Almost without exception, programs that take some kind of an input, fail sometimes. The more complex the program or the more complex the input format, the higher the chance that at least one corner case is not handled properly. Nowadays, when programming errors have higher probability to have security implications than ever, the quick fixing of a revealed issue is of increased importance. However, before the fixing of an observed failure can start, the faulty spot in the program has to be located, and even before that, the faulty (or rather, fault-inducing) part of the input must be found.

This first step of bug fixing, the minimization of the failure-inducing test case can be very resource consuming if done manually. Two prominent answers to the need for automation are Zeller’s original work on Delta Debugging [26] and its continuation for structured inputs, the Hierarchical Delta Debugging from Miserghi and Su [18]. As real-life minimization scenarios can take hours even with the application of these groundbreaking solutions, research is still ongoing on the field of test case reduction to find possibilities for improvements.

In this paper, we present such an improvement idea, a recursive variant of the hierarchical delta debugging algorithm. Our experiments show that the new algorithm is significantly faster than the original, yielding minimal results in 29–65% less time.

The rest of the paper is organized as follows: first, in Section 2, we give an overview of Delta Debugging and Hierarchical Delta Debugging. Then in Section 3, we describe what we mean by recursive HDD, and in Section 4, we present the detailed results of its evaluation. In Section 5, we discuss related work. Finally, in Section 6, we give a summary of our work and conclude the paper.

1 INTRODUCTION

Almost without exception, programs that take some kind of an input, fail sometimes. The more complex the program or the more complex the input format, the higher the chance that at least one corner case is not handled properly. Nowadays, when programming.
(i.e., the removal of any single unit from the configuration loses the interesting property).

The original definition of the minimizing delta debugging algorithm used a declarative formula, but later, we have given a procedural formulation [11]. The reformulation has helped identify some practically interesting and important aspects of the algorithm, e.g., that the order in which subsets or complements of the configurations are investigated can have significant effect on the performance of the algorithm, or that subset checks can be nearly useless for some input types thus omitting them can make the algorithm faster while still ensuring theoretical 1-minimality. (Note that 1-minimality is not necessarily unique in the general case.)

To make the paper self-contained, Figure 1 gives the procedural formulation of the algorithm. The notations in the algorithm follow the original notations of Zeller, and have the following meaning: TEST is a function that can determine the test outcome for a configuration, returning either \(X, \checkmark, \) or \(?\). The symbols \(\checkmark\) and \(?\) signal a pass or unresolved outcome, respectively, while \(X\) signals a fail outcome, which is actually the only relevant outcome from the perspective of the algorithm. (This notation makes it apparent that DD is rooted in failure-inducing test case minimization. In this paper, we will keep using the \(X\) symbol for historic reasons but will refer to it as an outcome signaling the keeping of an interesting property, which is a more general terminology and broadens the applicability of the test case reduction.) The \(\mathcal{C}_X\) parameter of the algorithm is the input configuration (which is expected to be interesting), and \(\Lambda^i(\mathcal{C}_X, n)\) is the \(i\)th subset of the \(n\)-partitioned \(\mathcal{C}_X\). \(\forall s\) stand for the complement sets, i.e., \(\forall_i(\mathcal{C}_X, n) = \mathcal{C}_X - \Lambda^i(\mathcal{C}_X, n)\).

As mentioned above, DD is typically used on inputs (files) to a program. But inputs are rarely free-form; more often than not, they have to conform to some structured format. The usual line units of DD can work for such inputs, but not necessarily optimally. Structural units of the input format may span multiple lines, may be smaller than a line, or can be completely unaligned with line boundaries. Thus, DD may try to remove such parts of the input that will turn the test syntactically incorrect. Unless syntactic incorrectness is the interesting invariant property of the input, this usually means lots of superfluous steps performed by DD. This observation has lead to the formulation of HDD, the Hierarchical Delta Debugging algorithm [18], which works on a tree (usually, on a parse or abstract syntax tree) representation of the input. The main idea of the algorithm is to progress from the root towards the leaves, level by level, and apply DD to the set of nodes on every level. If HDD is iterated until a fixed-point is reached, denoted as HDD*; it gives a 1-tree-minimal result.

