Avoiding the Familiar to Speed Up Test Case Reduction

Golnaz Gharachorlu Nick Sumner
School of Computing Science
Simon Fraser University
Burnaby, BC Canada
{ggharach,wsumner}@sfu.ca

***Abstract*—Delta Debugging is a longstanding approach to automated test case reduction. It divides an input into chunks and attempts to remove them to produce a smaller input. When a chunk is successfully removed, all chunks are *revisited*, as they may become removable from the smaller input. When no chunk can be removed, the chunks are subdivided and the process continues recursively. In the worst case, this revisiting behavior has an $O(n^2)$ running time. We explore the possibility that good test case reduction can be achieved *without* revisiting, yielding an $O(n)$ algorithm. We identify three independent conditions that can make this reasonable in practice and validate the hypothesis on a suite of user-reported and fuzzer-generated test cases. Results show that on a suite of large fuzzer-generated test cases for compilers, our $O(n)$ approach yields reduced test cases with similar size, while decreasing the reduction time by 65% on average.**

***Keywords*—Automated Debugging; Test Case Reduction; Delta Debugging**

I. Introduction

Reproducing a bug is a critical part of the debugging process [1]. One key component of this is a test case or input that exhibits the bug. However, these test cases may be large and unwieldy. They may include irrelevant information, making it unclear which parts of the test case are actually related to the buggy behavior. They may also include repetitive or unnecessary information that is related to the bug but not required in order to understand the bug's behavior and fix it. Thus, such extraneous information in a test case may hinder the task of understanding and fixing a bug. This can be particularly problematic for test cases that result from automated test generation techniques. While techniques like random testing [2] and fuzzing [3] have shown great effectiveness and grown in usage, part of that effectiveness comes from constructing large, complex test cases [4, 5]. As a result, tools that can help *automate* the process of reducing a test case can be of great value.

Delta Debugging [6] is a fundamental technique for automated test case reduction. It attempts to find a smaller input for a test case such that some property of interest (usually a failure that needs to be debugged) still holds in the reduced version. The technique works by first partitioning an input into chunks and then attempting to remove or retain individual chunks to produce a smaller input. Note that removing one part of an input may enable *other* parts of the input to also be removed. Thus, when the algorithm removes a chunk of the input while still reproducing the desired property, all remaining chunks are *revisited*, as they may become removable themselves. When no further chunks can be removed, the chunks are recursively partitioned and the process continues. Because successfully removing a chunk may cause all already visited chunks to be revisited and tested again, this revisiting behavior causes the algorithm to perform $O(n^2)$ tests in the worst case where n is the size of the input. For large inputs this can be costly, easily taking hours to reduce a single input [5]. Hence, techniques for further speeding up the test case reduction process are desirable.

By only slightly modifying the Delta Debugging algorithm to skip the process of revisiting already considered chunks of the input, we can reduce the number of tests in the worst case behavior from $O(n^2)$ to just $O(n)$. However, this revisiting behavior is a key part of ensuring the minimality guarantees that Delta Debugging provides [6]. By skipping this process, one might expect that the test case reduction becomes grossly less effective. However, the return on investment (time vs. reduction effectiveness) is not well studied.

In this paper, we explore the possibility that this expensive revisiting process *may not be necessary in practice*. We observe that three key formal and empirical conditions of test case reduction can enable the process to produce minimal or near minimal test cases while skipping revisiting of already considered chunks. This enables us to make use of test case reduction that in the worst case performs $O(n)$ tests in practice. More specifically, we make the following contributions:

1) We identify three independent conditions for test case reduction that can eliminate the need for revisiting: (1) *common dependence order*, (2) *unambiguity*, and (3) *deferred removal*. Among these conditions, the idea of common dependence order is briefly mentioned but not explored in some research and development works [7, 8, 9]. Unambiguity is proposed by Zeller in [10]. The last condition, deferred removal is a notion defined in this paper.
2) We empirically validate that these conditions fre-

quently occur in practice in a suite of buggy test cases for compilers (both reported and fuzzer-generated).
3) Under the assumption that these conditions hold, we propose *One Pass Delta Debugging*, a simpler formulation of Delta Debugging with linear time complexity. We also explore how dissatisfaction of these conditions does not negatively impact the correctness of our algorithm.
4) We empirically evaluate *One Pass Delta Debugging* on a set of test cases and compare its results with classic Delta Debugging in terms of the size of the reduced test cases, number of steps in the reduction process, and the reduction time. Our approach reduced the reduction time by 0-73% across all the benchmarks with an average improvement of 33%. This average was 65% for the more pathological fuzzer-generated test cases. Our final reduced test cases were the same in 9/15 benchmarks. Most of the remaining 6 differed by only a few tokens.

The rest of this paper is organized as follows: In the next section, we present a brief description of the Delta Debugging algorithm. In section III, we explain the above-mentioned three conditions and discuss how they can lead to a simpler version of Delta Debugging. In section IV, we propose our algorithm. Section V empirically evaluates the occurrence and frequency of the conditions defined in section III in practice and measures the practical impact of exploiting them on the performance of test case reduction. The paper closes with discussion, related work and conclusions.

II. Background

The aim of test case reduction is to take a test case satisfying some property and produce a smaller test case satisfying the same property. Often, the property of interest is that the test case exhibits a particular failure. From here on, we refer to a test satisfying a particular property as "inducing the failure" without loss of generality. More formally, we define an oracle ϕ as follows[1]:

Definition 2.1 (Oracle): An *oracle* ϕ is a function such that, for a test case τ,

$$\phi(\tau) = \begin{cases} \text{✗} & \text{if } \tau \text{ induces the failure} \\ \text{✓} & \text{else} \end{cases}$$

Delta Debugging is a well established approach for test case reduction [6]. Starting from a failure inducing test case τ, Delta Debugging explores the search space of subsequences of the test case using a greedy search algorithm. These smaller tests are constructed by removing portions of the original test case and running an oracle to determine whether they trigger the failure or not. If a smaller test induces the failure, then the process continues recursively on that smaller test. The *ddmin* algorithm for test case reduction using Delta Debugging is presented in Fig. 1.

Intuitively, the algorithm uses the helper $ddmin_2$ to consider test case $\tau_{\text{✗}}'$ partitioned into n subsets (with n initially 2). *ddmin* can be broken into the following steps:

1) **Reduce to subset.** It checks whether there exists a partition Δ_i such that $\phi(\Delta_i) = \text{✗}$. If so, the process continues recursively on that subset.
2) **Reduce to complement.** Otherwise, it checks whether there exists a complement $\nabla_i = \tau_{\text{✗}}' - \Delta_i$ such that $\phi(\nabla_i) = \text{✗}$. If so, the process continues recursively on that complement (without repartitioning).
3) **Increase granularity.** Otherwise, if the partitions have not reached the finest granularity, it subdivides each partition if possible and proceeds to step 1.
4) **Done.** It otherwise terminates and returns $\tau_{\text{✗}}'$ as the reduced test case.

