented an effective alternative to DD and presented the idea that DD/HDD is not the only syntax tree simplification option.

While all these resources find ways of improving the algorithm itself, there are plenty of other areas for performance boosts. Sun et al. observed that during the simplification process, many previous algorithms produced *syntactically invalid variants*. A futile compilation step needs to be performed before pruning the invalid variant. They proposed the *Perses* algorithm specifically to avoid generation of invalid variants [6]. By knowing about these *syntactically invalid variants* before compiling, it makes the application of these algorithms more time efficient since it reduces the amount of time compiling each variant.

Another useful algorithm was found with the need to generalize test inputs. Gopinath et al. utilized the *Perses* algorithm to propose the *DDSET* algorithm to abstract minimal failure-inducing input from a larger set using input grammar [7]. This is a very unique approach to the problem and allows information to be captured about the unit test failure at the source in order to provide a better picture about the issue.

A large problem with the research so far is the level of abstraction with it. There are several different languages that each provide their own syntax, creating issues with each specific language. This led *Picireny*, *Perses*, and *DDSET* to use Antlr, a powerful parser generator, to produce the AST for specific programming languages. Antlr provides the ability to produce a parser for several programming languages without creating each individually.

Binkley et al. proposed another useful resource as the *Observational-based Slicing* (ORBS) technique. This technique uses program line deletion as a fundamental operation to slice programs accurately and efficiently [9]. This deletes potential slices of the program and observes and compares the behavior of the program before and after deletion. If the program behaves the same in both the original and the slice, then the deletion is kept. This is another useful technique to use for different situations where the alternatives do not make sense.

An additional resource came from Christi et al. when they combined inverted HDD with statement deletion mutation to simplify programs for the purpose of resource adaptations. They argued that reduction is meaningful and useful at statement level and avoided non-statement level reductions [12]. This presented the perspective that simplification performance can be effective by simply focusing on statement reduction rather than the alternative.

CHAPTER 3

Motivation

3.1 Purpose of the minimized test

If test minimization is used for compiler testing, even a noncompilable piece of source code can be a useful artifact in debugging and bug isolation. Our focus is to reduce the failing unit tests to aid developers in debugging. Hence, the end product of test simplification must be compilable and executable tests that remain with the same failing logic. Any intermediate test that has compilation errors will be pruned and will not be used for further processing by the simplification process since the tool cannot produce a pass/fail result on such a test.

3.2 The cost of compilation

Whenever any changes are made in either the program or test, the source code needs to be compiled before executing the test. In test reduction, we always modify or reduce the test. Hence, the test project, library, or jar needs recompilation. For real-world test projects, the compilation time can be very high. For example, after a change is made in any of the tests, the `language-ext` project has a compilation time of approximately 11 seconds on a Windows machine with Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz processor and 16.0 GB RAM. While 11 seconds may not seem to be a significant cost, this time would need to be multiplied with every possible variant. Those 11 seconds of compilation time can turn into significantly larger execution times for simplification algorithms. Reducing the total number of variants needed to compile will significantly reduce the overall execution time.

```
[Fact]
public void ApplySomeArgs()
{
  var opt = Some(add)  // 1
    .Apply(Some(3))   // 2
    .Apply(Some(4));  // 3

  Assert.Equal(Some(7), opt);  // 4
}
```

Figure 3.1: `ApplySomeArgs` test in `language-ext`

3.3 Performance of other techniques

Since there are other techniques used for simplification, there are possibilities of better performance using one of those. However, if we simulate the behavior of the *ORBS* or *Perses* techniques on the provided test 3.1, we notice the potential of producing more variants that cannot compile. Therefore, while there are still noncompilable variants within this example, there is no alternative that provides a better approach.

The *ORBS* technique relies on line-level reduction and, hence, it may produce variants where line 1 or line 3 are removed from the test; both of which are noncompilable. The *Perses* technique attempts to produce a syntactically correct variant, but syntactic correctness does not always result in successful compilation. For example, in line 2, `.Apply(Some())` and `.Apply()` are syntactically correct variants, but are still noncompilable. The *Perses* technique will produce many such variants for the given test leading to more time put into compilation.

3.4 Using statements as the unit of reduction

Instead of using nodes in the AST or lines in the test file as the basis of reduction, ReduSharptor uses program statements for the unit of reduction. The statement is defined by the `StatementSyntax` class or other derived classes of the Roslyn compiler API class [13]. With statements as the unit of reduction, lines 2, 3, and 4 will be treated as a single statement of type `LocalDeclarationStatementSyntax` by the Roslyn compiler. Hence, It can only produce one variant that cannot compile - the variant where the entire first statement is deleted. This results in fewer noncompilable variants to be tested, meaning less time wasted on compilation in general.

3.5 Fewer intermediate variants

When we use statements as the unit of reduction, we are essentially considering the AST with significantly less nodes because we ignore the existence of nodes below the statement level. As the DD/HDD algorithm will have to process fewer nodes, a large number of variants will be pruned automatically, resulting in considerable reduction in the search space. Therefore, less time will be required to simplify the test because a large number of variants will not need to be compiled and tested.

