REDUSHARPTOR: A TOOL TO SIMPLIFY DEVELOPER-WRITTEN C# UNIT TESTS

David Weber and Arpit Christi

School of Computing, Weber State University, Ogden, UT, USA

ABSTRACT

Modern software systems are complex and locating, isolating and fixing a fault even with a failing test is tedious and time-consuming. Simplifying failing test(s) can significantly reduce the developer effort by reducing the irrelevant program entities that developers need to observe. Delta Debugging (DD) algorithm automatically reduces the failing tests. Hierarchical Delta Debugging (HDD) algorithm improves 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 complex structures, intricacies, and interdependence of program elements of a particular programming language. We propose a tool ReduSharptor to simplify C# tests that uses language-specific features and interdependence of C# program elements using Roslyn compiler APIs. We evaluate ReduSharptor on a set of 30 failing C# tests to demonstrate its applicability and accuracy.

KEYWORDS

program debugging, software testing, software maintenance.

1. INTRODUCTION

The complexity of modern software makes debugging difficult and time consuming even with an availability of failing test(s). To debug and fix 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 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 observe. It focuses developers' attention on faulty aspect of the program faster. Simplified failing tests are not only helpful aid to developers, but it can also significantly improve the accuracy of automatic fault localization techniques [1], [2].

The most widely known and utilized automatic test simplification technique is the Delta Debugging (DD) algorithm by Zeller and HildeBrandt that works well on test inputs that can be considered array or list like structures [3]. The DD algorithm is not most efficient on tests that are tree-like structures like HTML files, c or java programs, XML, etc. Mishreghi and Su proposed Hierarchical Delta Debugging (HDD) algorithm that works efficiently on tree like test inputs by utilizing the underlying Abstract Syntax Tree (AST) structure [4].

Recently a few researchers proposed modern implementations of HDD algorithm and their variants [5], [6], [7], [8]. Most of the implementations are language-agnostic and hence can reduce a variety of tests like HTML, source code in different programming languages, xml, and etc. Stepanov et al. noted the language-agnosticness of the HDD tools 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-dependence [8]. These tools rely on a generic AST or grammar in simplification process and produces many non-compilable intermediate variants before the convergence. Sun et al. noted the need of producing syntactically correct intermediate test variants while proposing 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 steps before the tool can be utilized to simplify tests [6], [5].

Instead of focusing on varying set of test inputs and test cases, we focus on developer-written C# unit tests. As we focus our attention, observe and study 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 novel features.

1. A tool specifically implemented for C# tests that utilizes language-specific features of C# programs and tests. In the process, it avoids many non-compilable variants.
2. A tool that utilizes multiple approaches to prune the search space.
3. A tool that exists as a stand-alone entity and does not require any further libraries and tool set. The tool can be invoked using an EXE.
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 high re call (96.45%). *ReduSharptor* is publicly available on GitHub (https://github.com/TheWebRage/ReduSharptor). The dataset that we use for evaluation is also available on GitHub (https://github.com/TheWebRage/CSharp-SyntheticBugs).

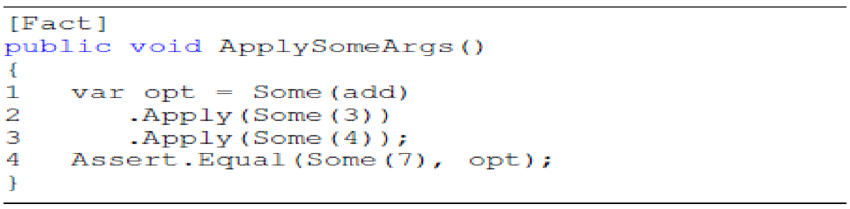
2. RELATED WORK

Delta Debugging (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 files, html, programs, etc. Misherghi and Su proposed HDD that works efficiently on tree like inputs by exploiting the underlying AST [4]. Both DD and HDD are theoretically sound algorithms that guarantee convergence and minimality.