While the best case performance of *ddmin* is logarithmic in the size of the test case, the worst case behavior is quadratic [6]. This quadratic behavior arises because of step (2) in the algorithm. When reducing to a complement, every partition within that complement is considered, even if it was already tested once. Pathologically, the last complement visited may succeed, causing all remaining partitions to be visited again recursively, and the process continues. Thus, this *revisiting* of already considered partitions is a cause of the worst case quadratic behavior of the algorithm

In practice, the pathological quadratic behavior can arise due to dependencies in the test case. That is, one part of a test may not be removed unless another part of the test is also removed. Consider the example in Fig. 2. The original test case is `abcdefgh`. Here, each character represents a chunk of the test case and the test case could be any type of an input that needs to be simplified (e.g. a program that causes a compiler to crash). n is the number of subsets that the test case is divided into in a given step. Assume that the minimal failure inducing test case is `aceg`, in which every single character is necessary to trigger the failure. However, `b` cannot be removed without first removing `d`, nor `d` without `f`, nor `f` without `h`. In this case, all trials until $ddmin_2$ reaches the finest granularity do *not* induce the failure. Then the complements of the last removable elements succeed in reverse order, requiring all of the remaining complements to be tried again. More specifically, in a subset or complement trial, the portion under test is a subset (Δ_i in Fig. 1) or complement of a subset (∇_i), respectively. If the oracle fails in a complement trial (Reduce to complement), the test case is updated by removing the complemented subset. A revisiting process is then performed without repartitioning to re-run all previous complement trials in that granularity with the new updated test case. Three rows are highlighted in Fig. 2 to depict this revisiting process clearly. After successfully

[1]Defining a third output for Delta Debugging oracle function to represent unresolved cases is considered unnecessary [11, 12, 13, 14, 15, 16].

Given an initial test case $\tau_{\text{✗}}$ such that $\phi(\tau_{\text{✗}})$=✗,

$$ddmin(\tau_{\text{✗}}) = ddmin_2(\tau_{\text{✗}}, 2)$$

$$ddmin_2(\tau_{\text{✗}}', n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \mid \phi(\Delta_i) = \text{✗} \\ ddmin_2(\nabla_i, max(n-1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \mid \phi(\nabla_i) = \text{✗} \\ ddmin_2(\tau_{\text{✗}}', min(|\tau_{\text{✗}}'|, 2n)) & \text{else if } n < |\tau_{\text{✗}}'| \\ \tau_{\text{✗}}' & \text{else} \end{cases}$$

where $\nabla_i = \tau_{\text{✗}}' - \Delta_i$, $\bigcup_{i=1}^{n} \Delta_i = \tau_{\text{✗}}'$, all Δ_i are pairwise disjoint and $\forall \Delta_i, |\Delta_i| \approx |\tau_{\text{✗}}'|/n$.

Fig. 1: The minimizing Delta Debugging algorithm [6].

n	Step Description	Test Case $\tau_{\text{✗}}$								$\phi_1(\tau_{\text{✗}})$
1	Initial	a	b	c	d	e	f	g	h	✗
2	Subset trial	a	b	c	d	-	-	-	-	✓
2	Subset trial	-	-	-	-	e	f	g	h	✓
4	Increase granularity Subset trial	a	b	-	-	-	-	-	-	✓
4	Subset trial	-	-	c	d	-	-	-	-	✓
		...								
4	Complement trial	-	-	c	d	e	f	g	h	✓
4	Complement trial	a	b	-	-	e	f	g	h	✓
		...								
8	Increase granularity Subset trial	a	-	-	-	-	-	-	-	✓
8	Subset trial	-	b	-	-	-	-	-	-	✓
		...								
8	Complement trial	-	b	c	d	e	f	g	h	✓
8	Complement trial	a	-	c	d	e	f	g	h	✓
		...								
8	Reduce to complement Test case updated	a	b	c	d	e	f	g	-	✗
7	Revisiting	-	b	c	d	e	f	g	-	✓
7	Revisiting	a	-	c	d	e	f	g	-	✓
		...								
7	Reduce to complement Test case updated	a	b	c	d	e	-	g	-	✗
6	Revisiting	-	b	c	d	e	-	g	-	✓
		...								
4	Final	a	-	c	-	e	-	g	-	✗

Fig. 2: Quadratic complexity in classic Delta Debugging. Removal order: h,f,d,b (Oracle ϕ_1 defined in Fig. 3)

removing `h` from the test case and reducing it to `abcdefg`, `a` is reconsidered for removal from the new test case in the next step. This revisiting occurs after each reduce to complement.

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

III. Conditions for Skipping Revisiting

Why might skipping revisits be okay? In this section, we identify three key reasons: common dependence order, unambiguity and deferred removal.

A. Common Dependence Order.

Just as in Fig. 2, sometimes there is a preferred ordering that can improve the overall efficiency of Delta Debugging. For example, when removing statements from a C program, uses of identifiers must be removed before their declarations in order to satisfy validity constraints. Since the uses mostly occur after their definitions, removing the lines of a C file in reverse order is likely to remove the uses first and thus enable removing declarations when they are first visited. When the common ordering of dependences for a particular test case structure is known, the complement trials can be ordered to respect the dependences. We refer to this condition as the *common dependence order*.

B. Unambiguity.

Unambiguity was a property proposed in the original version of Delta Debugging [10], dd^+.

Definition 3.1 (Unambiguity): A failing test τ is unambiguous if $\forall \tau_1, \tau_2 \subseteq \tau, \phi(\tau_1) = \text{✗} \wedge \phi(\tau_2) = \text{✗} \implies \phi(\tau_1 \cap \tau_2) = \text{✗}$.

Informally, an ambiguous test case (the one in which unambiguity does not hold) has multiple causes for its

failure. A reduced test case that satisfies any of these causes is failure-inducing. Consider the example in Fig. 4. ϕ_2 in Fig. 3 fails if both `a` and `c` are present in the test case and passes otherwise. Test case τ with this oracle is unambigious since both `ef` and `gh` can be removed from τ individually, and the test case generated by combining the results of these two removals still induces the failure. However, in τ with ϕ_3, the property of interest is the presence of either `ch` or `df`. Hence, 3.1 does not hold in this scenario because ϕ_3(`abcdef`)=✗ $\wedge$ ϕ_3(`abcdgh`)=✗ but ϕ_3(`abcd`)=✓. Note that an unambiguous test τ enables reasoning about subtests τ_1 and τ_2 independently because the results can be recombined using intersection, which is independent of order. Thus, if unambiguity holds for a test, the order in which reduce to complement operations are performed is irrelevant and revisiting need not be performed.