3.6 DD is sufficient

Consider the fictitious test case shown in figure 3.3. The corresponding AST representation is available in figure 3.2. The figure only shows statement nodes as we already argued for not using nodes below the statement level. Now consider two nodes that correspond to lines 1 and 2 of figure 3.3. Such statements do not have a sub tree with our statement deletion assumptions. The `if` statement spanned across lines 3, 4, and

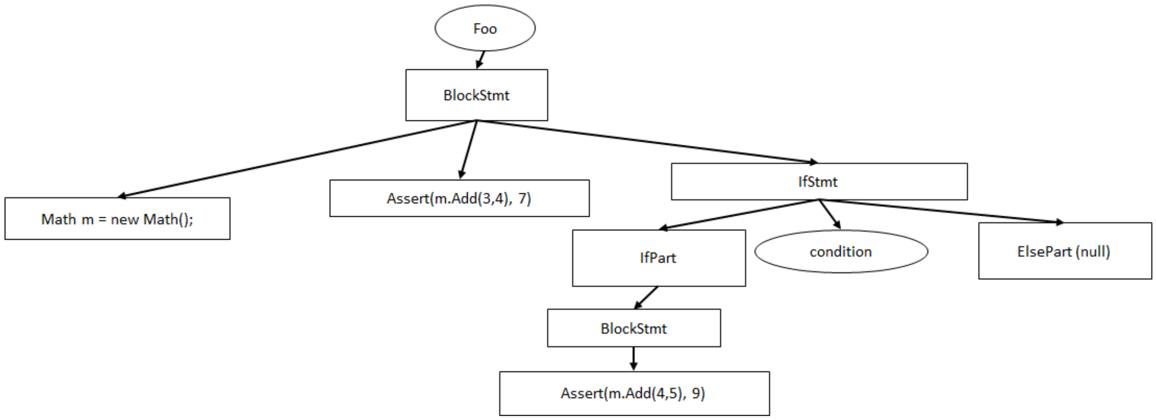

Figure 3.2: AST of code in Figure 3.3

```
1 [Fact]
2 public void Foo()
3 {
4   Math m = new Math();  // 1
5   Assert.Equal(m.Add(3,4),7);  // 2
6   if(true){  // 3
7     Assert.Equal(m.Add(4,5),9);  // 4
8   }  // 5
9 }
```

Figure 3.3: `foo` test to demonstrate AST

5 results in a tree. We divide the Roslyn compiler statement sets into two distinct sets: *NonTree* statements that *cannot* form sub trees, and *Tree* statements that *can* form sub trees. We conducted an empirical analysis on 1000 distinct developer written unit tests and observed the statement usage. We found *Tree* statements are infrequent in *developer-written* C# unit tests. Therefore, we will consider a *Tree* statement to be a *NonTree* statement. We will process the `if` statement as a single statement instead of processing the corresponding sub tree. For the figure 3.3 code, this means treating lines 3, 4, and 5 as a single statement. Either the entire block is removed or nothing is removed. We don't have any chance to separately process the `Assert` statement in line 4. This approach provides two advantages: fewer statements need to be processed, and all statements below block statements are considered *Tree* statements. We have a list or set of *NonTree* statements below the block statement level, and we can process them using the DD algorithm with $O(n^2)$ complexity instead of the HDD algorithm with $O(n^3)$ complexity. At first glance, we seem to be sacrificing accuracy for efficiency in the entire process, however, our results demonstrate that such simplification works well in practice.

CHAPTER 4

ReduSharptor: Usage, Architecture, and Implementation Details

4.1 Usage

For ease of this experiment, we have created a tool, ReduSharptor [14], that will simplify the unit tests. To use this tool, the developer will only have to provide the following: test file with full path, name of the test (as a single file can have many tests and we may want to reduce only one failing test), and the path of the `.csproj` file associated with the code. All of this information is already available to the developers. Optionally, the developer can provide a particular folder path if they want to use it to store intermediate results and the final output in that folder. An example of this execution can be seen in figure 4.1. The architecture from a developer's perspective is described in figure 4.2. If you compare the architecture figure with the *Perses* workflow figure and the *Picireny* architecture figure, the contrast is clear [5, 6]. Both the *Perses* and *Picireny* approaches require significant preprocessing steps that require other libraries, toolsets, and components. Both of them require a test script to be available, normally a shell file or a batch file. ReduSharptor does not require any of these as explained in the architecture section.

4.2 Architecture

As ReduSharptor is implemented for C#, it takes into consideration how C# programs are organized using `.sln` and `.csproj` files. In order to compile or run the test, ReduSharptor uses the `.csproj` file, the test, and the built-in build+run utility available as part of the `.NET` framework and Roslyn compiler to generate the necessary build+run script. The process is described in the right side of figure 4.3. On the left side, we describe how a test is processed first using the Roslyn compiler to generate the parse tree. The parse tree will go through a pruning and transformation process to produce a tree where *Tree* statements will be processed as *NonTree* statements. The test, the processing statement list, and the build+run script will then be passed to the DD algorithm to produce the minimized test. The *Perses* and *Picireny* approaches require the user of

```
ReduSharptor.exe ".\language-ext\LanguageExt.Tests\ApplicativeTests.cs" "
    ListCombineTest" ".\language-ext\LanguageExt.Tests\LanguageExt.Tests.csproj" ".\
    Simplified Test Results"
```

