On Reducibility of Developer-Written Unit Tests in C#

Arpit Christi
School of Computing
Weber State University
Ogden, UT, USA
email: arpitchristi@weber.edu

David Weber
Northrop Grumman
Roy, UT, USA
email: ein.nuff@gmail.com

***Abstract*—Test case reduction is employed to help developer isolate and locate faults in complex software systems if the failing test is complex and contains a lot of non failure-inducing elements. Reduced test still contains the exact same failure-inducing component as the original test. Thus, the smaller test assists developers by focusing their attention on the faulty aspect of the program quickly. Researchers have focused their attention on improvement of the test reduction process. The outcome of the test reduction is not studied thoroughly. We study the result of the test case reduction when algorithms like Delta Debugging are used to minimize the tests. We evaluate (1) test reduction size based on the category of a statement and (2) effect of the category of a statement on reduction. The developer-written tests are just like any other code - containing the same structures, elements, and components as the rest of the program. If we consider the program as an abstract syntax tree, our results demonstrate that (1) leaf nodes are removed in larger quantity and (2) leaf nodes have higher probability of removal.**

Keywords-program debugging; software testing; test case reduction.

I. INTRODUCTION

Program debugging is often tedious, time-consuming and challenging. Most of the developer time is spent on locating and isolating the fault. When utilizing a failed test to debug and fix a fault, developers may need to observe and transit through aspects of test/program that are non failure-inducing, resulting into developer time disuse. If the failed test is minimized while keeping the failure-inducing input, the developer may need to rummage through lesser program elements promoting optimal use of developer time and hence quicker debugging. Being orthogonal to aiding developer in debugging, test reduction is found useful in Automatic Fault Localization (FL) - a process to automatically locate the bug in a faulty program. The entire reduced test or the byproducts of test reduction are found to be useful in Automatic FL [1] [2].

Delta Debugging (DD) and Hierarchical Delta Debugging algorithms (HDD) were proposed to minimize failure-inducing tests [3] [4]. DD algorithm is optimal for test inputs that are flat structures like array, lists or sets. If you consider a test written in program like C#, in order to apply DD, one needs to consider test to be an array of lines, an array of characters or words. As it does not consider the tree like structure, interdependence between nodes and other such details, DD is not optimal for structures like HTML files and programs. HDD can process such tests better as it exploits the underlying tree structures to its advantage. Both algorithms are essentially a greedy search to systematically and incrementally find a smaller test until a minimal test is reached.

Many recent algorithms and implementations to minimize failing tests still rely on the DD and HDD algorithms as the foundation [5]–[11]. These tools and techniques mainly attempt to improve the test reduction process to efficiently and accurately reduce tests. Though test reduction process have been studied for a while, (1) the outcome of the test reduction and (2) the entities that were reduced as part of the reduction process have not been studied thoroughly.

Based on the type of test (program, html, xml, text files etc.), the reduction outcome and the reduced entities can be different. If we only consider tests written as a program in a particular programming language, we can define and study the reduction outcome and the reduced entities by considering the outcome as a reduced program and the entities as programming components like - program statements, program lines, nodes of Abstract Syntax Tree (AST) of the program. For this work, we consider reduced entities as nodes of AST. An example of the test is in Figure 1 and the corresponding AST is in Figure 4. We further categorize each statement node of the AST into non-tree statements and tree statements as explained in detail in Section IV.

We focus on test reduction for tests written in C# programming language. We study the reduction outcome and the reduced entities for 30 real world bugs in 5 open source C# projects. Based on our study, we provide the following insights into test reduction for C# tests.

1) The number of non-tree statements reduced are significantly larger than the number of tree statements reduced.
2) The chance of a non-tree statement removal is slightly higher than the chance of a tree statement removal.

The ReduSharptor tool that we used for test reduction is publicly available on GitHub [12].

The rest of the paper is organized as follows. In Section II, we discuss the related work. In section III, we discuss the background for our work and motivate the need for the study. In Section IV, we discuss the terminology and definitions based on the outcome of the reduction process to determine catogory of program statements. Section V depicts the test subjects, the experiments and the results. We mention how we mitigate the threats to validity in Section VI. Finally, Section VII concludes the paper by discussing the results and the future direction.