C. Deferred Removal.

Consider the example in Fig. 5. (a) performs revisits while (b) skips them. Assume that at some point in the reduction process, we have input `abcdefghijkl` with $n = 6$ and `ij` cannot get removed before `kl`. The highlighted rows in (a) are the tests that are skipped and not performed in (b). These tests are the ones that are executed after a reduce to complement and are referred to as *revisiting* as described in section II. As can be seen in Fig. 5(a), after successfully removing `kl` from the input, all previously visited subsets are tested for removal from the new updated input (highlighted rows). As described earlier in this section, if common dependence order and unambiguity hold, revisiting is not necessary. However, we can see that among the highlighted rows, there is a failing oracle that corresponds to the successful removal of `ij` in the revisiting round (the row marked with $(*)$ in Fig. 5(a)). A successful revisit test (with failing oracle) indicates that the two previous conditions, common dependence order and unambiguity, are not satisfied. However, there is also a third condition in the reduction process that we call *deferred removal* that can make revisiting unnecessary even if the other two conditions do not hold. The test marked with $(*)$ in Fig. 5(a) can be replaced with two tests marked with $(*)$ in Fig. 5(b) when the granularity increases ($n = 10$). The reduction process used by Delta Debugging performs refinement and reduction until each element of a test resides in its own partition and no one element can be removed. Thus, if an element of a test could not be removed while reducing at one granularity, it may be removed at a finer granularity when the element is considered in a refined partition. In our example, if `ij` could not get removed due to skipping revisits in (b), it can get removed as `i` and `j` when the granularity increases (from $n = 6$ to $n = 10$) through the natural flow of the Delta Debugging algorithm.

Note that deferred removal comes at a cost. Every time that refinement occurs, a subset that could have been removed but was not turns into two smaller subsets that will be tested independently. Thus, the cost of removing a missed subset may grow exponentially with the number of refinements. However, the worst case cost is actually linear in the size of the subset. At the same time, revisiting as in classic Delta Debugging would require reconsidering all reduce to complement opportunities, leading to the $O(n^2)$ behavior but possibly at much coarser granularities where n is smaller. It is not clear what the performance trade off is in practice. We empirically explore this trade-off in section V. The next section presents a formal definition of our algorithm.

IV. One Pass Delta Debugging

Inspired by the observation that revisiting subsets after a "reduce to complement" operation may not be necessary, we consider a variant of Delta Debugging that considers each subset and its complement only once per granularity. We call this variant *One Pass Delta Debugging (OPDD)* presented in Fig. 6. The key changes from *ddmin* are highlighted. Intuitively, the major difference in OPDD is in the handling of "reduce to complement" cases. Instead of recursively restarting the reduction process for each reduce to complement opportunity, OPDD successively tries removing each remaining unvisited subset from the current minimal test case.

This is captured by applying a standard left fold (*foldl*) of *complement*? on the list of remaining subsets in Fig. 6. *complement*? simply tries to remove a particular subset from $\tau_{✗}{}'$ and returns the smaller test when it still reproduces the failure. *foldl* is a higher-order function that combines the results of *complement*? executions in order from left to right. After applying the left fold, OPDD calls

$\phi_1(\tau_✗)$= ✗ if (a ∧ c ∧ e ∧ g) in $\tau_✗$; ✓ else

Dependencies: b ← d ← f ← h

$\phi_2(\tau_✗)$= ✗ if (a ∧ c) in $\tau_✗$; ✓ else

$\phi_3(\tau_✗)$= ✗ if (c∧h)∨(d∧f) in $\tau_✗$; ✓ else

Fig. 3: Oracles used in Fig. 2 and Fig. 4. Dependency ← for ϕ_1 means that removing the element on the left side of ← before removing the element on the right will not induce the failure.

	Test Case								$\phi_2(\tau_✗)$	$\phi_3(\tau_✗)$
τ :	a	b	c	d	e	f	g	h	✗	✗
τ_1 :	a	b	c	d	e	f	-	-	✗	✗
τ_2 :	a	b	c	d	-	-	g	h	✗	✗
$\tau_1 \cap \tau_2$:	a	b	c	d	-	-	-	-	✗	✓

Fig. 4: Satisfaction and dissatisfaction of unambiguity with ϕ_2 and ϕ_3 (defined in Fig. 3), respectively.

n	Test Case $\tau_{\text{✗}}$												$\phi(\tau_{\text{✗}})$
6	-	-	c	d	e	f	g	h	i	j	k	l	✓
6	a	b	-	-	e	f	g	h	i	j	k	l	✓
						...							
6	a	b	c	d	e	f	g	h	i	j	-	-	✗
5	-	-	c	d	e	f	g	h	i	j	-	-	✓
5	a	b	-	-	e	f	g	h	i	j	-	-	✓
5	a	b	c	d	-	-	g	h	i	j	-	-	✓
5	a	b	c	d	e	f	-	-	i	j	-	-	✓
5	a	b	c	d	e	f	g	h	-	-	-	-	✗ (∗)
4	-	-	c	d	e	f	g	h	-	-	-	-	✓
4	a	b	-	-	e	f	g	h	-	-	-	-	✓
						...							
8	-	b	c	d	e	f	g	h	-	-	-	-	✓
						...							

Removal order: {kl},{ij}

(a)

n	Test Case $\tau_{\text{✗}}$												$\phi(\tau_{\text{✗}})$
6	-	-	c	d	e	f	g	h	i	j	k	l	✓
6	a	b	-	-	e	f	g	h	i	j	k	l	✓
						...							
6	a	b	c	d	e	f	g	h	i	j	-	-	✗
10	-	b	c	d	e	f	g	h	i	j	-	-	✓
						...							
10	a	b	c	d	e	f	g	h	-	j	-	-	✗ (∗)
10	a	b	c	d	e	f	g	h	-	-	-	-	✗ (∗)

Removal order: {kl},{i},{j}

(b)

Fig. 5: (a) performs revisits while (b) skips them but they both remove the same elements due to satisfaction of the deferred removal condition.