This algorithm is shown, too, in Figure 2. In addition to the already discussed \(\text{DDMIN}\), it uses two helper functions, \(\text{TAGNODIES}\) and \(\text{PRUNE}\), which collect nodes at a given level of a tree and prune those which are not kept by DD, respectively.

Since its introduction, various improvements have been proposed to help the original HDD algorithm, e.g., on how to interpret the pruning of tree nodes (true removal or replacement with minimal syntactically correct fragments) [19], on how to build the input tree (with the help of classic or extended context-free grammars) [10], or on how to preprocess the built tree before passing it to HDD (by squeezing linear segments of the tree or flattening its recursive patterns) [13]. In the next section of this paper, we propose a change, hopefully an improvement, to the algorithm itself.

3 RECURSIVE HDD

First, in Figure 3a, we introduce a simple JavaScript program that calculates the sum and product of the first ten natural numbers, and will act as our running example of test case reduction. We

```
1 procedure DDMIN(c_X)
2 n ← 2
3 out: while true do
4 (* reduce to subset *)
5 forall i in 1..n do
6 if TEST(Δ^i(\mathcal{C}_X, n)) = X then
7 c_X ← Δ^i(\mathcal{C}_X, n)
8 n ← 2
9 continue out
10 end if
11 end forall
12 (* reduce to complement *)
13 forall i in 1..n do
14 if TEST(∀_i(\mathcal{C}_X, n)) = X then
15 c_X ← ∀_i(\mathcal{C}_X, n)
16 n ← max(n − 1, 2)
17 continue out
18 end if
19 end forall
20 (* increase granularity *)
21 if n < |\mathcal{C}_X| then
22 n ← min(⌈|\mathcal{C}_X| / 2n⌉)
23 continue out
24 end if
25 (* done *)
26 break out
27 end while
28 return c'_X
29 end procedure
```

```
1 procedure HDD(input_tree)
2 level ← 0
3 nodes ← TAGNODIES(input_tree, level)
4 while nodes ≠ ∅ do
5 minconfig ← DDMIN(nodes)
6 prune(input_tree, level, minconfig)
7 level ← level + 1
8 nodes ← TAGNODIES(input_tree, level)
9 end while
10 end procedure
```
will input this program to a JavaScript engine and define that the interesting property of the test case is that it executes properly (i.e., it terminates without errors) and prints "prod: 3628800" on its output. This is a variant of the classic example of program slicing [24] and also exemplifies that delta debugging, hierarchical delta debugging, or input reduction in general can not only be used for failure-inducing test case minimization but for other use cases, too. A possible solution of the reduction problem (now, a program slice) is shown in Figure 3b.

Figure 3c shows the tree representation of the example input. (Nodes crossed out mark tokens that de facto cannot be removed, or rather nodes where the actual token text is identical to the associated replacement fragment used during removal. Such nodes can be marked for removal before executing HDD, thus reducing its workload [13].) It points at a peculiarity of HDD, that when processing the tree level by level, it collects nodes representing very different concepts from unrelated parts of the tree into a configuration to be minimized by DD. Thus, as DD has no knowledge of the tree structure anymore when it is partitioning the configuration into subsets, those can easily unalign with the subtree boundaries. Such unaligned subsets typically lead to superfluous steps in the algorithm, which therefore often has to wait until the granularity
reaches the single unit to be able to remove anything. This can also be observed at the 10th level of the example tree. When HDD reaches that level, it collects all six not-yet-removed nodes (circled with dashed lines in the figure) and invokes DDMIN on them. Obviously, only the (root of the) sum += i; statement can be removed, but DDMIN will try to remove both halves of the configuration first (all literals as one subset, and the two statements in the loop plus the argument of the second print as the other). When it does not succeed, it will split the configuration into four parts (one subset will probably include the literal 10 and the summation statement from the loop body, crossing concept boundaries and preventing removal). And only when none of the removal attempts succeed, will it try a partitioning with single nodes, when it finally manages to remove the subtree of the summation.