II. Related Work

DD finds minimal failure inducing input by employing a greedy search that removes components that are unnecessary for triggering the bug [3]. HDD improves on DD for hierarchical inputs like xml, html, and programs. HDD achieves the improvement by considering the AST representation of hierarchical inputs [4].

Researchers proposed many recent algorithms, tools and techniques. They (1) improve over the original DD and HDD algorithms or (2) retrofit DD/HDD implementation for specific situation or programming languages. CReduce, Generalized Tree Reduction (GTR), Picireny, Perses, DDSET, Observational Based Slicing (ORBS), Reduktor, ProbDD, and ReduSharptor are a few such attempts [5]–[11], [13]–[15].

Christi et al. combined HDD with statement deletion mutation to propose Test-Based Software Minimization (TBSM) that reduces programs instead of tests to build a minimal resource adaptive software while sacrificing low-priority but resource-consuming functionality [16]. To improve the performance of TBSM, they study the outcome of the program reduction and the entities that were reduced. Based on that, they proposed multiple heuristics to improve the performance of TBMS [17].

Perses reduces the removal entities by only considering syntactically valid variants [6]. Wang et al. propose probabilistic delta debugging approach that uses AST, historic test results and syntactic relationships to assign probability to each element for removal [10] [11]. The approach showed significant improvement in performance because it reduces the number of entities under consideration for removal. We study the reduction process only in terms of the location of an entity within the program. Also, we use a different programming language and a different dataset to study the reduction process.

III. Background and Motivation

If we can categorize test program statements into distinct categories and establish an empirical link between the category of a statement and its probability of removal, we can preemptively choose to process or ignore certain types of statements in the test reduction process. To this end, the outcome of test reduction and reduced entities need to be studied further. Studying the relationship in detail can help to propose efficient approaches like perses and probabilistic delta debugging [6], [11]. It may help propose heuristics as proposed by Christi et al. to reduce the search space for the reduction [17]. So far, such categorization is not clearly established.

We will only consider tests written in C# programming language. When DD, HDD or any other techniques are applied on a test for test reduction, it produces a minimal test.

The test can be reduced at a different granularity. For example, a test can be considered as a series of characters and one or more characters can be reduced at a time to produce minimal test. If we consider HDD, reduction granularity can be a node of the AST of the program. Each test method is composed of program statements that are defined as *StatementNode*. In C# the *StatementNode* is implemented by Roslyn compiler as *StatementSytax* class or any other *StatementNode* that is derived from *StatementSyntax* class [18]. The statement node can be further decomposed for processing. Multiple previous works suggest that reduction is useful and meaningful at a statement level [9] [16]. Hence, we consider program statement or *StatementNode* as a unit of reduction. Consider a simple test as shown in Figure 1. The corresponding AST is shown in Figure 4. The dotted lines mean that the nodes can be further decomposed into non-statement nodes. But we avoid such decomposition to only consider statement level nodes. The reduced test is shown in Figure 2 and the corresponding AST is shown in Figure 5

```
[Fact]
public void Foo(Test)
{
1  Math m = new Math();
2  int sum1 = m.Add(3,4)
   // Assumption: Add method is written in a
   // peculiar way and cannot add 3 and 4
   // correctly.
3  Assert.Equal(sum1,7); //suppose sum1 is 8,
    hence the test is failing here.
4   if(true){
5       int sum2 = m.Add(-2,-3)
6      Assert.Equal(sum2,-5); // This assert
    passes.
7     }
}
```

Figure 1. `original` test, Line 3 is the failing statement.

```
[Fact]
public void Foo(Test) //The minimal reduced
    test
{
1  Math m = new Math();
2  int sum1 = m.Add(3,4)
   // Same assumption as the original test.
3  Assert.Equal(sum1,7); //suppose sum1 is 8,
    hence the test is failing here.

}
```

Figure 2. `minimal` test, All statements from line 4 in Figure 1 are removed.

When we compare the ASTs in Figure 4 and Figure 5, we note that two leaf statements are reduced, and one non-leaf statement is reduced (the if statement). The reduction in terms of program components is shown in Figiure 3. We want to experiment with real-world tests to study the reduction outcome and the reduced entities to establish categorization of statements and the probability of removal for each category.

IV. Terminology and Definitions Based on Reduction Process and the Outcome

We use the following terminology and definitions for the rest of the discussion.

```
if(true){
5       int sum2 = m.Add(-2,-3)
6      Assert.Equal(sum2,-5); // This assert
    passes.
7     }
```

Figure 3. `reduced-statements`. Line 4-7 in test in Figure 1 consist of reduced statements

If the test is in a programming language like C#, we can define the following.