Given an initial test case $\tau_{\text{✗}}$ such that $\phi(\tau_{\text{✗}})$=✗,

$$opdd(\tau_{\text{✗}}) = opdd_2(\tau_{\text{✗}}, 2)$$

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

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

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

Fig. 6: The One Pass Delta Debugging algorithm.

refine on the result to either refine the granularity or finish. This avoids a recursive invocation of $opdd_2$ that would reconsider removing particular subsets from the test case again. Since the "refine or finish" behavior is also present in the main algorithm, it has also been refactored to make use of *refine*.

Correctness. We consider a test case reduction algorithm correct when it is guaranteed to produce a test case that (1) induces the failure and (2) is no larger than the original test case. OPDD is an *anytime algorithm* that preserves correctness during all its executed steps meaning that in any step, the current version of the test case induces the failure. This holds regardless of whether the conditions described in section III are satisfied or not.

Proof 4.1: By induction: In the first step of the algorithm, the original test case triggers the failure. If the test case still induces the failure in step k, the test case in step $k+1$ either induces the failure or not. If it does, test case in step $k+1$ is trivially failure-inducing. If it does not, OPDD does not update the input since a failing oracle is used as an invariant in OPDD for failure-inducing inputs. Hence, the test case in step $k+1$ will remain the same as the one in step k. ■

Time Complexity. The interesting measure of algorithmic complexity is how the number of oracle queries grows with input size. Just like Delta Debugging, OPDD performs a logarithmic number of granularity refinements with a constant amount of work for each partition at a particular granularity (querying the oracle for each partition and its complement). As a result, the worst case number of oracle queries of OPDD is $O(n)$ where n is the size of the test case. In contrast, the revisiting of *ddmin* leads to quadratic complexity in the worst case [6] as observed in Fig. 2. In the best case scenario, both Delta Debugging and OPDD have logarithmic time complexity because both algorithms always perform a reduce to subset by dividing the test case into half. In practice, the closer the real-world cases are to the worst case scenario, the improvement in efficiency becomes more noticeable.

V. Empirical Validation

In this section, we explore the performance characteristics of OPDD in practice with a particular eye to the conditions mentioned in section III. In particular, we apply OPDD to a suite of real bugs in compilers, both user-reported and fuzzer-generated. We observe that not only can we pragmatically exploit common dependence order, unambiguity and deferred removal, but they can significantly impact the measured running time of the test case reduction process.

We continue this section by asking the following research questions:

- **RQ1.** Is there a common dependence order among elements of test cases in practice?

- **RQ2.** Does empirical evidence suggest practical satisfaction of unambiguity?
- **RQ3.** What is the performance of OPDD in terms of reduced test case size, reduction time and number of tests (number of times an oracle is queried)?
- **RQ4.** How can satisfaction of deferred removal play a role in OPDD test case reduction process? Does it help OPDD to generate reduced test cases comparable in size to those generated by classic Delta Debugging?

A. Test Cases and Experimental Set-up

To examine the impact of OPDD on the reduction of real test cases, we study two sets of test inputs presented in Table I: user-reported and fuzzer-generated.

The user-reported inputs are failure-inducing test cases in five different input domains (C, C++, Rust [17] , Go [18] and Dot [19]) collected from various online bug repositories [20, 21, 22, 23]. These test cases cause specific version(s) of the language compiler to crash or produce unexpected output. Starting from the most recently reported bugs at the time of conducting our experiments, we selected this set under the condition that we could reproduce the failure.

For fuzzer-generated test cases (fuzz_.c), we used Csmith [24], a tool for generating random C programs that statically and dynamically conform to the C99 standard [25] and can help to find and understand bugs in C compilers [4]. We generated five large C programs causing the Clang compiler [26] (version 3.2) to produce outputs that are behaviourally different at different optimization levels. These five test cases are the first generated ones that exhibited the bug deterministically. In total, we conducted our experiments on 15 test cases.

We chose Hierarchical Delta Debugging (HDD) [27, 28] as a variant of test case reduction that is more suitable for structured inputs such as programs written in different programming languages. In HDD, nodes of the program's abstract syntax tree (AST) are defined as deltas (Δ_i in Fig. 1). Delta Debugging is then applied at each level of the AST. Our comparison then involves selecting either Delta Debugging or OPDD when performing test case reduction in each level of the AST. Threats to validity are discussed in Section VI. The next subsection presents an overview of HDD.

TABLE I: Test cases used in our experiments

test case	size		failing compiler
	tokens	bytes	
1.c	9062	45507	clang r209066
2.c	1511	6513	gcc 6.0.1
3.c	30645	200819	gcc 6.0.1
1.cpp	426	2600	clang++ 3.7
2.cpp	917	3900	g++ 6.3
1.rs	399	1567	rustc 1.16 beta
2.rs	405	1693	rustc 1.9 nightly
3.rs	519	2514	rustc 1.17 stable
1.go	1732	7554	go 1.7
1.gv	13449	77148	dot-graphviz 2.38
fuzz1.c	62300	191350	clang 3.2
fuzz2.c	35629	117838	clang 3.2
fuzz3.c	61771	198742	clang 3.2
fuzz4.c	42410	133634	clang 3.2
fuzz5.c	44473	146778	clang 3.2

B. Hierarchical Delta Debugging

As discussed in subsection V-A, our experimental inputs consist of test cases written in programming languages. These inputs are structured and defined based on the language grammar. We use Hierarchical Delta Debugging (HDD), a variant of Delta Debugging that is more suitable for structured inputs.

HDD applies Delta Debugging on levels of an input parse tree or abstract syntax tree (AST) with the goal of generating more valid test variants and pruning away larger portions of the input at an early stage.

To clarify how HDD works, consider the example in Listing 1. This example is a C program with a buggy statement marked with (*). To reduce this test case, Delta Debugging simply partitions the input based on tokens (or characters) and starts by generating the two subsets $\triangle_1$={`void f() {int x = 5;} void g()`} and $\triangle_2$={{`int y = 10; int z = 15/0;`}} that are both invalid C programs. In contrast, HDD generates input variants by applying Delta Debugging on levels of an input AST.

In this example, HDD applies Delta Debugging on level 1 of the tree (the node at level 0 is `compilationUnit` and cannot be removed) and removes the node corresponding to function `f()` and all its descendents since they are irrelevant to the buggy statement. Function `g()` cannot be removed because its buggy child node is required for the failure. On the next level, HDD removes `int y = 10;` while preserving the buggy statement.

Applying HDD on structured inputs not only increases the likelihood of generating grammatically valid input variants but also speeds up the reduction process by removing irrelevant nodes and their descendents simultaneously.