Based on these observations, we propose to explore the idea of applying DD not to all the nodes at a given level of the tree but to sibling nodes only (but, of course, to all sibling groups, one by one). This can ensure that DD will respect the syntactic boundaries of the format. The most intuitive formalization of this idea is shown in Figure 4, and the recursive nature of this algorithm gives the name of the technique: recursive hierarchical delta debugging, or HDDr. The pseudocode of the algorithm is quite self-explaining as it uses similar concepts to HDD. The helper functions tagChildren and pruneChildren differ from their tagNodes and prune counterparts only in the set of nodes they work on, i.e., on the children of a given node instead of all nodes at a given level. (We acknowledge that the idea has been mentioned briefly by the authors of HDD, too [18], but to our best understanding, no-one has investigated or evaluated the recursive approach ever since, neither in academic papers nor in available implementations.)

As all recursive algorithms can be rewritten to loops, we also give such a reformulation of HDDr in Figure 5. This iterative reformulation of the idea allows us to experiment easier with various traversals of the tree: depending on whether new nodes are popped from the beginning or from the end of the queue (by helper function pop in line 4), the algorithm becomes a breadth-first (BF) or depth-first (DF) traversal, respectively. Moreover, our previous experiences with DD have shown that the order in which the elements of a set (i.e., the subsets of the configuration) are investigated can affect performance, thus the order in which the new nodes are added to the queue (by helper function append in line 8) can also be interesting.

Independent from these variation possibilities — i.e., whether HDDr is implemented in a recursive or iterative way, whether it uses BF or DF traversal, or how it adds nodes to the queue —, HDDr has the same theoretical minimality guarantees as HDD. If the algorithm is iteratively applied to the tree until a fixed-point is reached, let us denote this with HDDr*, it gives a 1-tree-minimal result. That is, no single node (or subtree) can be removed from the tree without losing the interesting property of the tree (or of the corresponding test case). The practical results of the proposed HDDr algorithm are presented in the next section.

4 EXPERIMENTAL RESULTS

To evaluate the idea, we have created a loop-and-queue based prototype implementation of HDDr in the Picireny project\footnote{https://github.com/renatahodovan/picireny}, a Python 3 based open source hierarchical delta debugging framework. The project helped us focus on HDDr, as it already contained solutions for building tree representation for any input format with ANTLR v4 grammar available, as well as an HDD implementation utilizing the DD implementation of the Picire project\footnote{https://github.com/renatahodovan/picire}.

As test inputs, we have selected some examples from the literature that have been previously used for test case reduction benchmarking. Our first test case, sumprod10.js, is our JavaScript example from Section 3, and we reduce it while keeping the correct result of \(\prod_{i=1}^{10} i\) on its output. The second test case, bug.c, is a C program that caused internal compiler error (ICE) in gcc version 2.95.2: it was one of the original examples of DD [9]. The property we want to keep during its reduction is the triggering of the ICE during the compilation of the source. We have also used a Java source implementing the SHA1 algorithm with an artificially injected division-by-zero exception, which was an example to HDD [19]. The property we want to keep during the reduction of SHA1.java is error-free compilation and the occurrence of the erroneous division in the source. And finally, we have borrowed a test case of considerable size from Binkley et al. [1] who applied tree-oriented observation-based slicing (T-ORBS) to a designated code point of a source file of the GNU bash project. We have reconstructed their scenario and reduce expr.c of bash 4.2 while keeping the project compiling and the value of variable val unchanged at line 1393 when executing the arith.tests test case.