1) *original-test*: The test without any reduction. We consider the test to be an AST with a set of *StatementNode*s. Figure 1 shows the original test and Figure 4 shows the corresponding AST.
2) *minimal-test*: The remaining test after a test is reduced. If the *original-test* is reducible, *minimal-test* has one or more statements removed. In Figure 2, the statements from line 4 onwards in *original-test* are removed. The AST in Figure 5 depicts the AST for *minimal-test*.
3) *reduced-entities*: The program statements that are reduced during the reduction process. If original test was T and the minimal test was T' then *reduced-entities* are T - T'. *Reduced-entities* are a set of *StatementNode* or statements in our case. The *reduced-entities* are showin in Figure 3. If you compare Figure 4 and Figure 5, the *reducued-entities* are the removed subtree.
4) *TreeNode*: A *TreeNode* is a *StatementNode* that has at least one subtree that consists of one or more *StatementNode*. For example, in Figure 4, *IfStmt* is the *TreeNode*, because it contains two statements that are *TreeNode*. (1) *int sum2 = m.Add(-2,-3)* and (2) *Assert.Equal(sum2,-5)*. (Note: we are ignoring *BlockStmt*, that is explained later). In our experiment subjects, we found conditional statements, loop statements and action statements as the majority tree nodes. As we only consider statement nodes, *Treenode* can also be referred as *TreeStmt*.
5) *NonTreeNode*: A *NonTreeNode* is a *StatementNode* that does not have any subtree that consists of *StatementNode*. In Figure 4 *Math m = new Math(), int sum1 = m.Add(3,4)* etc. are *NonTreeNode*. *NonTreeNode* can also be referred as *NonTreeStmt*.

Each *BlockStmt* consists of one or more *StatementNode*s. Removing the *BlockStmt* can disturb the tree structure such that Roslyn compiler may not create syntactially correct variants [18]. Hence, *BlockStmt* is never considered during reduction. Only the statements that are below *BlockStmt* are considered for reduction as it was done with *ReduSharptor*.

We categorize the statements of a C# tests into the categories based on its location in the program: *TreeStmt* and *NonTreeStmt*. We wan to study how the quantity of removal and the probability of removal are dependent on this categorization.

V. Experiments

We want to study both *minimal-test* and *reduced-entities* to understand the effect of statement category on removal process and reduction outcome.

For that we use the same subjects and procedure used in the previous research by Weber et al [9] to evaluate a test-reduction tool *ReduSharptor*.

A. Subjects

Weber et al. used 30 real-world failing tests across five open source C# projects as the subjects. Four out of these five open source projects are under active development. Each subject was selected such that it has one or more *reduced-entities*. If the *original-test* is already minimal and cannot be reduced any further, *original-test* and *minimal-test* are the same. Hence, comparison and further evaluations are not meaningful. Accuracy of our analysis depends on the accuracy of *ReduSharptor* - *ReduSharptor* has high precision (96.58%) and high recall (96.45%). The projects that were used as subjects are enumerated in in Table 1 in the work of Weber et al. [9].

B. Process and Measurement

We apply *ReduSharptor* on each failing test, reduced the failing test and generated failure-inducing *minimal-test*. We compare failing *original-test* with the failing *minimal-test*. We collect the following information.

1) *absolute-reduction-size* (ARS): The number of statements that are reduced as part of reduction process. This is essentially the size of *reduced-entities* in terms of statements. In the example in section IV, *absolute-reduction-size* is three statements - the IfStmt and two other statements at the leaf of the IfStmt subtree.
2) *percentage-reduction-size* (PRS): The percentage of total statements reduced. In the example, the percentage is 50% - total statements are 6 and reduced statements are 3.
3) *absolutte-TreeStmt-reduction-size* (ATRS): The number of *TreeNode*s reduced. In the example, 1 *TreeNode* is reduced - the *ItStmt*.
4) *percentage-TreeStmt-reduction-size* (PTRS): The percentage of *TreeNode*s reduced. In the example, the PRTS is 16.67%.
5) *absolutte-NonTreeStmt-reduction-size* (ANTRS): The number of *NonTreeNode*s reduced. In the example, 2 *NonTreeNode*s are reduced.
6) *percentage-NonTreeStmt-reduction-size* (PNTRS): The percentage of *NonTreeNode*s reduced. In the example, PNTRS is 33.33%.