C. RQ1. Common Dependence Order in Practice

As described in section III, if the preferred dependence order among elements of a test case is known, complement

```
void f() {
 int x = 5;
}
void g() {
 int y = 10;
 int z = 15/0; (*)
}
```

Listing 1: A structured test case (C program)

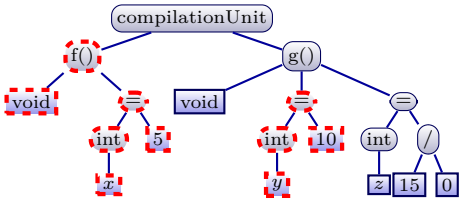


Fig. 7: Applying HDD on AST of input in Listing 1

trials can be ordered in a way such that performing revisits becomes unnecessary. In this section, we would like to examine whether this common dependence order exists in practice or not.

To conduct this study, we measure the number of successful revisiting subset removals. A successful subset removal means that the test case without the subset (the complement of the subset) causes the oracle to fail (induces the failure). The number of successful revisiting subset removals is the number of subsets that were *not* successfully removed in the first round of running the oracle, but after updating the test case, they were successfully removed in a revisiting round. We consider two natural orderings for subset removals: forward($\rightarrow$) from the beginning of the test case to the end and backward($\leftarrow$) from the end of the test case to the front. The idea is that the ordering with fewer successful revisiting subset removals is the preferred ordering. For instance in Fig. 2, the preferred ordering is backward with no successful revisits while the forward ordering causes quadratic behavior.

Results are shown in the second and third columns in Table II. As can be seen, for most of the test cases (9 out of 15), the backward ordering is the preferred ordering, which is expected when reducing compiler test cases. For instance, the use of a variable is usually after its declaration and should get removed first to make removal of the declaration possible. Hence, based on this evidence, we can infer that the common dependence order among programming language test cases is backward. However, we can see cases in which the forward ordering is better or where there is no difference between the two orderings. In addition, we can clearly see that even for backward dependence orders, successful revisits still occur. Hence, we *cannot* conclude that if subsets of a test case are removed in backward order, then performing revisits becomes completely unnecessary.

We later show that deferred removal helps to address cases where a common dependence ordering fails.

D. RQ2. Unambiguity in Practice

As described earlier, if test cases are unambiguous, the removal of different subsets should be independent from each other. Thus, OPDD should produce the same results as classic Delta Debugging.

Parallel Subset Removal. To examine occurrence of unambiguity in practice, we implement a parallel subset removal mechanism. Recall that Delta Debugging and therefore OPDD partition a test into n disjoint subsets where n defines the granularity. At any given granularity with input $\tau = \cup_{i=1}^{n} s_i$, we compute $\phi(\tau - s_i)$ for all subsets in parallel (LHS of 3.1), maintaining a list S of subsets that can be potentially removed successfully from τ (decided based on the outcome of $\phi(\tau - s_i)$). Finally, at the end of the given granularity analysis, we construct a cumulative input I consisting of subsets in S and then we compute $\phi(\tau - I)$ (RHS of 3.1). If $\phi(\tau - I) = \text{✗}$, then unambiguity holds at the given granularity with respect to the subsets S and testing continues with the reduced test case. Otherwise, unambiguity does not hold for the subsets in S.

For each test case, we compute the total number of granularities with at least one individual successful subset removal that were reached and analyzed during complement trials in the reduction process. The fourth column in Table II shows the number of granularities with only one successful subset removal. Note that unambiguity *trivially holds* for these granularities since there are no other complements to conflict with the successful test. The fifth column shows the number of granularities with at least two individual successful removals and which the combined set S was also successfully removed. The sixth column is the number of granularities with at least two individual successful removals but where removing the combined set does not produce a test case that induces the failure. As can be seen, the last case where unambiguity fails rarely occurs, which results in a large proportion of cases where unambiguity holds in practice as shown in the last column. This number is the ratio of granularities satisfying the unambiguity condition to the total number of granularities with at least one successful subset removal.

E. RQ3. OPDD Performance

Thus, both common dependence order (to some degree) and unambiguity (strong) have some empirical support in our suite of test cases. This provides some intuition that OPDD may be successful in practice. We run OPDD to compare its performance with classic Delta Debugging. Our metrics are the size of the final reduced test case, the reduction time, and the number of performed tests in both techniques. We discuss the deferred removal condition and its impact in subsection V-F.

Similar to [11], our implementation of Delta Debugging in this paper takes advantage of parallel workers for subset and complement trials to converge to a local minimum as fast as possible. That unambiguity mostly holds in practice also motivates the parallelization of Delta Debugging.

Results are shown in Table III. Interestingly, in 9 out of 15 cases, OPDD generates a reduced test case with *exactly the same size* of an input produced by classic Delta Debugging. In 5 cases, the increase in the size of the reduced input is negligible (less than 0.2% of original input size). The application under test for one of our test cases (Dot with 1.gv) shows nondeterministic behavior, meaning that results are different for each run of the Delta Debugging on the same input. Hence, we include the average result obtained from multiple runs of this input. In section VI, we discuss nondeterminism in more detail. These results strongly suggest that revisiting may not be necessary in practice.

Now we measure how skipping revisiting as in OPDD can improve the performance of test case reduction in terms of reduction time and the number of tests performed. The fourth column in Table III shows OPDD time improvement over classic Delta Debugging. Based on the results for user-

TABLE II: Statistics on successful revisits and subsets unambiguity in practice

test case	# successful revisits		# granularities with successful removal			unambiguity ratio (%)
	→	←	trivial (single removal)	successful combination	unsuccessful combination	
1.c	35	18	237	13	0	100
2.c	5	1	62	0	0	100
3.c	42	33	645	110	0	100
1.cpp	0	0	7	0	0	100
2.cpp	5	3	23	0	0	100
1.rs	0	0	22	0	0	100
2.rs	0	0	64	2	0	100
3.rs	4	4	60	0	0	100
1.go	2	0	7	0	0	100
1.gv	50	92	546	151	50	93.3
fuzz1.c	106	91	587	60	2	99.7
fuzz2.c	69	58	643	15	0	100
fuzz3.c	93	67	830	15	8	99.1
fuzz4.c	67	48	534	41	3	99.5
fuzz5.c	86	95	969	30	1	99.9