\begin{figure}[h]
\centering
\begin{minipage}{.49\textwidth}
\begin{lstlisting}[language=Haskell]
procedure HDDr(root_node)
  nodes ← tagChildren(root_node)
  minconfig ← DDMIN(nodes)
  pruneChildren(root_node, minconfig)
  forall child in minconfig do
    HDDr(child)
  end forall
end procedure
\end{lstlisting}
\caption{The Recursive Hierarchical Delta Debugging Algorithm.}
\end{minipage}\hfill
\begin{minipage}{.49\textwidth}
\begin{lstlisting}[language=Haskell]
procedure HDDr*(root_node)
  queue ← (root_node)
  while queue ≠ () do
    current_node ← pop(queue)
    nodes ← tagChildren(current_node)
    minconfig ← DDMIN(nodes)
    pruneChildren(current_node, minconfig)
    append(queue, minconfig)
  end while
end procedure
\end{lstlisting}
\caption{An iterative reformulation of the Recursive Hierarchical Delta Debugging Algorithm.}
\end{minipage}
\end{figure}
Table 1: Test Sizes

<table>
<thead>
<tr>
<th>Test</th>
<th>Input File</th>
<th>Input Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Chars</td>
<td>Words</td>
</tr>
<tr>
<td>sumprod10.js</td>
<td>136</td>
<td>33</td>
</tr>
<tr>
<td>bug.c</td>
<td>740</td>
<td>115</td>
</tr>
<tr>
<td>SHA1.java</td>
<td>18,804</td>
<td>3,073</td>
</tr>
<tr>
<td>expr.c (bash 4.2)</td>
<td>30,431</td>
<td>4,651</td>
</tr>
</tbody>
</table>

Table 2: Baseline HDD Results

<table>
<thead>
<tr>
<th>Test</th>
<th>HDD</th>
<th>HDD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Steps</td>
<td>Time (s)</td>
</tr>
<tr>
<td>sumprod10.js</td>
<td>111</td>
<td>12.54</td>
</tr>
<tr>
<td>bug.c</td>
<td>132</td>
<td>4.59</td>
</tr>
<tr>
<td>SHA1.java</td>
<td>1,106</td>
<td>878.32</td>
</tr>
<tr>
<td>expr.c (bash 4.2)</td>
<td>21,645</td>
<td>18,799.36</td>
</tr>
</tbody>
</table>

Table 3: HDDr Results

<table>
<thead>
<tr>
<th>Test</th>
<th>HDDr</th>
<th>HDDr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Performance</td>
<td>Output</td>
</tr>
<tr>
<td></td>
<td>Steps</td>
<td>Time (s)</td>
</tr>
<tr>
<td>sumprod10.js</td>
<td>DF BW</td>
<td>63 (57%)</td>
</tr>
<tr>
<td>bug.c</td>
<td>DF FW</td>
<td>66 (62%)</td>
</tr>
<tr>
<td>SHA1.java</td>
<td>BF BW</td>
<td>68 (61%)</td>
</tr>
<tr>
<td>expr.c (bash 4.2)</td>
<td>BF FW</td>
<td>83 (63%)</td>
</tr>
<tr>
<td></td>
<td>DF FW</td>
<td>86 (65%)</td>
</tr>
<tr>
<td></td>
<td>BF FW</td>
<td>89 (67%)</td>
</tr>
<tr>
<td></td>
<td>BF BW</td>
<td>90 (68%)</td>
</tr>
<tr>
<td>SHA1.java</td>
<td>DF BW</td>
<td>404 (37%)</td>
</tr>
<tr>
<td></td>
<td>DF FW</td>
<td>549 (50%)</td>
</tr>
<tr>
<td></td>
<td>BF FW</td>
<td>581 (53%)</td>
</tr>
<tr>
<td></td>
<td>BF BW</td>
<td>582 (53%)</td>
</tr>
<tr>
<td>expr.c (bash 4.2)</td>
<td>DF BW</td>
<td>8,423 (39%)</td>
</tr>
<tr>
<td></td>
<td>DF FW</td>
<td>8,759 (40%)</td>
</tr>
<tr>
<td></td>
<td>BF FW</td>
<td>8,807 (41%)</td>
</tr>
<tr>
<td></td>
<td>BF BW</td>
<td>8,729 (40%)</td>
</tr>
</tbody>
</table>