Figure 4.1: command line execution of ReduSharptor

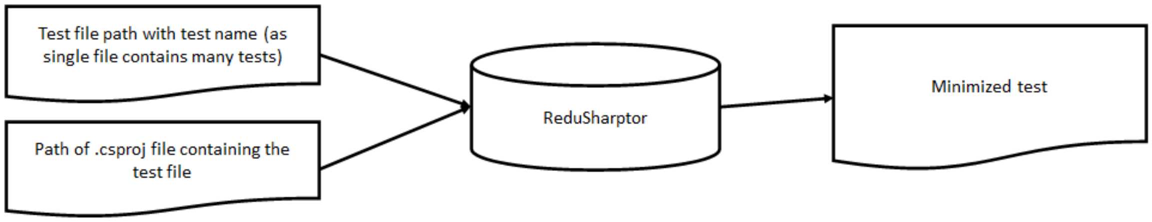

Figure 4.2: ReduSharptor architecture from a user's perspective

their tool to provide a test script which may increase in complexity over time as both approaches will require a new test script for each test minimization.

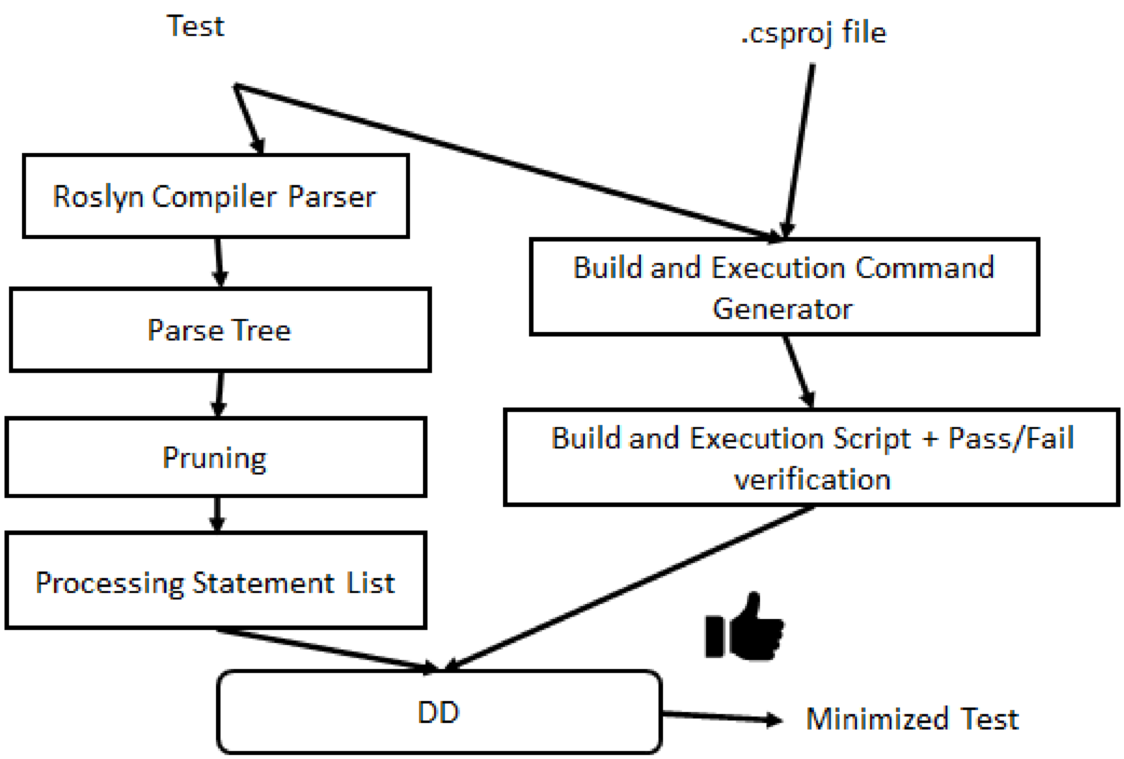

Figure 4.3: ReduSharptor internal architecture with implementation details

In addition, a great effort was made to not have any external dependencies, libraries, or tool sets and, as a result, ReduSharptor only utilizes the `.NET` framework and Roslyn compiler API, which is available as part of the `Microsoft.CodeAnalysis` library. Therefore, a user can easily invoke ReduSharptor as a command line utility without the needing to download or maintain any external components or libraries. Because ReduSharptor is a C# specific tool designed for C# unit tests, the need for preprocessing steps is eliminated.

4.3 Implementation

The execution of the simplification process is rather simple. It separates the tests into sections of statements and attempts to run each section as a standalone test. If any of the sections result in a failed test, that section is made the new test and the process is repeated. Otherwise, the complement of each section is attempted and, likewise, is made the new test if any of them fail. If none of the sections or their complements fail, then the granularity of the sections is increased and the process is repeated. This process is continued until all sections result in successful runs and the sections cannot be reduced any further. This is an adapted form of the work of A. Zeller et al. [3]. However, instead of using a set of failing input values, this approach simplifies and isolates the unit test statements themselves.

In ReduSharptor, this primary simplification code exists within the `FindSmallestFailingInput` method from figure A.2. Each `foreach` block tests the sections and their complements and are then checked to see if any of the tests are successful. Following those blocks, the granularity of the sections is increased on line 51, then checked against the number of remaining statements to ensure the number of sections does not exceed the amount of statements. If the level of granularity if more than the remaining statements, then the simplification process is finished.