Regehr et al. utilized test reduction to propose CReduce to minimize C programs for compiler testing [10]. Hodovan and Kiss observed that Extended Context Free Grammar produces better balanced tree than Context Free Grammar and utilized it in implementing modernized HDD tool called picireny [5]. Herfert et al. proposed Generalized Tree Reduction (GTR) algorithm that relies on (1) operations other than removal or deletion (2) replacing a tree node with similar tree node [11]. Sun et al. observed that during the simplification process, many previous algorithms produce syntactically invalid variants. A futile compilation step needs to be performed before pruning the invalid variant. They proposed Perses algorithm specifically to avoid generation of invalid variants [6]. Gopinath et al. utilized Perses algorithm to propose DDSET algorithm to abstract failure inducing minimal input from a larger input using input grammar [7]. Picireny, Perses and DDSET use language specific grammar of Antlr to produce the AST for specific programming languages. Binkley et al. proposed Observational-based Slicing (ORBS) technique that uses program line deletion as a fundamental operation to slice program accurately and efficiently [9]. Christi et al. combined inverted HDD with statement deletion mutation to simplify programs for the purpose of resource adaptations [12]. They argued that reduction is meaningful and useful at statement level and avoided non-statement level reductions [12,13].

3. MOTIVATION AND NEED FOR REDUSHARPTOR

Consider the test *ApplySomeArgs* taken from *language-ext* project, one of the projects in our empirical analysis. The test source code is shown in Figure 1.



```
[Fact]
public void ApplySomeArgs()
{
1   var opt = Some(add)
        .Apply(Some(3))
        .Apply(Some(4));
4   Assert.Equal(Some(7), opt);
}
```

Figure 1. *ApplySomeArgs* test in *language-ext*

3.1. PURPOSE OF THE MINIMIZED TEST

If test minimization is used for compiler testing, even a non-compilable 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 andexecutable tests. Any intermediate test that has compilation error will be pruned and will not be used for further processing by the simplification process because the tool cannot produce a pass/fail results 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, at least the test project, library or jar needs recompilation. For real-world test projects, the compilation time can be very high. For example, for the *language-ext* project, after a change is made in any of the tests, the compilation time is approximately 11 seconds on a windows machine with Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz processor and 16.0 GB RAM.

3.3. PERFORMANCE OF OTHER TECHNIQUES

If we simulate the behavior of ORBS or Perses on the test, we notice the potential of producing many variants that cannot compile.
ORBS relies on line level reduction and hence, for the test in Figure 1, it may produce variants where line 1 is removed from the test or line 3 is removed from the test; both are not compilable. Perses attempts to produce syntactically correct variants but syntactic correctness does not always result in successful compilation. For example, for the test in Figure 1, in line 2, *.Apply(Some())* and *.Apply()* are correct syntactic variants that don't compile. Perses will produce many such variants for the given test. If the minimized test is going to be used by developers, avoiding such variants can improve the performance significantly considering the large compilation time for most test projects.

3.4. Using Statement as the Unit of Reduction

Instead of using (1) any node in the AST or (2) line in the test as the basis of the reduction, *ReduSharptor* uses program statement as the unit of reduction. The idea of using statement level reduction is based on previous work where researchers found statement level reductions to be most useful and applicable [12,13]. The statement is defined by *StatementSyntax* class or other derived classes of the *StatementSyntax* in Roslyn compiler API [16]. With statement as unit of reduction; line 2, line 3, and line 4 will be treated as a single statement of type *LocalDeclarationStatementSyntax* by Roslyn Compiler. Hence, it can only produce one variant that cannot compile - the variant where the entire first statement is deleted.

3.5. Search Space Reduction Due to Less Intermediate Variants

When we use statement as a unit of reduction, we are essentially considering the AST with significantly less nodes as we just ignore the existence of nodes below the statement level. As the DD/HDD algorithm will have to process less nodes, many variants will be pruned automatically resulting in a considerable reduction in the search space. As DD algorithm is O(n^2) and HDD algorithm is O(n^3), any reduction in search space will result in significant performance improvement.