As the evaluation platform, we have used a dual-socket Supermicro X9DRG-QF machine equipped with 2 Intel Xeon E5-2695 v2 (x86-64) CPUs clocked at 2.40 GHz and 64 GB DDR3 RAM at 1600 MHz. The machine was running Ubuntu 18.04.1 with Linux kernel 4.15.0, the native compiler was gcc 6.4.0, the Java compiler was javac 1.8.0, and the JavaScript execution environment was provided by the JerryScript engine.

For all test cases, we have built the tree representation with the grammar available for the input format (C, Java, and JavaScript) from the official ANTLR v4 grammars repository. Then, before any reduction, we have applied squeezing of linear tree components and flattening of recursive structures to the trees [13]. Thus we got our input trees, for which the size metrics (together with the size metrics of the input files) are presented in Table 1 (for bash, we only give the size of expr.c). As a baseline, we have reduced our test cases with the HDD algorithm. For the sake of reproducibility, we mention that subset checks were omitted [11], and complement sets were checked using backward iteration (i.e., DD tried to remove the last subset of the input configuration first). Moreover, we have cached test outcomes based on the content of the investigated test cases to avoid redundant checks as much as possible [13]. Table 2 gives the details of the baseline reductions, listing information about the execution of HDD (both as number of tests and wall clock time required to reduce the input), and size metrics of the results (both after a single run of the algorithm and after the end of the fixed-point iteration).

The prototype HDDr implementation was executed on the same input trees with the same DD parameters (i.e., subset checks skipped, complement sets iterated backwards, using content-based caching). However, we have varied the parameters of HDDr itself, popping items both from the beginning and from the end of its queue, and appending child nodes both in forward and backward syntactic order to the queue. The ‘pop first’ and ‘forward append’ strategy
Table 3 gives the results of the executions both performance and result size-wise. In addition to the end results, Figure 6 also shows how HDD and HDDr reduced our largest test case, expr.c from bash, step-by-step.

When performing only a single iteration, the depth-first traversal with backward visitation order (i.e., DF BW, or ‘pop last’ with ‘forward append’ strategy) performed the best: it executed 37–63% less testing steps to reach its result, which also took 36–65% less time. Interestingly, it also always gave a smaller output than HDD.

The results of the fixed-point iterated variants are not so unanimous but still promising: except for expr.c, the DF BW variant still performed best requiring 36–61% less testing steps and 37–63% less execution time. However, for the last test case, a breadth-first variant (BF BW) was the fastest (with 50% less steps and 48% less time, where DF BW only saved 36% in steps and 35% in time). The size metrics also show more variance, e.g., all HDDr∗ versions gave somewhat larger results for expr.c than HDD∗. Fortunately, from a practical perspective, that size increase is at most 286 bytes only (making only a 2.5% relative increase), which can be considered as an acceptable trade-off for a 35–48% shorter execution (and that means 4 1/4–6 1/2 hours of wall clock time saved in this case). Notwithstanding the variance in output size, note that all HDD∗ and HDDr∗ results are 1-tree-minimal, they are simply not necessarily unique.

In summary, all variants of HDDr and HDDr∗ outperformed their HDD and HDD∗ counterparts, and the most promising variant (DF BW) could save one to two thirds of testing steps or wall clock time.