In order to understand the tool fully, a great understanding must be had about how these algorithms were originally implemented. In figure A.3 rests the ReduSharptor entry point, the `Main` method. After taking input from the command line arguments, this gets the statements from the provided unit test and creates a copy as a backup. The `Main` method then calls `FindSmallestFailingInput`, which starts the simplification algorithm. After producing the simplified output, it reverts the test file to its original state and outputs the simplified statements into a separate file.

The Roslyn compiler is used in a few of the methods: figures A.6, A.7, and A.9. `GetTestStatements` and `SetTestStatements` both use Roslyn in similar ways to break apart the file and find the unit test from a name, however, `SetTestStatements` also uses Roslyn in order to write a new list of statements to the unit test. `GetTestCallString` is a little different as it is only used to create a string used to build the unit test for each compilation process. This is an additional performance increase that was utilized to speed up the process.

In figure A.1, the `BuildAndRunTest` method is relatively straightforward; it first builds and then runs the test. If the test passes, it will return `true` for the `FindSmallestFailingInput` test to be used in that algorithm. However, one piece of logic needs clarification: if the test fails to build, we return `true`. Returning a `true` value will notify the algorithm to continue with the process. Additionally, `BuildAndRunTest` is passed to the `FindSmallestFailingInput` method as an action to call. This will allow the algorithm to test each variant with this method.

In correspondance with the previously mentioned methods, figures A.4, A.5, A.8, and A.10 are also included. These are used to reduce the amount of code, and allow for an easy interpretation.

CHAPTER 5

Experiments

To evaluate ReduSharptor, we ask the following questions:

1. RQ1: How applicable is ReduSharptor?

2. RQ2: How accurate is ReduSharptor when performing failing test minimization?

5.1 Subjects

We want to use any existing C# bug repositories like Defects4J for our evaluation [15]. We are unaware of any such repository. Even the benchmark list on the program repair website does not mention any C# benchmarks [16]. We used 5 open source C# projects listed in Table 5.1. These projects are `language-ext` [17], `Umbraco-CMS` [18], `Fleck` [19], `BizHawk` [20], and `Skclusive.Mobx.Observable` [21]. Among the five, all except `Skclusive.Mobx.Observable` are under active development. After selecting the subjects, we looked for existing bugs in those projects. We went through commits to see if any of the commits or any snapshot of the software had a failing test. It seems that conscious developers normally run unit tests before committing to the repository, so we were not able to find failing tests in any snapshot of these repositories. We then searched for commits whose description seemed to be associated with a bug. We used the current version of their source code and attempted to undo the commit that appeared to fix a bug by changing the code manually, hoping to regenerate a bug. Sometimes we needed to utilize more than one related commit to recreate a bug. When a particular reversal of source code produced a failing test case, we preserved those changes as a bug and noted the failing test. The bugs, or failing tests, that we have are based on commits, but we hesitate to call them real bugs. We will call them synthetic bugs instead and hope that they bear a close resemblance to real bugs. The synthetic bugs seem to be a good intermediate solution between real bugs and mutants.

Once we had a failing test, we needed to ensure that it had at least one removable component in it, such as a statement, a block of code, or a part of an expression statement such that, after it was removed, the test continued to fail the exact same way. We pruned the failing test if we did not find any such component. Applying ReduSharptor was meaningless if there were no removable components as it would not reduce anything.

Using this process, we created 30 synthetic bugs that had 30 failing tests that were reducible.

Table 5.1: Subject projects, LOC (Line of Code), # of tests, and total commits.

Project	LOC	# Tests	# Commits
language-ext	318157	2610	3032
Umbraco-CMS	156992	2637	42491
Fleck	3576	92	237
BizHawk	1686865	98	19860
Skclusive.Mobx.Observable	7970	41	26

5.2 Process

For each failing test, we manually found the optimal minimal test that continued to fail the same way, which we refer to as the *gold standard*. As *developer-written* unit tests were simple enough to work with, it was not difficult to manually find minimal tests. We then used ReduSharptor to reduce the original failing tests.

5.3 Measurement

In order to measure the results of the experiment, a comparison was made between the results generated by ReduSharptor and the *gold standard* unit tests. The following information was collected:

1. *True-Positive* (TP) - Statements that were removed correctly.

2. *False-Positive* (FP) - Statements that were removed incorrectly.

3. *False-Negative* (FN) - Statements that were missed but should have been removed.

Using this information, it was not difficult to calculate the precision and recall and infer an analysis on the results.

CHAPTER 6

Results

6.1 Applicability

We applied ReduSharptor to 30 failing tests for synthetic bugs from 5 open-source C# projects, as seen in Table 6.1. During the application, ReduSharptor processed 759 statements and did not have any exceptions or unexpected behavior. A few issues were encountered, but they were quickly resolved. ReduSharptor was able to successfully finish and produce the minimal failing tests. The 5 projects we selected were from a range of applications, used for a variety of different purposes, and consisted of different development styles. We claim that ReduSharptor is highly applicable due to the result of the experiments on the range of subjects.

