Received November 30, 2020, accepted December 1, 2020, date of publication December 7, 2020,
date of current version December 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3043027

Generalizing the Split Factor of the Minimizing Delta Debugging Algorithm

AKOS KISS
Department of Software Engineering, University of Szeged, 6720 Szeged, Hungary
e-mail: akiss@inf.u-szeged.hu

This work was supported in part by the EU-supported Hungarian National Grant GINOP-2.3.2-15-2016-00037, and in part by the Ministry for Innovation and Technology, Hungary, under Grant NKFIH-1279-2/2020. The Open Access APC is covered by the "University of Szeged Open Access Fund" under Grant number 5120.

ABSTRACT One of the first attempts at the automation of test case reduction was the minimizing delta debugging algorithm, widely known as ddmin. Despite its age, it is still an unavoidable cornerstone of this field of research. One criticism against ddmin is that it can take too long to reach the granularity where it can perform actual reduction. Therefore, in this paper, ddmin is generalized with respect to the granularity by introducing a new split factor parameter, leading to the formalization of a parametric algorithm variant. The complexity analysis of this parametric variant reveals that the theoretical worst and best-case behavior of ddmin can be improved. Moreover, the results of experiments with the generalized algorithm show that the reduction can be sped up significantly by choosing the right split factor: up to 84% of the test steps can be eliminated in practice.

INDEX TERMS Delta debugging, granularity, split factor, test case reduction.

I. INTRODUCTION

When a fault is first triggered in a software, the test case that triggers it is rarely concise. This holds true even when the software is used 'properly' and the bug is 'just hit', but it becomes increasingly true for scenarios where the test cases are automatically and randomly generated, e.g., for fuzzing [1]. Trimming down a verbose failure-inducing test case to a minimal subset that is still interesting – i.e., which reproduces the original issue – can be a long and tedious process. So, it is no wonder that attempts have been made at its automation.

One of the first attempts was Zeller's minimizing *delta debugging* algorithm, widely known as *ddmin*. Although the first paper on delta debugging [2] is over twenty years old, its age does not change the fact that it is still an unavoidable reference in the field of automated test case reduction. Many researchers have experimented with different reduction techniques, trying to improve performance [3], [4], aiming for smaller results, or specializing in various input domains [5], [6] – but *ddmin* was always there in the past two decades, either as a basis to build an improved approach upon [3] or as a baseline to compare improvements against [7]. Both its intuitive approach and its theoretical well-foundedness may have led to it becoming such a cornerstone of this field of research.

The associate editor coordinating the review of this manuscript and approving it for publication was Abdullah Iliyasu.

One criticism that is sometimes raised against *ddmin* is that although it is intuitive (i.e., it follows the steps a human tester would take for test case reduction), it can be very expensive in practice [8] (i.e., for the automated approach) because it can take too many steps to reach the *granularity* where it becomes possible to perform actual reduction (more on that in Section II). Fortunately, thanks to the well-formalized definition of the algorithm, it is possible to thoroughly investigate this issue.

Therefore, in this paper, we investigate the original *ddmin* algorithm and generalize it with respect to the granularity it uses, by introducing a new *split factor* parameter. With the help of this generalization, we seek answers to the following research questions:

(RQ1) What components of the original formalization of the *ddmin* algorithm are essential to its theoretical guarantees?
(RQ2) Is the original formalization the fastest to give results, or are there other parameterizations that can be faster?
(RQ3) Does the original formalization give the smallest result, or are there other parameterizations that can yield a smaller output?
(RQ4) Is there a best parameterization for size or for performance?

The rest of this paper is structured as follows: Section II discusses the original minimizing delta debugging algorithm

Algorithm 1 Zeller and Hildebrandt's

Let *test* and $c_{✗}$ be given such that $test(\emptyset) = ✓ \wedge test(c_{✗}) = ✗$ hold.
The goal is to find $c'_{✗} = ddmin(c_{✗})$ such that $c'_{✗} \subseteq c_{✗}$, $test(c'_{✗}) = ✗$, and $c'_{✗}$ is 1-minimal.
The *minimizing Delta Debugging algorithm ddmin*(*c*) is

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

$$ddmin_2(c'_{✗}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = ✗ \text{ (``reduce to subset'')} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = ✗ \text{ (``reduce to complement'')} \\ ddmin_2(c'_{✗}, \min(|c'_{✗}|, 2n)) & \text{else if } n < |c'_{✗}| \text{ (``increase granularity'')} \\ c'_{✗} & \text{otherwise (``done'').} \end{cases}$$

where $\nabla_i = c'_{✗} - \Delta_i$, $c'_{✗} = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{✗}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{✗}) = ✗ \wedge n \leq |c'_{✗}|$.

to give the necessary background information in order to make this paper self-contained. Section III analyzes the original algorithm, generalizes it with the help of a new split factor parameter, and gives the theoretical complexity analysis of the generalized algorithm. Section IV presents the details and the practical results of the experiments conducted with the generalized algorithm, and tries to give answers to the research questions raised above. Section V overviews the related literature, and finally, Section VI concludes the paper with a summary and with directions for future work.