5 RELATED WORK

One of the first and most influential work in the field of test case reduction is the Delta Debugging approach introduced by Zeller et al. [9, 27], which can be applied to arbitrary input without having any a priori knowledge about the test case format. In exchange for this flexibility, it generates a large number of syntactically incorrect test cases that lowers its performance. Hodován et al. suggested several speed-up improvements to the original algorithm, like parallelisation or configuration reordering [11], while keeping its guarantee of 1-minimality.

To lower the number of syntactically broken intermediate test cases, Misergi and Su used context-free grammars to preprocess the test cases [18]. They converted the textual inputs to a tree representation and applied the DDMIN algorithm to the levels of the tree. With this approach, called Hierarchical Delta Debugging, they could remove parts that aligned with syntactic unit boundaries. Although it substantially improved DD both output quality and performance-wise, it still created syntactically incorrect test cases as it tried to remove every node even if that caused syntax errors. As an improvement, Misergi analyzed the input grammar to decide which node can be completely removed and which should be replaced with a minimal but syntactically correct string [19]. This change guaranteed the intermediate test cases to be syntactically correct.

The original HDD approach used traditional context-free grammars to parse the input, which could produce highly unbalanced tree representations and cause inefficient reduction. For this reason, Hodován et al. suggested to use extended context-free grammars for tree building [10]. With the help of quantifiers enabled by eCFGs, they got more balanced tree representations and smaller outputs in less time. To facilitate the reuse of available non-extended CFG grammars, they applied automatic transformations to parse trees to balance recursive structures [12]. They also realized that by analyzing the grammars to help avoiding superfluous removals and by using a new caching approach, they could speed up reduction even further [13].

Tree-based test case reduction does not necessarily mean subtree removal. Bruno suggested to use hoisting as an alternative transformation in his framework called SIMP [3], which was designed to reduce database-related inputs. In every reduction step, SIMP tried to replace a node with a compatible descendant. In a follow-up work, they combined SIMP and Delta Debugging [20].

Chengnian et al. combine the above techniques in their Perseus framework [23]. They are also utilizing quantifiers, but instead of parse tree transformations they normalize the grammars by rewriting recursive rules to use quantified expressions instead. During reduction, they apply DDMIN to quantified nodes and hoisting to non-quantified ones. (Since this approach is language independent,
it could have been a reasonable baseline for us, too, but we could not
find the prototype tool publicly available.)

Herfert et al. [8] also combined subtree removal and hoisting in
their Generalized Tree Reduction (GTR) algorithm but instead of
analyzing a grammar to decide about the applicability of a certain
transformation, they learned this information from an existing
test corpus. Regehr et al. also used transformations in their tool
C-Reduce [21], which is used to reduce C/C++ sources, but they
applied language-specific transformations based on the semantics
obtained by Clang.

All the above works targeted textual failure-inducing inputs, but
test case reduction has a much broader application area. Colin et
al. [22] minimized faulty event sequences of distributed systems.
Brummayer et al. [2] used Delta-Debugging in order to minimize
failure-inducing SMT solver formulas. Hammoudi et al. [7] adapted
Delta Debugging to be applicable to bug-inducing web applica-
tion event sequences. Clapp et al. [4] aimed at reducing faulty
Android GUI event sequences with an improved DDMIN variant
called ND3MIN. SimplyDroid [14] also targeted Android GUI event
minimization but it represented input events as a hierarchy tree
and applied HD and two new variants for reduction. Delta De-
bugging was also used to reduce unit tests [15, 16] or even unit test
suites [6].

The efficiency of reduction can be improved with additional
information. The authors of Pneumatic [5] used dynamic tainting
to identify failure-relevant inputs. Wang [25] optimized event trace
minimization by specifying constraints on events and failures. Lin
et al. [17] used lightweight user-feedback information to guide the
recognition of suspicious traces.