6.2 Accuracy

We report accuracy using the standard measure of precision and recall. Precision is used as the measure of correctness of a result. Recall is used to determine the true positive rate, or how many true positives are in the result. Using these as a standard, a result can be analyzed to infer the number of correct statements left, and how much confidence there is in the result.

This can be calculated with the following formulas where TP, FP, and FN have been previously collected: $recall = \frac{TP}{TP+FN}$, $precision = \frac{TP}{TP+FP}$. Using these formulas, we found ReduSharptor has 96.58% precision and 96.45% recall. We claim that ReduSharptor is highly accurate in performing failing test minimization.

6.3 Inaccuracy

Though we did not have a large data set, we evaluated our inaccuracies to further understand it. These inaccuracies consisted of false positives and false negatives. False positives are statements that ReduSharptor left in the test and parsed as needed statements when not needed for the failing logic. False negatives are statements that were removed and parsed as unneeded statements when in reality they were needed to keep the failing logic. Both of these types of inaccuracies are not ideal to have, but false positive statements are better to handle because they only take extra time to parse. False negatives are worse since they remove necessary statements.

Note that most of the false negatives are due to the *Tree* statements. This makes sense as *NonTree* statements are processed just below the Roslyn [13] `BlockStatementSyntax` level, or method level. If a *Tree* statement is present, we treated it as a single *NonTree* statement based on our observation and simplified assumption. The presence of a *Tree* statement caused missed opportunities in processing that resulted in the

Table 6.1: Simplified unit test and results of each

Unit Test	% Reduced	True Positives	False Positives	False Negatives
ListCombineTest	60%	3	0	0
EqualsTest	86%	1	0	0
ReverseListTest3	40%	3	0	0
WriterTest	47%	9	0	0
Existential	79%	3	0	0
TestMore	85%	6	2	0
CreatedBranchIsOk	72%	7	8	0
CanCheckIfUserHasAccessToLanguage	32%	12	1	0
Can_Unpublish_ContentVariation	89%	3	0	0
EnumMap	55%	5	0	0
InheritedMap	65%	4	2	0
Get_All_Blueprints	88%	3	0	11
ShouldStart	43%	4	0	0
ShouldSupportDualStackListenWhenServerV4All	75%	1	0	0
ShouldRespondToCompleteRequestCorrectly	73%	4	0	0
ConcurrentBeginWrites	86%	4	1	0
ConcurrentBeginWritesFirstEndWriteFails	81%	5	0	1
HeadersShouldBeCaseInsensitive	71%	2	0	0
TestNullability	87%	2	0	0
TestCheatcodeParsing	88%	1	0	0
SaveCreateBufferRoundTrip	77%	7	0	0
TestCRC32Stability	48%	9	5	0
TestSHA1LessSimple	50%	5	2	0
TestRemovePrefix	93%	1	0	0
TestActionModificationPickup1	39%	14	0	0
TestObservableAutoRun	88%	3	0	5
TestMapCrud	95%	2	0	0
TestObserver	97%	2	1	3
TestObserveValue	94%	2	2	4
TestTypeDefProxy	83%	8	1	1

missed removal of statements. The high precision and recall numbers suggest that our observation was correct: even if *Tree* statements are treated as a single *NonTree* statement, test minimization is still very accurate in practice.

Most of the false positives are due to tool limitations. Further investigation is needed to determine the exact cause of these false positives.

6.4 Tool comparison

To the best of our knowledge, none of the test minimization tools that we previously discussed have a C# implementation. To implement those techniques and algorithms in C#, for comparison purposes, is beyond the scope of this paper. However, because of the results we have received, this research was submitted to the SANER2023 conference for their review. This will allow ReduSharptor to gain more attention and open the door for more comparisons to be made in further research.

CHAPTER 7

Conclusion

7.1 Further Work

Even though ReduSharptor shows usefulness and effectiveness, there is still much that can be improved. One major downside to this approach is the lack of HDD in the program. ReduSharptor will attempt to parse the unit tests as a flat structure every time, regardless of statement structure. This has the potential to perform poorly and inaccurately for more complex tree structures that may contain loops, conditional statements, or action type statements. The results of this research show that a majority of C# tests consist of flat statement structures. Therefore, we can take advantage of this and use DD for the majority of these tests. If we then implement an HDD approach as well, we can benefit from both approaches for this tool. Using DD for flat structures for faster parsing and using HDD for complex trees allows for a greater effectiveness of the simplification.

Another area of improvement that can be researched is performance enhancement. Simplifying these 30 unit tests using ReduSharptor took a considerable amount of time. For example, the language-ext project has over 2600 unit tests. ReduSharptor could take up to days, or even weeks, to simplify all of these tests. Additionally, if this was introduced as a step in a pipeline, then it would be expensive to utilize. However, if performance increases were found and implemented, then ReduSharptor would be an even more efficient and effective tool.

7.2 Conclusion

After running through these tests and conducting experiments on 30 synthetic bugs, ReduSharptor performed well with great results. In was able to parse nearly all of the unit tests correctly and even a majority of the tests were simplified perfectly. Only a handful of these unit tests had necessary statements removed. However, from an initial perspective, implementing another HDD approach alongside this DD approach seems to be the solution for most of these issues. Additionally, it seems current coding standards are a great contributor to the effectiveness of this simple tool. Since C# coding standards lead developers to write simple unit tests with a flat structure, it allows for the fast, yet simple, DD algorithm to be useful.