II. BACKGROUND

To give the necessary background information in order to make this paper self-contained, Zeller and Hildebrandt's latest formulation of the minimizing delta debugging algorithm (*ddmin*) [9] is given verbatim in Algorithm 1. (For an overview of other works that are related to test case reduction, but not directly connected to the scope of this paper, the reader is referred to Section V.)

The algorithm takes an initial configuration ($c_{✗}$) and a testing function (*test*) as parameters. The configuration, a set, represents the failure-inducing test case that needs to be minimized and the elements it is composed of. The testing function is used to determine about any subset of the initial configuration whether it induces the same failure (by returning ✗, i.e., *fail* outcome) or not (by returning ✓, i.e., *pass* outcome). (Additionally, the original definition of testing functions allowed for a third type of outcome as well, **?** or *unresolved*, but that outcome type is irrelevant to the algorithm.)

The algorithm assumes that the size of the initial configuration is greater than two (otherwise there would be no minimization necessary), and so it splits up the configuration into two parts. In other words, it starts with the granularity of two (which is kept track of as parameter *n* of the helper function $ddmin_2$). Then, at all granularities, the algorithm tries several steps to make progress. First, hoping for a faster progression, the algorithm tests each subset (Δ_i) of the configuration, for whether any partition resulting from the splitting at the current granularity reproduces the failure. If there is a failing subset, then the rest of the configuration is discarded and the algorithm starts over at the granularity of two (i.e., it splits the failing subset in two halves again and starts their testing). When this "reduce to subset" step is not successful, the algorithm moves on to the "reduce to complement" step and tests each complement ($\nabla_i = c'_{✗} - \Delta_i$). If any of them reproduces the failure, the algorithm starts over again for the failing complement, but this time it only reduces the granularity by one in order to keep the existing splitting. If none of the reduction steps can make progress, the algorithm tries to increase the granularity to work with smaller subsets (which also means larger complements that have a better chance to reproduce the failure). As long as the size of the configuration allows, the increase of granularity is by the factor of two, which can also be interpreted as the further splitting of the current subsets into halves. However, if the granularity cannot be increased further because all subsets already contain a single element of the initial configuration, the algorithm stops.

An important property of *ddmin* is that when it stops, it is guaranteed to have reached a (local) minimum. More precisely, it is proven that its result is 1-minimal according to Definition 1 below.

Definition 1 (n-Minimal Test Case): A test case $c \subseteq c_{✗}$ is *n-minimal* if $\forall c' \subset c \cdot |c| - |c'| \leq n \Rightarrow \big(test(c') \neq ✗\big)$ holds. Consequently, *c* is 1-*minimal* if $\forall \delta_i \in c \cdot test\ (c - \{\delta_i\}) \neq ✗$ holds.

I.e., no single element can be removed from the result of *ddmin* without losing its ability to reproduce the failure.

III. GENERALIZING THE SPLIT FACTOR

As the discussion in the previous section shows, the minimizing delta debugging algorithm (*ddmin*) tries to balance between performance and theoretical guarantees. To speed up the progress of reduction, it splits up configurations into halves, quarters, and so on, before reaching the finest granularity, and it also tries to reduce to these subsets before reducing to complements. However, it is easy to recognize that 1-minimality can be guaranteed by much simpler algorithms as well. All that is actually required is reduction to complements at the finest granularity. In Algorithm 2, the simplest algorithm that can yield 1-minimal results is shown, *onemin*, worded in the likeness of *ddmin*.