TABLE III: Performance statistics for OPDD and classic Delta Debugging

test case	reduction time (sec)[1]		OPDD time improvement (%)[2]	# tests[3]		size of reduced test case (# tokens)		reduction percentage (size)[4]	
	classic DD	OPDD		classic DD	OPDD	classic DD	OPDD	classic DD	OPDD
1.c	45	29	35.6	10810	9504	339	345	96.3	96.2
2.c	31	24	22.6	4991	4570	218	218	85.6	85.6
3.c	52	49	5.8	5736	5621	239	239	99.2	99.2
1.cpp	17	17	0.0	1858	1834	95	95	77.7	77.7
2.cpp	149	147	1.3	501	483	116	116	87.4	87.4
1.rs	12	10	16.7	687	579	55	55	86.2	86.2
2.rs	52	39	25.0	4846	4574	244	244	39.8	39.8
3.rs	449	385	14.3	6176	5952	274	274	47.2	47.2
1.go	33	28	15.2	2427	2371	166	166	90.4	90.4
1.gv	1243	709	43.0	38487	22741	332	215	97.5	98.4
fuzz1.c	34126	14200	58.4	174449	162469	2502	2506	96.0	96.0
fuzz2.c	14919	4953	66.8	81379	76994	1463	1518	96.0	96.0
fuzz3.c	33433	8772	73.8	101971	82735	1046	1049	98.3	98.3
fuzz4.c	10553	3588	66.0	63291	53404	910	915	97.9	97.8
fuzz5.c	20252	8163	60.0	103769	97122	1336	1336	97.0	97.0

[1] Oracle execution time included. [2] (DD.time-OPDD.time)/DD.time*100
[3] Number of times the oracle is executed. [4] (Original.size-DD(OPDD).size)/Original.size*100

reported test cases, except for one where OPDD had the same reduction time (1.cpp), we were able to decrease the time by 1.3% to 43%. That corresponds to between 2 and 64 seconds. The more interesting results are related to large fuzzer-generated test cases for which we could decrease the reduction time by 58% to 74%, an average of more than 4 hours. The number of tests also decreases significantly when skipping subset revisits. It is worth noting that these skipped tests are among expensive ones since they examine complements that are larger than subsets and can take longer to be processed.

Note that on average, 90% of successful tests were "reduce to complement" for our set of test cases. Recall that this step enforces the revisiting process in the classic Delta Debugging algorithm. As a result, revisiting tests are common in practice (between 29 to 19,252 tests for our set of test cases), but successful ones are rare (second and third columns in Table II). Although the low frequency of successful revisits is more evidence for supporting OPDD and the idea of revisiting subsets not being required in practice, we can see that successful revisits still occur in scenarios such where unambiguity does not hold or when a straightforward ordering (forward or backward) is not the preferred ordering of complement trials. Thus, an immediate question is why do we either not see any difference in the size of the reduced test case or see only negligible differences? More specifically, the last column in Table III shows the same reduction power for both OPDD and classic Delta Debugging. Why? Subsection V-F addresses this question.

F. RQ4. Deferred Removal: A Case Study

Why do we not see a more significant difference in the size of the reduced test cases generated by the two techniques? To answer this question, we set up an experiment to record when each unique token is removed from the input in the reduction process of both OPDD and classic Delta Debugging. Interestingly, we observed that if a token is not removed at a given granularity due to skipping revisits, it is very likely to get removed in subsequent finer granularities. This is precisely the motivation for deferred removal.

To explore further, we select a Csmith-generated test case (fuzz4.c) for a case study in this section. We explore the number of refinement steps between the time that a token is removed using Delta Debugging and the time a token is removed in OPDD. Fig. 8 (a) illustrates the

distribution of token removals. In all, 41,500 tokens were removed by classic Delta Debugging. We denote the time of removal in classic Delta Debugging as i regardless of its actual time. In practice, i ranges over different values since tokens were removed at different granularities. Recall that granularity increases by performing step 3 of classic Delta Debugging algorithm (see Fig. 1). As can be seen, when skipping subsets in OPDD, more than 92% of tokens are still removed at the same granularity as they were removed when subsets were revisited (granularity i). From the remaining ones not removed at granularity i, 56% are caught at the next granularity and very few are left for granularity $i + 2$ and deeper. In 41,500 tokens, only five (0.01%) were not removed at all when subsets were skipped.

Fig. 8 (b) also shows the distribution of successful removals. Coarse granularities comprise the first one-third of the reached granularities in total and consist of larger subsets. Finer granularities are the middle one-third, while the finest ones are the last one-third of granularities with the smallest subsets. As can be seen, at coarse granularities, the ratio of successful removals is almost the same for both techniques (23% vs. 22%). At granularities in between (not very coarse and not very fine), the technique with subset revisits (classic Delta Debugging) has higher frequency of successful removals, while skipping subsets (OPDD) yields a higher success ratio at finest granularities (70% without revisits vs. 61% with revisits) which also indicates that if some successful removals are skipped at previous granularities, they are still caught and removed at later ones.

These results indicate that in practice, we do not necessarily require a strict dependence order among elements of a test case or full unambiguity to be able to skip subset revisits without adversely affecting the performance of Delta Debugging in terms of the size of the reduced test case. It is worth mentioning that although postponing the successful removal of a subset to a later granularity increases the number of tests (since a subset is divided into two in the following granularity), sparsity of successful subset revisits in addition to the large number of unsuccessful revisits that are skipped should decrease the overall number of tests. For instance, in the reduction process of fuzz4.c, we skipped 11,000 tests while increasing the number of tests by only 1000. This results in an overall reduction of 10,000 tests leading to a decrease in reduction time by 66%. Finally, if deferred removal does not occur (a rare situation in practice for our benchmarks), OPDD will fail only in terms of effectiveness that is the size of the reduced test case. Its efficiency and correctness will not be affected due to skipping revisits, as seen in section IV.

VI. Discussion

In this section, we briefly describe an existing technique, dd+ [10] and also discuss some threats to validity of our study.

A. dd+: A version of Delta Debugging

dd+ is an older version of classic Delta Debugging with a linear time algorithm for test case reduction. Here, we explain how dd+ can generate an incorrect outcome (a reduced test case that is not failure-inducing) if unambiguity is not satisfied. This is in contrast to OPDD and the classic Delta Debugging version studied in this paper, the correctness of which does not depend on satisfaction of any conditions.