C. Results

Across 30 failing tests, we process 759 total statements. The results of reductions are shown in Table I. We show total number of statements for each test, the number of *NonTreeStmt*s, the number of *TreeStmtss*, ARS, PRS, ANTRS, PNTRS, ATRS, and PRS. On average, we processed 25.3

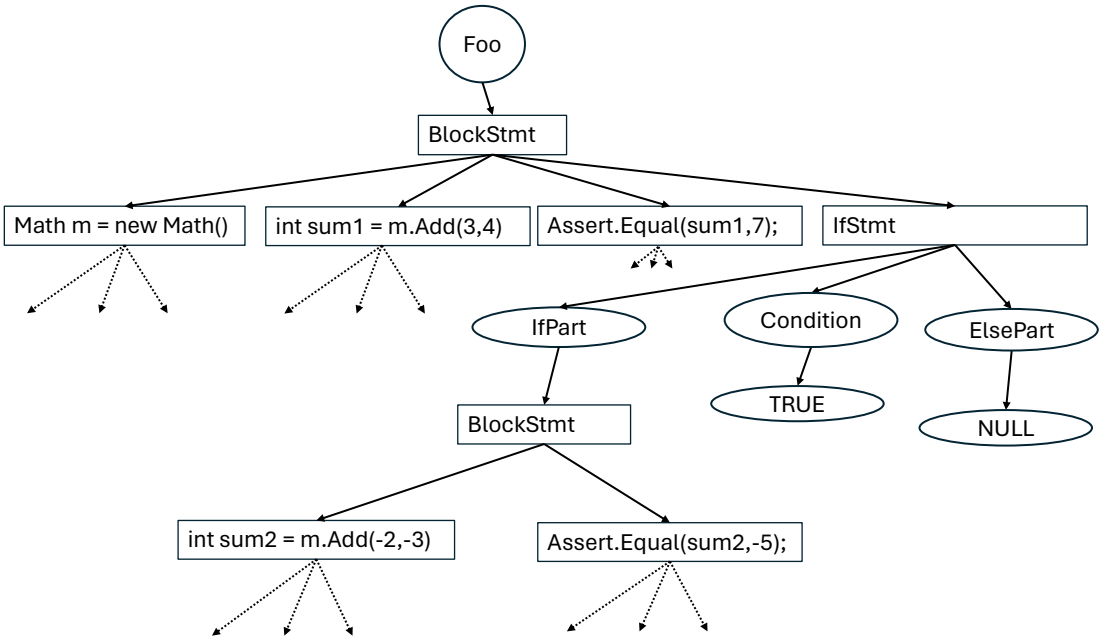

Figure 4. AST of code in Figure 1

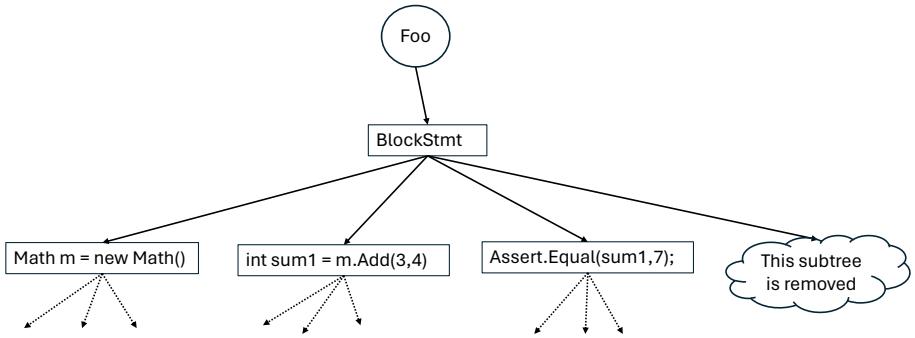

Figure 5. AST of code in Figure 2

statements per test that include 24.4 *NonTreeStmt*s and 0.9 *TreeStmt*s. We reduced on average 19.93 statements or 71.87% per failing test. For *NonTreeStmt*s, the average reduction was 19.56 or 70.44%. The same numbers for *TreeStmt*s are 0.433 and 1.43%.