3.6. DD is Sufficient

Consider the fictitious test case shown in figure 2. The corresponding AST representation is available in Figure 3. The figure only shows statement nodes as we already argued for not using nodes below the statement level. Now consider two nodes that corresponds to line 1 and 2 of figure 4. Such statements don’t have a sub tree with our statement deletion assumption. The if statement spanned across line 3, 4 and 5 results into a tree. We divide Roslyn compiler statement set into two distinct sets - (1) NonTree - statements that cannot form further sub trees (2) Tree-statements that can form further sub trees. We conducted a formative study on 100 distinct developer-written randomly-chosen unit tests across 10 real-world projects and observed the statement usage. The possibility that a Tree statement exists immediately after the method- level block statement in AST is low in **developer-written C# unit tests** - only 3.30%. If a single Tree statement cannot immediately exist after the method-level block statement in AST, the test method cannot have any more Tree statements. So, we consider the Tree statement as a NonTree statement for processing purposes. We will process IfStmt as a single statement instead of processing the corresponding sub tree separately. For Figure 2 code, this means treating line 3, line 4, and line 5 as a single block. Either the entire block is removed or nothing is removed. We don’t have any chance to separately processing *Assert* in line 4. We have two advantages. (1) We need to process less statements (2) All statements below block statements are considered NonTree statement. We have a list or set of ―NonTree statements below *BlockStmt* and we can process them using DD algorithm (O(n^2)) instead of HDD algorithm (O(n^3)). This can be done because we abolished tree structures and converted into a flat non-tree structure. At first glance we seem to be sacrificing accuracy (in the foo example, probably missing line 4 Assert statement) for efficiency but later our results demonstrate that such simplification works well in practice.

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

Figure 2. *foo* test to demonstrate AST

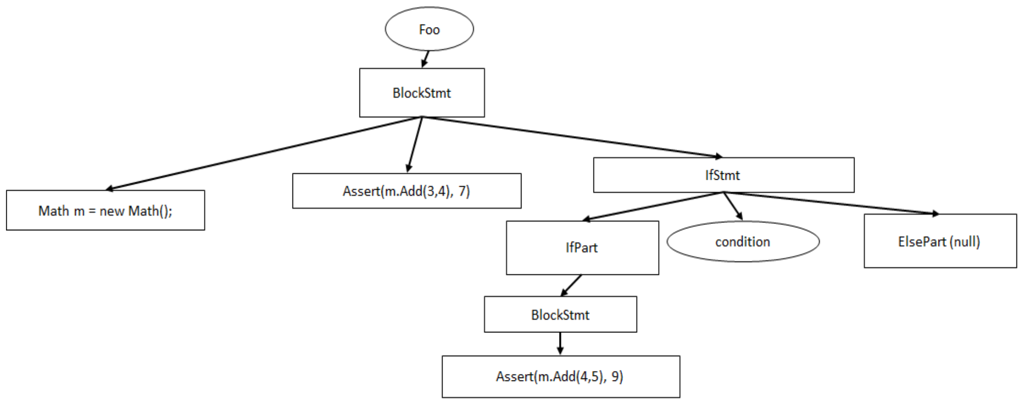


Figure 3. AST of code in Figure 2

4. REDUSHARPTOR: USAGE, ARCHITECTURE, AND IMPLEMENTATION DETAILS

4.1. USAGE

To use our tool, the developer will have to only provide the following: test file with full path, name of the test (as single file has many tests and we may want to reduce only one failing test), the path of the .csproj file associated with the code. Figure 4 shows a usage example of *ReduSharptor*. All 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 final output in that folder. In the example provided in Figure 4, the first argument is the test file with complete path, the second argument is the name of the test, the third argument is the csproj file with complete path and the last argument is the folder where intermediate results and the final results will be stored.

The architecture from a perspective of a user is described in Figure 5. If you compare the architecture figure with Perses workflow figure and picereny architecture figure in the corresponding works, the contrast is clear [5], [6]. Both Perses and picireny need significant preprocessing steps that require other libraries, toolset, and components. *ReduSharptor* does not require any preprocessing steps, the developer has all the necessary information and they provide this information as it is. Both Perses and require a test script to be available, normally a .sh file or a batch file or similar. *ReduSharptor* does not require any of these as explained in the architecture in the next sections.

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

Figure 4. command line execution of ReduSharptor

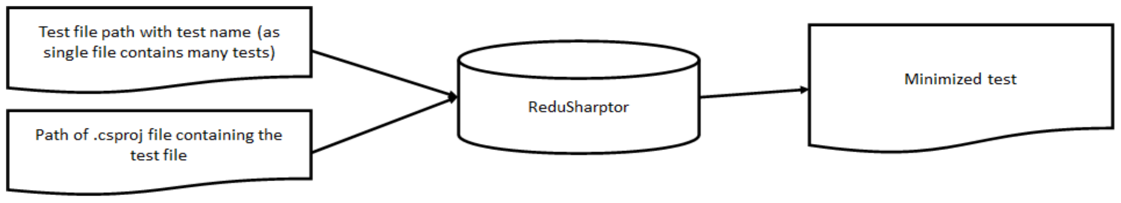

Test file path with test name (as single file contains many tests)
Path of .csproj file containing the test file
ReduSharptor
Minimized test

Figure 5. *ReduSharptor* architecture from a user's perspective

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 and run utility available as part of .NET framework and Roslyn compiler to generate necessary build and run script. The process is described in the right side of Figure 6. On the left side, we describe how a test is processed first using 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 DD algorithm to produce minimized test. Perses and Picireny require the user of their tool to provide the test script - test script may get complex sometimes. Also, both require a new test script for each test minimization.