Appendix A

Methods

```
/// <summary>
/// Build and run the test. Return the result
/// </summary>
/// <param name="testStatements">Test statements to test if successful</param>
/// <returns>True if the test is successful. False if unsuccessful</returns>
static public bool BuildAndRunTest(List<StatementSyntax> testStatements)
{
  // Write out statements to file
  Extentions.SetTestStatements(testExample, testExample, testName, testStatements);

  Console.WriteLine("Building current version of test.");

  // Run the build command
  if (!Extentions.ExecuteCommand("dotnet", "build \"" + testProj + "\""))
  {
    Console.WriteLine("Build failed. Continue searching for failing test.");

    // We don't want to record build failures, so we return true to not remember them in the algorithm
    return true;
  }

  Console.WriteLine("Running test for failure...");

  bool isSuccessful = Extentions.ExecuteCommand("dotnet", "test \"" + testProj + "\" --filter \"FullyQualifiedName=" + Extentions.GetTestCallString(testExample, testName) + "\"");

  if (isSuccessful)
  {
    Console.WriteLine("Test was successful. Continue looking for failing test.");
  }
  else
  {
    Console.WriteLine("Test was unsuccessful. Shrink test statements.");
  }

  return isSuccessful;
}
```

Figure A.1: BuildAndRunTest method in ReduSharptor

```
/// <summary>
/// Finds the smallest input for the test and input provided for the test to continue
    to fail
/// </summary>
/// <typeparam name="T">Type of the list in the input</typeparam>
/// <param name="array">Input array for the failing test</param>
/// <param name="compareTestInput">Function to compare the test input against</param>
/// <returns>A list of the smallest failing input for the test to continue to fail</
    returns>
static public List<T> FindSmallestFailingInput<T>(List<T> array, Func<List<T>, bool>
    compareTestInput)
{
  int numSections = 2;

  while (true)
  {
    bool isSuccessful = true;
    List<List<T>> sectionedArray = GetDividedSections(numSections, array);

    // Test the sections for failing input
    foreach (List<T> arrSection in sectionedArray)
    {
      isSuccessful = compareTestInput(arrSection);

      if (!isSuccessful && arrSection.Any())
      {
        // Section off failing input and try again
        array = arrSection;
        numSections = 2;
        break;
      }
    }

    if (!isSuccessful)  continue;

    // Test the compliments of the sections for failing input
    foreach (List<T> arrSection in sectionedArray)
    {
      List<T> compliment = GetSectionCompliment(sectionedArray, sectionedArray.IndexOf(
    arrSection));
      isSuccessful = compareTestInput(compliment);

      if (!isSuccessful && compliment.Any())
      {
        // Section off failing input and try again
        array = compliment;
        numSections = Math.Max(numSections - 1, 2);
        break;
      }
    }

    if (!isSuccessful)  continue;

    // If all previous inputs pass, increase granularity, create more equal parts
    numSections = 2 * numSections;

    if (numSections > array.Count)
    {
      return array;
    }
  }
}
```

Figure A.2: FindSmallestFailingInput method in ReduSharptor

```
/// <summary>
/// Shows a few examples about using the Adaptive Extention methods
/// </summary>
/// <param name="args">Arguments to control what to simplify; (Path to test, name of
    test, path to testProj, output path)</param>
static void Main(string[] args)
{
  if (args.Length != 4)
  {
    Console.WriteLine("Incorrect arguments\n");
    return;
  } else {
    Console.WriteLine("Using command line arguments");
    testExample = Path.GetFullPath(args[0]);
    testName = args[1];
    testProj = Path.GetFullPath(args[2]);
    outputFilePath = Path.GetFullPath(args[3]);
  }

  SyntaxList<StatementSyntax> testStatementsRaw = Extentions.GetTestStatements(
    testExample, testName);

  List<StatementSyntax> testStatements = new List<StatementSyntax>(testStatementsRaw);
  Func<List<StatementSyntax>, bool> buildAndCompareTest = BuildAndRunTest;

  Extentions.SetTestStatements(testExample, Path.Combine(outputFilePath, "Original",
    testName + "_" + Path.GetFileName(testExample)), testName, testStatements);

  List<StatementSyntax> simplifiedStatements = new List<StatementSyntax>();

  try
  {
    // Run algorithm with parameters
    simplifiedStatements = Extentions.FindSmallestFailingInput<StatementSyntax>(
    testStatements, buildAndCompareTest);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
  finally
  {
    // Revert the original test file back to the original form
    Extentions.SetTestStatements(testExample, testExample, testName, testStatements);
    Console.WriteLine("Reverting the original file.\nHere is the original file");
  }

  Extentions.SetTestStatements(testExample, Path.Combine(outputFilePath, "Simplified",
    testName + "_" + Path.GetFileName(testExample)), testName, simplifiedStatements);
  Console.WriteLine("Here are the simpified results.");
}
```