The most important entities are the PRS, PNTRS and PTRS. Consider two tests - one contains 100 statements and another 10 statements. If the 20 statements are reduced in the first test and 4 statements are reduced in the the second test, the ARS, ANTRS, and ATRS numbers can be misleading. For the first test 20% statements are reduced and for the second test 40% statements are reduced. Reduction is significant for the second test.

More than half of the PTRS values are 0 and the data is not normally distributed violating the t-test assumptions. We confirm this using Shapiro-Wilk test for normality [19]. Hence, we perform paired Wilcoxon signed rank test on PNTRS and PTRS that has $V = 465$ and p-value = $1.825e-06$ ($p << 0.05$) [20]. Wilcoxon test suggests significant difference between PNTRS and PTRS. To exactly understand the difference, we draw PNTRS vs PTRS boxplot in Figure 6. We can conclude that *NonTreeStmt*s are reduced in large numbers compared to *TreeStmt*s (approximately 50 times).

We also want to know the probability of removal of a randomly chosen statement based on its category - *TreeStmt* or *NonTreeStmt*. From the results above, we may think that if a randomly chosen statement is *NonTreeStmt*, it has more chances of removal. That may be misleading. Consider the *TestObserve* test in Table I. The test has three *TreeStmt*s and all of them are being removed resulting into 100% removal

of *TreeStmt*s. For the same test, out of 101 *NonTreeStmt*s 98 are removed resulting into 97% removal. For the test *TreeStmt*s have higher probability of removal. To consider this, we define two new terms: (1) ***PrNTRS*** - probability of removal of *NonTreeStmt*s defined as number of *NonTreeStmt*s removed over total *NonTreeStmts* in the test, $(ANTRS \div \#NTN) * 100$ as per Table I (2) ***PrTRS*** - probability of removal of *TreeStmt*s defined same as above but for *TreeStmt*s, $(ATRS \div \#TN) * 100$. For a certain rows in Table I, $\#TN$ is 0 and hence *PrTRS* is undefined (divide by 0). We cannot use such tests for evaluation. The results excluding the tests that have undefined *PrNTRS* are shown in Table II.

Both *PrNTRS* and *PrTRS* are not normally distributed (Shapiro-Wilk test). So, we use paired Wilcoxon signed rank test that has $V = 99$ and $p = 0.02877$ ($p < 0.05$). So, *PrNTRS* and *PrTRS* are different. To find the difference, we again draw the values using boxplot shown in figure 7. We conclude that the probability of removal of a non-tree statement is slightly higher (approximately 1.7 times) than that of a tree statement based on the boxplot.

D. Discussion on Possible Limitations

Our experiments and results depend on how the existing tests are written for open-source C# projects. We notice that the tree-statements per test is 0.9, which is very low. For a tree statement to be reduced, the test must contain at least one or more tree statements. The PTRS entity is dependent on the availability of tree statements.

If a failed assert is at the beginning or in the middle of a test, the entities after the failed assert will always be removed. If we move that failed assert even at the end of the test, those entities will still be removed. DD and HDD algorithms guarantee 1-minimality. 1-minimality ensures that the test reduction and our analysis are not dependent on the location of the failed assert.

VI. Threats to Validity

Now, we discuss threats to validity and the steps we take to mitigate them.

A. Construct Validity

Do our results truly compare non-tree statements and tree-statements for test reduction? The *ReduSharptor* tool used for experiments has high precision and high recall. It is easy to identify tree statements and non-tree statements in a test.

B. Internal Validity

Do we mitigate bias during experiments? All the projects and tests are part of open-source projects available online. Bugs were randomly sampled such that it has one or more *reduced-entities*.

C. External Validity

Do our results generalize? We only performed experiments on C# projects and tests. As tests in other programming languages have similar structures and program statement types, we expect the results to generalize.