Algorithm 2 Trivial 1-Minimizing

Let *test* and $c_{\text{✗}}$ be given such that $test(\emptyset) = \text{✓} \wedge test(c_{\text{✗}}) = \text{✗}$ hold.
The goal is to find $c'_{\text{✗}} = onemin(c_{\text{✗}})$ such that $c'_{\text{✗}} \subseteq c_{\text{✗}}$, $test(c'_{\text{✗}}) = \text{✗}$, and $c'_{\text{✗}}$ is 1-minimal.
The *trivial 1-minimizing algorithm onemin*(c) is

$$onemin(c_{\text{✗}}) = \begin{cases} onemin(\nabla_i) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = \text{✗ (“reduce to complement”)} \\ c_{\text{✗}} & \text{otherwise (“done”).} \end{cases}$$

where $n = |c_{\text{✗}}|$, $\nabla_i = c_{\text{✗}} - \Delta_i$, $c_{\text{✗}} = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| = 1$ holds.
The recursion invariant (and thus precondition) for *onemin* is $test(c_{\text{✗}}) = \text{✗}$.

Algorithm 3 Minimizing Delta Debugging With Generalized Split Factor

Let *test* and $c_{\text{✗}}$ be given such that $test(\emptyset) = \text{✓} \wedge test(c_{\text{✗}}) = \text{✗}$ hold. Let $\nu \geq 2$.
The goal is to find $c'_{\text{✗}} = ddmin^{\nu}(c_{\text{✗}})$ such that $c'_{\text{✗}} \subseteq c_{\text{✗}}$, $test(c'_{\text{✗}}) = \text{✗}$, and $c'_{\text{✗}}$ is 1-minimal.
The *minimizing Delta Debugging algorithm with generalized split factor ddmin*$^{\nu}(c)$ is

$$ddmin^{\nu}(c_{\text{✗}}) = ddmin_2^{\nu}(c_{\text{✗}}, \min(|c_{\text{✗}}|, \nu)) \text{ where}$$

$$ddmin_2^{\nu}(c'_{\text{✗}}, n) = \begin{cases} ddmin_2^{\nu}(\Delta_i, \min(|\Delta_i|, \nu)) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = \text{✗ (“reduce to subset”)} \\ ddmin_2^{\nu}(\nabla_i, \min(|\nabla_i|, \mu^{\nu}(n-1))) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = \text{✗ (“reduce to complement”)} \\ ddmin_2^{\nu}(c'_{\text{✗}}, \min(|c'_{\text{✗}}|, \nu n)) & \text{else if } n < |c'_{\text{✗}}| \text{ (“increase granularity”)} \\ c'_{\text{✗}} & \text{otherwise (“done”).} \end{cases}$$

where $\mu^{\nu}(n) = \begin{cases} n & \text{if } n > 1 \\ \nu & \text{otherwise} \end{cases}$, and $\nabla_i = c'_{\text{✗}} - \Delta_i$, $c'_{\text{✗}} = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{\text{✗}}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2^{\nu}$ is $test(c'_{\text{✗}}) = \text{✗} \wedge n \leq |c'_{\text{✗}}|$.

Obviously, there is a wide range of possible algorithm variants between *onemin* and *ddmin*. In a previous work [4], we have already experimented with discarding the “reduce to subset” step of *ddmin*. In this paper, the granularity of the algorithm is in focus, or more precisely, how the initial granularity is determined and how it is increased later on. The original minimizing delta debugging algorithm always uses the factor of two: at the start of the algorithm, n is set to two, then whenever a “reduce to subset” step succeeds, n becomes two again, and finally, when granularity has to be increased, n is doubled. However, this “magic number” of 2 is not mandatory for the proper behavior of the algorithm. It was used at the inception of *ddmin* because it was presumed that it will make the algorithm efficient and it will help fast(er) progress towards the 1-minimal result.

However, it is exactly this assumption that raises questions, but has not been thoroughly analyzed yet. To facilitate the analysis, this work proposes to generalize *ddmin* by replacing the hard-coded constant of 2 with a parametric split factor. In Algorithm 3, this parametric variant of *ddmin* is given, denoted as *ddmin*$^{\nu}$, where ν (lowercase Greek Nu) stands for the number that determines the initial value as well as the growth factor of n.

It is easy to see that with $\nu = 2$ (the smallest possible value for ν), the parametric variant becomes equivalent to the original *ddmin* algorithm. On the other hand, as ν gets larger, the initial granularity becomes finer, and as soon as it grows beyond the size of the configuration, the subsets of the initial configuration become singletons and the “increase granularity” step effectively vanishes. If we also opt to skip the “reduce to subset” step, as discussed in [4], we get the equivalent of *onemin*.

As this new parameter ν can be used to describe both *ddmin* and the simplest *onemin*, as well as many intermediate variants (all of which guarantee the 1-minimality of the result), *ddmin*$^{\nu}$ can be considered a good generalization of the original algorithm.

Now that it has been formally defined, the complexity analysis of *ddmin*$^{\nu}$ can be performed.