6 SUMMARY

In this paper, we have presented a recursive variant of the decade-
old HDD test case reduction algorithm, named HDDr, and we have
also identified that there are at least four ways of parameterising
the new algorithm. After evaluating it on various test cases, we
have found that all parameterisations of HDDr outperform HDD
by requiring at least 29% less testing steps or wall clock time, and
in one case the improvement was higher than 60%. On the largest
test case, where performance was the most important, a variant of
HDDr could yield a 1-treemimal result in half the steps of HDD,
saving several hours of running time. An implementation of the
algorithm was prototyped in the open source Picireny project.

ACKNOWLEDGMENTS

This research was partially supported by the Ministry of Finance
of Hungary under EU-supported Hungarian national grant GINOP-
2.3.2-15-2016-00037.

REFERENCES

[1] David Binkley, Nicolas Gold, Syed Islam, Jens Krinke, and Shun Yoo. 2017. Tree-
Oriented vs. Line-Oriented Observation-Based Slicing. In Proceedings of the 17th
IEEE International Working Conference on Source Code Analysis and Manipulation
Solvers. In Proceedings of the 7th International Workshop on Satisfiability Modula-
In Proceedings of the 10th International Conference on Extending Database Tech-
nology (EDBT ’10). ACM, 382–393.
[4] Lazarro Clapp, Osbert Bastani, Sasiwat Anand, and Alex Aiken. 2016. Minimiz-
ing GUI Event Traces. In Proceedings of the 24th ACM SIGSOFT International
Failure-relevant Inputs Using Dynamic Tainting. In Proceedings of the 18th
International Symposium on Software Testing and Analysis (ISSTA ’09). ACM,
249–260.
[6] Alex Groce, Mohammed Amin Alipour, Chaoqiang Zhang, Yang Chen, and John
International Conference on Software Testing, Verification, and Validation (ICST
2017. Facilitating debugging of web applications through recording reduction.
In Proceedings of the 2000 ACM SIGSOFT International Symposium on Software
Testing and Analysis (ISSTA ’00). ACM, 135–145.
In Proceedings of the 7th International Workshop on Automating Test Case Design,
Delta Debugging Algorithm. In Proceedings of the 11th International Joint Confer-
ence on Software Technologies (ICSOT 2016) – Volume 1: ICSOT-EE. SciTePress,
241–248.
Delta Debugging. 2017 IEEE International Conference on Software Maintenance
and Evolution (ICSM 2017), 194–203.
Test Outcome Caching for Efficient Hierarchical Delta Debugging. In Proceedings
of the 12th IEEE/ACM International Workshop on Automation of Software Testing
Event Sequence Simplification for Android Application. In Proceedings of the
Cases. In Proceedings of the 16th IEEE International Symposium on Software Reliabil-
International Conference on Automated Software Engineering. (ASE ’07). ACM,
417–420.
[17] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-
Based Debugging. In Proceedings of the 39th International Conference on Software
In Proceedings of the 28th International Conference on Software Engineering
(ICSE ’06). ACM, 142–151.
University of California, Davis.
Bug Isolation in DBMS Software. In Proceedings of the 4th International Workshop
[21] John Regehr, Yang Chen, Pascal Cuoco, Eric Eide, Chucky Ellison, and Xuejun
ACM SIGPLAN Conference on Programming Language Design and Implementation
[23] Chengxuan Sun, Yuanbo Li, Qiyun Zhang, Tianxiao Gu, and Zhendong Su. 2018.
Perses: Syntax-guided Program Reduction. In Proceedings of the 40th International
CWI (Centre for Mathematics and Computer Science), Amsterdam, The Nether-
lands.
ACM SIGSOFT International Symposium on Foundations of Software Engineering
In Proceedings of the 7th European Software Engineering Conference Held Jointly
with the 7th ACM SIGSOFT International Symposium on Foundations of Software
Springer-Verlag, 253–267.
[27] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-

22

HDDr: A Recursive Variant of the Hierarchical Delta Debugging Algorithm

A-TEST ’18, November 5, 2018, Lake Buena Vista, FL, USA