4.3. IMPLEMENTATION

We sincerely attempted not to have any external dependencies, libraries or tool set. The implementation completely use .NET framework and Roslyn compiler API available as part of *Microsoft.CodeAnalysis* library. Because of this, a user can easily invoke *ReduSharptor* as a command line utility without worrying about downloading or maintaining any external components or libraries. The synergy between using a C# based tool and C# unit tests helps to have no preprocessing steps.

Table 1. Subject projects, LOC (Line of Code), # of tests, and total commits..

Project	LOC	# of Tests	# of Commits
language-ext [17]	318157	2610	3032
Umbraco-CMS [18]	156992	2637	42491
Fleck [19]	3576	92	237
BizHawk [20]	1686865	98	19860
Skclusive.Mobx.Observable [21]	7970	41	26

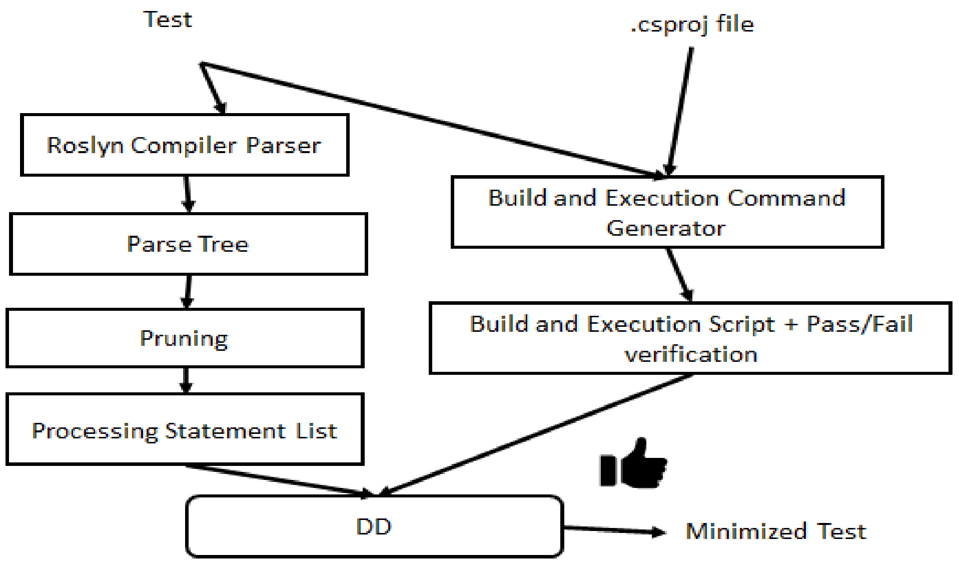

Test
.csproj file
Roslyn Compiler Parser
Parse Tree
Pruning
Processing Statement List
Build and Execution Command Generator
Build and Execution Script + Pass/Fail verification
DD
Minimized Test

Figure 6. *ReduSharptor* internal architecture with implementation details

5. Evaluation

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. Experiments

5.1.1 Subjects

We want to use any existing C# bug repositories like Defects4J for java for our evaluation [14]. We are unaware of any such repository. Even the benchmark list on the program repair website, does not mention any C# benchmarks [15]. We use 5 open-source C# projects listed in Table I. Among the five, except *Skclusive.Mobx.Observable*, others are under active development. After selecting the subjects, we looked for existing bugs in those projects. We went through commits and see if any of the commits or any snapshot of the software has a failing test. It seems that conscious developers normally run unit tests before committing to the repository and hence, we cannot find failing tests with any snapshot of the repository. We then searched for commits whose description seems to be associated with some bug. We grab the current version of source code and apply the commit in reverse as best as we can manually until it produces at least one failing test. Sometimes we need to utilize more than one related commits to recreate a bug. When a particular reversal of source code produced a failing test case, we preserve those changes as a bug and note down the failing test. The bugs (failing tests) that we have are based on commits but we resist to call them real bugs. We call them synthetic bugs and hope that they are close resemblance to real bugs. The synthetic bugs is good intermediate solution between real bugs and mutants.