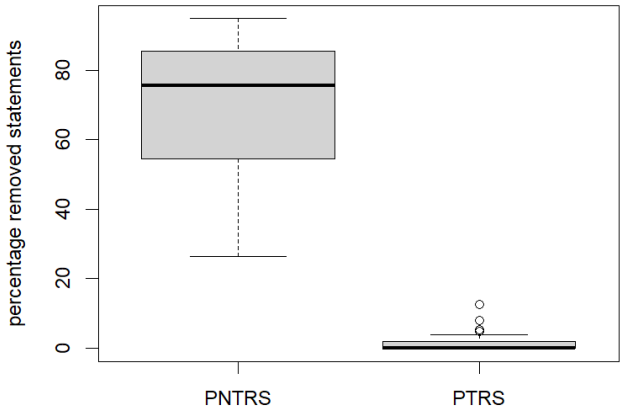

Figure 6. comparison between PNTRS vs PTRS

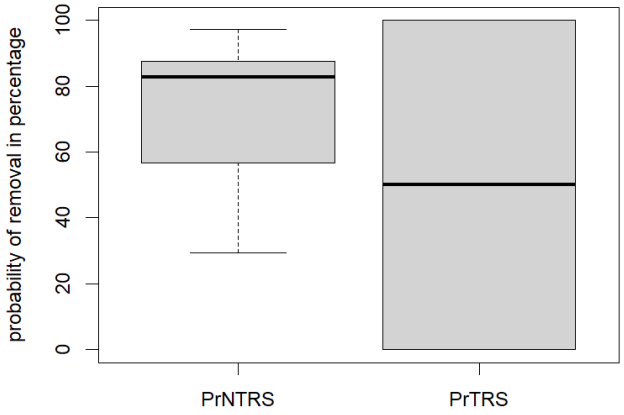

Figure 7. comparison between PrNTRS vs PrTRS

VII. Conclusion and Future Work

Based on our results, our contribution is as follows. We also plan to extend our work to further investigate the relationship between test reduction outcome and possible categorization of statements.

A. Contribution

Very few DD and HDD implementations assign priority to an entity based on the category of the entity. DD and HDD work on wide range of inputs and hence defining a generic category is difficult. If we only consider tests written as programs, we can come up with a broad generic category. Based on the categorization, we study the effect of a statement

TABLE I
TEST, PROJECT, TOTAL STMTS, *#NTN* - NUMBER OF *NonTreeNodes*, *#TN* - NUMBER OF *TreeNodes*, ARS, PRS, ANTRS, PNTRS, ATRS, PTRS

Test	Project	Stmts	*#NTN*	*#TN*	ARS	PRS	ANTRS	PNTRS	ATRS	PTRS
ListCombineTest	language-ext	10	10	0	6	60.00%	6	60.00%	0	0%
EqualsTest	language-ext	7	7	0	6	85.71%	6	85.71%	0	0%
ReverseListTest3	language-ext	5	5	0	2	40.00%	2	40.00%	0	0%
WriterTest	language-ext	17	15	2	8	47.06%	8	47.06%	0	0%
Existential	language-ext	14	14	0	11	78.57%	11	78.57%	0	0%
TestMore	language-ext	55	55	0	47	85.45%	47	85.45%	0	0%
CreatedBranchIsOk	Umbrraco-C..	54	54	0	39	72%	39	72%	0	0%
CanCheckIfUserHasAccessToLanguage	Umbrraco-C..	19	17	2	6	31.58%	5	26.32%	1	5.26%
Can _Unpublish_ContentVariation	Umbrraco-C..	28	28	0	25	89.29%	25	89.29%	0	0%
EnumMap	Umbrraco-C..	11	11	0	6	54.55%	6	54.55%	0	0%
InheritedMap	Umbrraco-C..	17	17	0	11	64.71%	11	64.71%	0	0%
Get_All_Blueprints	Umbrraco-C..	25	23	2	22	88.00%	20	80.00%	2	8.00%
ShouldStart	Fleck	7	5	2	3	42.86%	3	42.86%	0	0%
ShouldSupportDualStackListenWhenServerV..	Fleck	4	3	1	3	75.00%	3	75.00%	0	0%
ShouldRespondToCompleteRequestCorrectly	Fleck	15	15	0	11	73.33%	11	73.33%	0	0%
ConcurrentBeginWrites	Fleck	21	21	0	16	76.19%	16	76.19%	0	0%
ConcurrentBeginWritesFirstEndWriteFails	Fleck	27	26	1	22	81.48%	21	77.78%	1	3.70%
HeadersShouldBeCaseInsensitive	Fleck	7	7	0	5	71.43%	5	71.43%	0	0%
TestNullability	BizHawk	15	15	0	13	86.67%	13	86.67%	0	0%
TestCheatcodeParsing	BizHawk	8	7	1	7	87.50%	6	75.00%	1	12.50%
SaveCreateBufferRoundTrip	BizHawk	31	29	2	24	77.42%	24	77.42%	0	0%
TestCRC32Stability	BizHawk	27	25	2	13	48.15%	13	48.15%	0	0%
TestSHA1LessSimple	BizHawk	14	14	0	7	50.00%	7	50.00%	0	0%
TestRemovePrefix	BizHawk	14	14	0	13	92.86%	13	92.86%	0	0%
TestActionModificationPickup1	Skclusive.Mob..	23	21	2	9	39.13%	9	39.13%	0	0%
TestObservableAutoRun	Skclusive.Mob..	26	25	1	23	88.46%	22	84.62%	1	3.85%
TestMapCrud	Skclusive.Mob..	39	38	1	37	94.87%	37	94.87%	0	0%
TestObserver	Skclusive.Mob..	104	101	3	101	97.12%	98	94.23%	3	2.88%
TestObserveValue	Skclusive.Mob..	62	59	3	58	93.55%	56	88.71%	3	4.84%
TestTypeDefProxy	Skclusive.Mob..	53	51	2	44	83.02%	43	81.13%	1	1.89%
Mean		**25.3**	**24.4**	**0.9**	**19.93**	**71.87%**	**19.56**	**70.44%**	**0.433**	**1.43%**