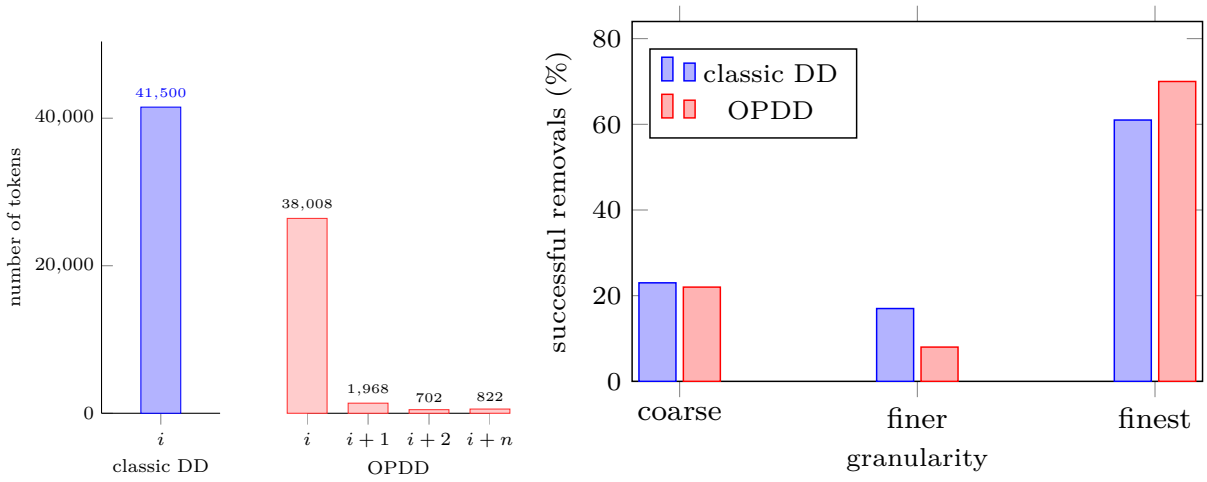
The dd+ algorithm relies on a notion called "remains set". It intuitively means that when searching through a portion of an input to find a smaller part that potentially causes the oracle to fail, the remaining part of the input that is not included in the portion under test will remain applied when the oracle is executed. Fig. 9 depicts how dd+ falsely reports a reduced test case as failure-inducing for the example in Fig. 4 when unambigiuity is not satisfied. This incorrectness suffices to not consider dd+ as a baseline in our experiments.

B. Threats to Validity.

In this subsection, we discuss how different factors such as input size, domain, number of successful tests and the oracle verification time can impact the performance of test case reduction with OPDD.

Input Size and Domain. The difference between the performance of classic Delta Debugging and OPDD may vary for different test cases. To address the generality of findings, we considered two sets of test cases, real-world bug reports and fuzzer-generated inputs with different size and domain. Intuitively, a larger input is more likely to benefit from skipping subset revisits if it reaches fine levels of granularity. The reason is that at a fine granularity, a larger input is partitioned into a larger number of subsets that increases the likelihood of redundant subset revisits. However, the greedy search algorithm of Delta Debugging can potentially shrink a large input to a small one quickly without getting into rounds of revisiting. Our results show that these cases are rare in practice. In our experiments, we observed the most significant improvement in terms of the reduction time and number of tests for large inputs. Moreover, we studied inputs written in various programming languages. The low success ratio of subset revisits obtained in our results indicate that subsets are mostly unambiguous in practice regardless of the considered language. However, the focus of this study is on minimizing test cases for compilers and the input domains are programming languages. A study of other domains can be pursued as a future work.

Oracle Verification Time. The time required to verify whether an oracle is satisfied for a test variant or not is dependent on various factors such as the size and structure of the test variant, the degree of complexity inside the oracle and the testing environment which itself may directly impact the ability to obtain the exact same results in terms of the test case reduction time. A larger test variant with



(a) Distribution of tokens removed at each granularity ($\forall i : 1 \leqslant i \leqslant |G|$ where $|G|$ is the total number of granularities and $min(2, |G| - i) < n \leqslant |G| - i$)

(b) Distribution of successful removals

Fig. 8: Statistics on tests performed on fuzz4.c with OPDD and classic DD)

Index	Step Description	Test Case $\tau_{\text{✗}}$								$\phi_3(\tau_{\text{✗}})$: (c∧h)∨(d∧f)
1	Initial	a	b	c	d	e	f	g	h	✗
2	Subset trial	a	b	c	d	-	-	-	-	✓
3	Subset trial	-	-	-	-	e	f	g	h	✓
4	subdivide the first half e f g h remain applied	a	b	-	-	e	f	g	h	✓
5	e f g h remain applied	-	-	c	d	e	f	g	h	✗
6	Increase granularity c found and returned	-	-	c	-	e	f	g	h	✗
7	subdivide the second half a b c d remain applied	a	b	c	d	e	f	-	-	✗
8	Increase granularity f found and returned	a	b	c	d	-	f	-	-	✗
9	Combine outcomes of steps 6 and 8 supposed to be failure-inducing but it is not	-	-	c	-	-	f	-	-	✓

Fig. 9: dd+ does not preserve correctness when unambiguity is not satisfied for the example in Fig. 4.

a more complex structure usually takes longer to parse and compile. In addition, oracles with complex structures are more expensive. For instance, the oracles for fuzzer-generated test cases in our study consist of 20 different steps (conditions). The oracle running time of skipped tests directly impacts the amount of time decreased in the reduction time of a test case.

Nondeterministic Behavior. In addition to the possible nondeterminism caused by parallelism, applications under test may also show nondeterministic behavior. Compilers (as examined in our study) are deterministic in general. However, address space layout randomization (ASLR) may cause a nondeterministic behavior of a compiler by randomizing loading memory locations of executables. To mitigate this problem, in addition to the multiple runs, we disabled ASLR during our studies. All applications under test in our experiments showed deterministic behavior except for Dot, which has nondeterminism resulting from factors such as the way it represents some data structures with its nondeterministic data storage mechanism. For this test case, we computed the average of values obtained in different runs.

Number of Tests. Finally, we used different metrics including the size of the final reduced input, reduction time and number of performed tests to measure the performance of OPDD. The number of tests in a parallel implementation of Delta Debugging may not be a precise indicator since multiple threads work simultaneously and increase the number of tests rapidly. However, this is more problematic when comparing a parallel algorithm with a sequential one.

VII. Related Work

Our work is closely related to Delta Debugging [10, 6] since it studies a possible improvement for this algorithm in practice. We show that the unambiguity property described by Zeller [10] strongly holds in practice. In addition, we define the notion of deferred removal and suggest that fully unambiguous test cases are not strictly required to produce test cases comparable to classic Delta Debugging. A finding

that makes the formulation of Delta Debugging simpler and more efficient for large test cases.

Here, we classify some related work into three different subcategories.

A. Delta Debugging Applications.

The generalized nature of Delta Debugging enables its use in a variety of domains where an input with a property of interest needs to be simplified. Any use of Delta Debugging can benefit from findings in this study. Some examples are as follows: Orso et al. [29] automatically isolate the subset of the interactions between components of a large system and their environment, while Hammoudi et al. [30] reduce recordings of events that lead to a failure of a web application using Delta Debugging. DEMi, a tool developed by Scott et al. [14], minimizes faulty executions of distributed systems by applying Delta Debugging as part of its minimization phase and Clapp et al. [13] propose a technique for minimizing GUI event traces generated for Android apps by utilizing a variant of Delta Debugging.