Table 2. Subject tests, Stmts - number of statements , TreeStmt – number of Tree Statements and NonTreeStmt – number of non tree statements.

Tests	**Stmts**	**NonTreeStmts**	**TreeStmts**
ListCombineTest	10	10	0
EqualsTest	7	7	0
ReverseListTest3	5	5	0
WriterTest	17	15	2
Existential	14	14	0
TestMore	55	55	0
CreatedBranchIsOk	54	54	0
CanCheckIfUserHasAccessToLanguage	19	17	2
Can_Unpublish_ContentVariation	28	28	0
EnumMap	11	11	0
InheritedMap	17	17	0
Get_All_BluePrints	25	23	2
ShouldStart	7	5	2
ShouldSupportDualStackListenWhenServerV4All	4	3	1
ShouldRespondToCompleteRequestCorrectly	15	15	0
ConcurrentBeginWrites	21	21	0
ConcurrentBeginWritesFirstEndWriteFails	27	26	1
HeadersShouldBeCaseInsensitive	7	7	0
TestNullability	15	15	0
TestCheatcodeParsing	8	7	1
SaveCreateBufferRoundTrip	31	29	2
TestCRC32Stability	27	25	2
TestSHA1LessSimple	14	14	0
TestRemovePrefix	14	14	0
TestActionModificationPickup1	23	21	2
TestObservableAutoRun	26	25	1
TestMapCrud	39	38	1
TestObserver	104	101	3
TestObserveValue	62	59	3
TestTypeDefProxy	53	51	2
Total	**759**	**732**	**27**

Once we have a failing test the story doesn't end there. We need to ensure that the failing test should at least have some removable component(s) - a statement, a block of code or a part of an expression statement such that after it is removed the test continue to fail the exact same way. We prune the failing test if we don't find any such component. Applying ReduSharptor is meaningless as it won't reduce anything. The same is true if a developer attempts to reduce the test manually.

Using the process, we created 30 synthetic bugs that has 30 failing tests that are reducible. Table 2 describes each individual failing test we consider, the corresponding statements (as defined by *StatementSyntax* in Roslyn compiler), the number of NonTree statements and the number of Tree statements. .

5.1.2 PROCESS

For each failing test, we manually find a minimal test that still continues to fail the same way. As developer-written unit tests are simple enough to work with, it was not difficult to manually find minimal tests. For all 30 failing tests we create minimal tests and build a gold standard for comparison. Then we use *ReduSharptor* to reduce the failing tests.

5.1.3 MEASUREMENTS

We compare the results generated by the tool with the gold standard, matching each failing test in the gold standard with the corresponding failing test generated by *ReduSharptor*. We collect the following information.

1. True-Positive - Statements that are removed correctly and matches with the gold standard.
2. False-Positive - Statements that are incorrectly removed.
3. False-Negative - Statements that are missed.

5.2. RESULTS

Now we present our findings.

5.2.1 APPLICABILITY

We applied *ReduSharptor* on 30 failing tests for 30 synthetic bugs of 5 open-source C# projects. We process 759 statements. During the application *ReduSharptor* did not have any exceptions or unexpected behaviour. We ran into a few issues but were quickly able to fix them. It was successfully able to finish and produce the minimal failing test. So, ReduSharptor is highly applicable.

In addition to this, most of the unit tests that were simplified have mostly non-tree structures and allow for *ReduSharptor* to work effectively on them. Out of all the statements across all of the unit tests we simplified, only 27 statements had tree-like structures as seen in figure 6. Therefore, even though *ReduSharptor* did not simplify these statements to the granularity of an HDD algorithm, it was still very effective for these tests.

5.2.2 ACCURACY

We report accuracy using the standard measure of precision and recall as given below. Precision measures how many statements we removed that were not supposed to be removed. Recall measures how many statements we missed that we were supposed to remove.

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

We found *ReduSharptor* has 96.58% precision and 96.45% recall. That means we missed approximately only 3.5% of statements that were supposed to be removed and approximately 3.5% statements were inaccurately removed that were not supposed to be removed. Hence, we can claim that *ReduSharptor* is highly accurate in performing failing test minimization.