Figure A.3: Main method in ReduSharptor

```
/// <summary>
/// Divides the array into equal parts
/// </summary>
/// <typeparam name="T">Type of the list to be split</typeparam>
/// <param name="sizeOfArrays">Size of the parts to be split into</param>
/// <param name="array">Array to be split</param>
/// <returns>A list of equal parts of the array</returns>
static private List<List<T>> GetDividedSections<T>(int numSections, List<T> array)
{
  List<List<T>> result = new List<List<T>>();

  // Add all sub lists in list array
  for (int i = 0; i < numSections; i++)
  {
    result.Add(new List<T>());
  }

  int split = array.Count / numSections;
  int innerList = 0;

  if (split == 0)
  {
    return result;
  }

  for (int i = 0; i < array.Count; i += split)
  {
    for (int j = i; j < array.Count && j < i + split; j++)
    {
      if (innerList >= numSections)
      {
        innerList--;
      }
      result[innerList].Add(array[j]);
    }

    innerList++;
  }

  return result;
}
```

Figure A.4: GetDividedSections method in ReduSharptor

```
/// <summary>
/// Gets the compliment of the section provided
/// </summary>
/// <typeparam name="T">Type of the list to get the compliment of</typeparam>
/// <param name="array">Array to get compliment from</param>
/// <param name="sectionIndex">Index of the section to get the compliment of</param>
/// <returns>Compliment of the section index provided</returns>
static private List<T> GetSectionCompliment<T>(List<List<T>> array, int sectionIndex)
{
  List<T> compliment = new List<T>();

  foreach (List<T> section in array)
  {
    if (array.IndexOf(section) == sectionIndex)
    {
      continue;
    }

    compliment.AddRange(section);
  }

  return compliment;
}
```

Figure A.5: GetSectionCompliment method in ReduSharptor

```
/// <summary>
/// Gets the statement list for the test file provided
/// </summary>
/// <param name="testFilePath">Test file path for the statement list</param>
/// <returns>Statement list for the test provided</returns>
static public SyntaxList<StatementSyntax> GetTestStatements(string testFilePath, string
     testName)
{
  string text = File.ReadAllText(testFilePath);
  SyntaxTree tree = CSharpSyntaxTree.ParseText(text);

  CompilationUnitSyntax input = tree.GetCompilationUnitRoot();
  var nameSpaceOriginal = ((NamespaceDeclarationSyntax)input.Members[0]);
  var classOriginal = (ClassDeclarationSyntax)nameSpaceOriginal.Members[0];

  var classMembers = classOriginal.DescendantNodes().OfType<MemberDeclarationSyntax>();
  MethodDeclarationSyntax method = null;

  foreach (var member in classMembers)
  {
    var potentialMethod = member as MethodDeclarationSyntax;
    if (potentialMethod != null)
    {
      if (potentialMethod.Identifier.ToString() == testName)
      {
        method = potentialMethod;
      }
    }
  }

  var blockX = (BlockSyntax)method?.Body;

  return blockX.Statements;
}
```

Figure A.6: GetTestStatements method in ReduSharptor

```
/// <summary>
/// Gets the statement list for the test file provided
/// </summary>
/// <param name="testFilePath">Test file path for the statement list</param>
/// <returns>Statement list for the test provided</returns>
static public bool SetTestStatements(string testFilePath, string outputFilePath, string
    testName, List<StatementSyntax> statementsToReplace)
{
  if (!File.Exists(outputFilePath)) {
    try {
      if (!Directory.Exists(Path.GetDirectoryName(outputFilePath))) {
        Directory.CreateDirectory(Path.GetDirectoryName(outputFilePath));
      }

      FileStream file = File.Create(outputFilePath);
      file.Close();
    }
    catch (Exception ex)
    {
      return false;
    }
  }

  string text = File.ReadAllText(testFilePath);
  SyntaxTree tree = CSharpSyntaxTree.ParseText(text);

  CompilationUnitSyntax input = tree.GetCompilationUnitRoot();
  var nameSpaceOriginal = ((NamespaceDeclarationSyntax)input.Members[0]);
  var classOriginal = (ClassDeclarationSyntax)nameSpaceOriginal.Members[0];

  MethodDeclarationSyntax methodSyntax = null;
  var classMembers = classOriginal.DescendantNodes().OfType<MemberDeclarationSyntax>();

  // Search the members looking for a method with the same name as the test
  foreach (var member in classMembers)
  {
    var method = member as MethodDeclarationSyntax;
    if (method != null)
    {
      if (method.Identifier.ToString() == testName)
      {
        methodSyntax = method;
        break;
      }
    }
  }

  if (methodSyntax == null) return false;

  var blockX = (BlockSyntax)methodSyntax.Body;
  var statements = blockX.RemoveNodes(blockX.Statements, SyntaxRemoveOptions.
    KeepNoTrivia);
  var x = statements.AddStatements(statementsToReplace.ToArray());

  MethodDeclarationSyntax tempMethod = methodSyntax.WithBody(x);
  var newClass = classOriginal.ReplaceNode(methodSyntax, tempMethod);
  var output = input.ReplaceNode(classOriginal, newClass);

  WaitForFile(outputFilePath);
  File.WriteAllText(outputFilePath, output.ToString());
  return true;
}
```