With regard to application of Delta Debugging in fuzz testing [3], Lei and Andrews [15] generate randomized unit tests and show that applying Delta Debugging is very effective for reducing sequences of method calls in the failing tests. Leitner et al. [31] propose that applying static slicing in conjunction with Delta Debugging could result in an even more efficient test case minimization technique. Brummayer and Biere [16] devise a grammar-based black box fuzz testing technique for generating robust SMT solvers and integrate Delta Debugging into their technique to obtain minimized SMT formulas that are more useful for test case generation, debugging and verification.

B. Improvements to Delta Debugging.

Some research works suggest improvements for the Delta Debugging algorithm. Kiss and Hodován [11] propose practical improvements such as parallelization to speed up test case reduction. Another closely related work is Hierarchical Delta Debugging (HDD) proposed by Misherghi and Su [27] and further extended to take advantage of the input structure such as parse trees for inputs written in programming languages to guide a structured pruning mechanism [28]. HDD itself has been widely studied for potential improvements [7, 12, 32, 33, 34]. Lithium [9] is another tool which performs revisits only at the finest granularity. Our work is the first to propose the notion of deferred removal and exploiting it along with unambiguity and common dependence order in practice to speed up test case reduction. It is applicable to any variant of Delta Debugging including HDD.

C. Other Input Minimization Techniques

In general, we can consider any test case reduction technique as related to our work. Among these techniques, Berkeley Delta [35], an approach similar to Delta Debugging that defines line-based subsets to obtain more meaningful reduced versions of an input can also benefit from our work. Other techniques do not use Delta Debugging. They are either domain-specific such as C-Reduce [5], a tool suitable for reducing C/C++ test cases or use other methods such as dynamic tainting [36], searching through file-based subsets of a test case exhaustively [37] and optimizing test suites rather than simplifying test cases [38].

VIII. Conclusion

We identified three reasons why revisiting may not be a critical component of Delta Debugging for test case reduction in practice. We showed that test case reduction can be done with $O(n)$ tests given input size n rather than $O(n^2)$ as in classical Delta Debugging without significant adverse impact on the size of the reduced test case. We also showed that full satisfaction of common dependence order and unambiguity are not required to achieve such results in practice as a result of deferred removal. Results of this study support OPDD as a simpler formulation of Delta Debugging that can be more efficient for reducing large test cases.

References

[1] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss, "What makes a good bug report?," *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, 2010.

[2] C. Pacheco, *Directed random testing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2009.

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[4] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pp. 283–294, 2011.

[5] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pp. 335–346, 2012.

[6] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[7] R. Hodován and Á. Kiss, "Modernizing hierarchical delta debugging," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2016, Seattle, WA, USA, November 18, 2016*, pp. 31–37, 2016.

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

[9] J. Ruderman, *Lithium*. https://www.squarefree.com/lithium.

[10] A. Zeller, "Yesterday, my program worked. today, it does not. why?," in *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, pp. 253–267, 1999.

[11] R. Hodován and Á. Kiss, "Practical improvements to the minimizing delta debugging algorithm," in *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, July 24 - 26, 2016.*, pp. 241–248, 2016.

[12] 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, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pp. 861–871, 2017.

[13] L. Clapp, O. Bastani, S. Anand, and A. Aiken, "Minimizing GUI event traces," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pp. 422–434, 2016.

[14] C. Scott, A. Panda, V. Brajkovic, G. C. Necula, A. Krishnamurthy, and S. Shenker, "Minimizing faulty executions of distributed systems," in *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pp. 291–309, 2016.

[15] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *16th International Symposium on Software Reliability Engineering (ISSRE 2005), 8-11 November 2005, Chicago, IL, USA*, pp. 267–276, 2005.

[16] R. Brummayer and A. Biere, "Fuzzing and delta-debugging smt solvers," in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories, SMT 2009, Montreal, Canada. August 02-03, 2009*, pp. 1–5, 2009.

[17] *The Rust Programming Language*. https://www.rust-lang.org.

[18] *The Go Programming Language*. https://golang.org.

[19] *The DOT Language, Graphviz - Graph Visualization Software*. http://www.graphviz.org/content/dot-language.

[20] *GCC Bug Tracker*. https://gcc.gnu.org/bugzilla.

[21] *Rust Issue Tracker*. https://github.com/rust-lang/rust/issues.

[22] *Golang Issue Tracker*. https://github.com/golang/go/issues.

[23] *Graphviz Issue Tracker*. http://www.graphviz.org/content/graphviz-issue-tracker.

[24] *Csmith*. https://embed.cs.utah.edu/csmith.

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

[26] *The LLVM Compiler Infrastructure Project*. https://clang.llvm.org.

[27] G. Misherghi and Z. Su, "HDD: hierarchical delta debugging," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pp. 142–151, 2006.

[28] G. S. Misherghi, "Hierarchical delta debugging," Master's thesis, University of California Davis, 2007. (Approved).

[29] A. Orso, S. Joshi, M. Burger, and A. Zeller, "Isolating relevant component interactions with jinsi," in *Proceedings of the 2006 international workshop on Dynamic systems analysis, WODA 2006, Shanghai, China. May 23-23, 2006*, pp. 3–10, 2006.

[30] M. Hammoudi, B. Burg, G. Bae, and G. Rothermel, "On the use of delta debugging to reduce recordings and facilitate debugging of web applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pp. 333–344, 2015.

[31] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pp. 417–420, 2007.

[32] R. Hodován and Á. Kiss, "Coarse hierarchical delta debugging," in *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China. September 20-22, 2017*, pp. 194–203, 2017.

[33] N. Bruno, "Minimizing database repros using language grammars," in *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pp. 382–393, 2010.

[34] Z. Lin and X. Zhang, "Deriving input syntactic structure from execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pp. 83–93, 2008.

[35] S. McPeak, D. S. Wilkerson, and S. Goldsmith, *Delta*, July 2003. http://delta.stage.tigris.org/.

[36] J. A. Clause and A. Orso, "Penumbra: automatically identifying failure-relevant inputs using dynamic tainting," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, pp. 249–260, 2009.

[37] J. M. Caron and P. A. Darnell, "Bugfind: A tool for debugging optimizing compilers," *SIGPLAN Notices*, vol. 25, no. 1, pp. 17–22, 1990.

[38] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.