TABLE II
TEST, *PrNTRS* VS *PrTRS* FOR INDIVIDUAL TEST. TESTS WITH UNDEFINED *PrTRS* ARE NOT INCLUDED.

Test	PrNTRS	PrTRS
WriterTest	53.33%	0.00%
CanCheckIfUserHasAccessToLanguage	19.41%	50%
Get_All_Blueprints	86.95%	100%
ShouldStart	60.00%	0.00%
ShouldSupportDualStackListenWhenServerV4All	75.00%	0.00%
ConcurrentBeginWritesFirstEndWriteFails	80.76%	100.00%
TestCheatcodeParsing	85.71%	50.00%
SaveCreateBufferRoundTrip	82.75%	0.00%
TestCRC32Stability	52.00%	0.00%
TestActionModificationPickup1	42.87%	0.00%
TestObservableAutoRun	88.00%	100.00%
TestMapCurd	97.36%	0.00%
TestObserver	97.02%	100.00%
TestObserveValue	93.22%	100.00%
TestTypeDefProxy	81.31%	50.00%

category on the reduction outcome and on the removal process. We conclude that the location of a program within the AST has an effect on the reduction outcome and the removal process. The non-tree statements (leaf nodes) will be removed in larger numbers and they will have slightly higher chance of removal.
.

B. Future Work

Our work focuses on C# tests. A very obvious extension would be to verify the results on tests written in other programming languages like Java, Python, and etc. We expect similar results for other programming languages also. Currently we categorize test statements into two categories based on the location within the AST. One area of extension would be to use other kind of categories. For example, types of the statements like declaration statement, method call statement, if statement, loop statement, try-catch statement etc. A broader extension would be to derive generic categories for non-program test inputs like html, xml, and text files.

REFERENCES

[1] 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.

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

[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 *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. Weber and A. Christi, "Redusharptor: A tool to simplify developer-written c# unit tests," *International Journal of Software Engineering & Applications*, vol. 14, pp. 29–40, 09 2023.

[10] G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang, "Probabilistic delta debugging," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 881–892.

[11] G. Wang *et al.*, "A probabilistic delta debugging approach for abstract syntax trees," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 763–773.

[12] "ReduSharptor Tool," https://github.com/amchristi/ReduSharptor, accessed: 2024-04-24.

[13] J. Regehr *et al.*, "Test-case reduction for c compiler bugs," in *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, pp. 335–346.

[14] 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.

[15] D. Binkley *et al.*, "Orbs: Language-independent program slicing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 109–120.

[16] 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)*. IEEE, 2017, pp. 61–70.

[17] A. Christi and A. Groce, "Target selection for test-based resource adaptation," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2018, pp. 458–469.

[18] "Roslyn Compiler Documentation," https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/, accessed: 2024-03-03.

[19] "Descriptive Statistics and Normality Tests for Statistical Data," https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6350423/, accessed: 2024-03-07.

[20] "Wilcoxon Signed Ranked Test," https://www.sciencedirect.com/topics/medicine-and-dentistry/wilcoxon-signed-ranks-test, accessed: 2024-03-07.