Figure A.7: SetTestStatements method in ReduSharptor

```
/// <summary>
/// Blocks until the file is not locked any more.
/// </summary>
/// <param name="fullPath"></param>
public static bool WaitForFile(string fullPath)
{
  int numTries = 0;
  while (true)
  {
    ++numTries;
    try
    {
      // Attempt to open the file exclusively.
      using (FileStream fs = new FileStream(fullPath,  FileMode.Open, FileAccess.
    ReadWrite, FileShare.None, 100))
      {
        fs.ReadByte();

        // If we got this far the file is ready
        break;
      }
    }
    catch (Exception ex)
    {
      Thread.Sleep(500);
      if (numTries > 100)
      {
        return false;
      }

      // Wait for the lock to be released
      System.Threading.Thread.Sleep(500);
    }
  }

  return true;
}
```

Figure A.8: WaitForFile method in ReduSharptor

```
/// <summary>
/// Gets the string to only build the one test instead of the entire project.
/// </summary>
/// <param name="testFilePath">Path to the test file</param>
/// <param name="testName">Name of the test</param>
/// <returns></returns>
static public string GetTestCallString(string testFilePath, string testName)
{
  string text = File.ReadAllText(testFilePath);
  SyntaxTree tree = CSharpSyntaxTree.ParseText(text);

  CompilationUnitSyntax input = tree.GetCompilationUnitRoot();
  var nameSpaceOriginal = ((NamespaceDeclarationSyntax)input.Members[0]);
  var classOriginal = (ClassDeclarationSyntax)nameSpaceOriginal.Members[0];

  return nameSpaceOriginal.Name + "." + classOriginal.Identifier + "." + testName;
}
```

Figure A.9: GetTestCallString method in ReduSharptor

```
/// <summary>
/// Runs a cmd command from another process
/// </summary>
/// <param name="fileName">Command to run</param>
/// <param name="arguments">Arguments to run with the command</param>
/// <returns>True if successful; False if unsucessful</returns>
static public bool ExecuteCommand(string fileName, string arguments, int timeout =
    5000)
{
  try
  {
    ProcessStartInfo processInfo;
    Process process;

    processInfo = new ProcessStartInfo(fileName, arguments);
    //processInfo.CreateNoWindow = false;
    //processInfo.UseShellExecute = false;
    processInfo.RedirectStandardOutput = true;

    process = new Process();
    process.StartInfo = processInfo;

    process.Start();

    process.WaitForExit(timeout);
    string output = process.StandardOutput.ReadToEnd();
    return process.ExitCode == 0;
  }
  catch (Exception ex)
  {
    return false;
  }
}
```

Figure A.10: ExecuteCommand method in ReduSharptor

References

[1] D. Vince, R. Hodován, and Á. Kiss, "Reduction-assisted fault localization: Don't throw away the by-products!" in *ICSOFT*, 2021, pp. 196–206.

[2] A. Christi, M. L. Olson, M. A. Alipour, and A. Groce, "Reduce before you localize: Delta-debugging and spectrum-based fault localization," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Memphis, TN, USA, October 15-18, 2018*, 2018, pp. 184–191.

[3] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[4] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, 2006, pp. 142–151.

[5] R. Hodován and A. Kiss, "Modernizing hierarchical delta debugging," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, ser. A-TEST 2016. ACM, 2016, pp. 31–37.

[6] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proceedings of the 40th International Conference on Software Engineering*. Association for Computing Machinery, 2018, p. 361–371.

[7] R. Gopinath, A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller, "Abstracting failure-inducing inputs," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 237–248.

[8] D. Stepanov, M. Akhin, and M. Belyaev, "Reduktor: How we stopped worrying about bugs in kotlin compiler," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 317–326.

[9] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "Orbs: Language-independent program slicing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 109–120.

[10] J. Regehr, E. Eide, and Y. Chen, May 2019. [Online]. Available: https://embed.cs.utah.edu/creduce/

[11] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree-structured test inputs," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017, 2017, pp. 861–871.

[12] A. Christi, A. Groce, and R. Gopinath, "Resource adaptation via test-based software minimization," in *2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, Sept 2017, pp. 61–70.

[13] B. Wagner, "The .net compiler platform sdk (roslyn apis)," Sep 2021. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/

[14] D. Weber, "Redusharptor," Nov 2022. [Online]. Available: https://github.com/TheWebRage

[15] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[16] "Program repair benchmark bugs list," https://program-repair.org/benchmarks.html, accessed: 2022-11-17.

[17] P. Louth, "Louthy/language-ext: C# functional language extensions - a base class library for functional programming." [Online]. Available: https://github.com/louthy/language-ext

[18] S. Deminick, "Umbraco/umbraco-cms: The simple, flexible and friendly asp.net cms used by more than 730.000 websites." [Online]. Available: https://github.com/umbraco/Umbraco-CMS

[19] J. Staten, "Statianzo/fleck: C# websocket implementation." [Online]. Available: https://github.com/statianzo/Fleck

[20] Adelikat, "Tasemulators/bizhawk: Bizhawk is a multi-system emulator written in c#. bizhawk provides nice features for casual gamers such as full screen, and joypad support in addition to full rerecording and debugging tools for all system cores." [Online]. Available: https://github.com/TASEmulators/BizHawk

[21] Skclusive, "Skclusive/skclusive.mobx.observable: Mobx port of c# language for blazor." [Online]. Available: https://github.com/skclusive/Skclusive.Mobx.Observable