5.2.3 A Discussion On Inaccuracy

Though we don't have a large data set, we evaluate our inaccuracies to further understand the results. We note that most of the false negatives are due to Tree statements. This makes sense: We only process NonTree statements that are just below the *BlockStatmentSyntax* and if a Tree statement is present,we treat it as a single NonTree statement. The present of a Tree statement will cause missed opportunities in processing that may turn into missed removal of statements. The high precision and recall numbers suggest that our observation was correct. Even we treat Tree statement as a single NonTree statement, test minimization is very accurate in practice. Most of the false positives are due to tool limitations and other issues.

5.2.4 Why don't we compare with other tools?

None of the test minimization tools that we discussed before has a C# implementation to the best of our knowledge. To implement those techniques and algorithms in C# for comparison purpose is beyond the scope of this work.

6. Threats To Validity

6.1. Construct Validity

Do our results indeed reflect the advantage of *ReduSharptor*?
Our results fail to reflect the advantage if (1) Tool is inaccurate or (2) If the gold standard we created is inaccurate. We extensively tested our tool on different statement types, tests and programs. We carefully and accurately constructed the gold standard. As the developer-written unit tests were simple and small, it was easy to create the gold standard.

6.2. Internal Validity

Did we mitigate bias during our experiments using *ReduSharptor*?
All 5 projects that we used were open-source projects and none of the authors work on these projects. We randomly sampled 6 bugs from each project such that (1) the bug has one failing test (2) the test was reducible.

6.3. External Validity

Do our results generalize?
The five projects that we use are from different domains. They have different development team, different timeline, and different purpose. We expect our results to generalize with most C# projects and unit tests. If a tool like *ReduSharptor* is developed for other programming languages like Java, Python; we expect our results to generalize as most programming languages and unit tests have similar structure and similar compiler API.

6.3. Reliability

Is our evaluation reliable?
All the projects used for this evaluation is available online. Our tool *ReduSharptor* is also available online. The tool is published at https://github.com/TheWebRage/ReduSharptor.

CONCLUSIONS

Research tools are mostly focused on a very limited set of programming languages - mainly C and Java. As more C# projects are available in open source, availability of the tools in C# will let us compare and validate concepts and tools. If we want to see widespread adoption of research tools in the industry, we need to factor in the ecosystem that a particular programming language developers use. Ease of use should be given the utmost priority. Developers time is very costly. If they need to perform time-consuming preprocessing steps and produce test script that can be used by a particular tool for test case reduction, they won't likely use it until they see a significant benefit. While developing *ReduSharptor*, we considered the C# ecosystem that uses visual studio, .NET cs projects (.csproj), solutions (.sln), and unit test frameworks. Because of this, *ReduSharptor* is easy to use, applicable, and accurate.

ACKNOWLEDGEMENTS

The authors would like to thank Dr. Nicole Anderson and Dr. Yong Zhang for their important feedback.

REFERENCES

[1] D. Vince, R. Hodova´n, and A´. Kiss, "Reduction-assisted fault localiza- tion: 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 Pro- ceedings of the 28th International Conference on Software Engineering, ser. ICSE '06, 2006, pp. 142–151.

[5] R. Hodova´n 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, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '12. ACM, 2012, pp. 335–346.

[11] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree- structured test inputs," in Proceedings of the 32Nd IEEE/ACM Interna- tional 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] A. Christi and A. Groce, "Target selection for test-based resource adaptation," in 2018 IEEE

InternationalConference on Software Quality, Reliability and Security (QRS), July 2018, pp. 458–469.

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

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

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

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

[18] "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 [24] J. Staten, "Statianzo/fleck: C# websocket implementation." [Online].

[19] Available: https://github.com/statianzo/Fleck

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

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

AUTHORS

David Weber is an embedded software engineer at Northrop Grumman at Roy UT, USA. He completed his Masters Graduate studies at Weber State University in Ogden UT, USA focusing his research in adaptive programming, high performance computing, and embedded systems. He has experience in working in domains like defense applications and embedded software.

Dr. Arpit Christi is an assistant professor at School of computing, Weber State University, Ogen, UT, USA. He did his Ph.D. in Computer science from Oregon State University, Corvallis, OR, USA. His Research interests are program debugging, software testing, and self-adaptive software. He has many years of industry experience working in domains like law and justice, oil and natural gas industry, and financial planning.