Worst-case complexity: First, the worst case starts with every test having a non-failing result until we have a maximum granularity of $n = |c_{\text{✗}}|$. This results in a re-invocation of $ddmin_2^{\nu}$ with a growing number of subsets (where the multiplication factor is ν), until $|\Delta_i| = 1$ holds. The number of tests to be carried out is $2\nu + 2\nu^2 + 2\nu^3 + \cdots + 2|c_{\text{✗}}| = 2(\frac{|c_{\text{✗}}|}{\nu^0} + \frac{|c_{\text{✗}}|}{\nu^1} + \frac{|c_{\text{✗}}|}{\nu^2} + \ldots) = 2|c_{\text{✗}}|\frac{\nu}{\nu-1}$. Then, the worst case is that testing only the last complement ∇_n results in a failure until $n = 2$ is reached[1]. This means that $ddmin_2^{\nu}$ is re-invoked with $ddmin_2^{\nu}(\nabla_n, n-1)$, resulting in $|c_{\text{✗}}| - 1$ calls of $ddmin_2^{\nu}$ with $2n$ tests per call, or $2(|c_{\text{✗}}| - 1) + 2(|c_{\text{✗}}| - 2) + \cdots + 2 = |c_{\text{✗}}|^2 - |c_{\text{✗}}|$ tests. The overall number of tests is thus $2|c_{\text{✗}}|\frac{\nu}{\nu-1} + |c_{\text{✗}}|^2 - |c_{\text{✗}}| = |c_{\text{✗}}|^2 + \frac{\nu+1}{\nu-1}|c_{\text{✗}}|$.

[1]This second phase of the worst case does not depend on the value of ν, thus this part of the analysis is identical to that given for *ddmin* [9].

Best-case complexity: The best case is when there is only one failure-inducing element $\delta_i \in c_{\chi}$, all test cases that include δ_i cause a failure as well, and testing the first subset in $ddmin_2^{\nu}$ always fails[2]. In this case, the number of tests t is limited by $t \leq log_{\nu}(|c_{\chi}|)$.

With $\nu = 2$, the formulas above give the worst and best-case behavior of *ddmin*, naturally. They also show that by increasing ν, the theoretical worst and best-case complexities both improve (although in the worst case, only the linear component decreases, the cubic part is independent from ν). In practice, however, neither the worst, nor the best case are likely to occur. The next section will investigate the practical effect of the split factor on the performance of $ddmin^{\nu}$.

IV. EXPERIMENTAL RESULTS

To allow experiments with the generalized split factor of the minimizing delta debugging algorithm (*ddmin*), Algorithm 3 has been implemented into the Picire project, a syntax-unaware test case reduction framework written in Python.[3] The evaluation platform of the experiments was a computer equipped with an i5-8250U CPU clocked at 1.6 GHz and with 8 GB RAM. The machine was running Ubuntu 18.04.4 with Linux kernel 4.15.0. The prototype implementation was executed with CPython interpreter version 3.6.9 and the C test cases were compiled with gcc 7.5.0.

A. TEST INPUTS FROM THE LITERATURE

In the first set of experiments, four test inputs have been chosen from the literature that have been previously used for test case reduction benchmarking.

Our first test case (*irrational.json*) is a JSON file containing an array of numbers: integers and floats. An artificial consumer of the input signals error for non-integer elements, and so the reduction criterion (i.e., the property to keep during reduction) is to keep the input as a valid JSON file, but also reproduce the error signal in the consumer. This test case was used as a motivating example in [10].

The second test case (*issue1387.js*) is a JavaScript program that aborts version 1.0 of the JerryScript lightweight JavaScript engine.[4] The reduction criterion for this test case is to keep the crashing behavior of the engine. The test case comes from the public issue tracker of the engine (as #1387) and has also been showcased in [11].

The third test case (*helloworld.c*) is a C program that prints the classic "Hello world!" message among some other lines. In this case, the reduction criterion is to keep the input compilable by a C compiler and also keep the "Hello world!" message on the output of the built binary when executed. This program was an example in [7].

[2]The original best-case complexity analysis of *ddmin* [9] misses to recognize that the best case happens when testing the first subset always results in a failure. Thus, it includes a constant factor of 2 in the upper bound for the number of tests. However, a more precise analysis of the best case allows for a tighter limit.

[3]https://github.com/renatahodovan/picire

[4]https://github.com/jerryscript-project/jerryscript

TABLE 1. Size of test inputs from the literature.

Test Input	Lines	Characters
irrational.json	5	33
issue1387.js	5	74
helloworld.c	10	145
bug.c	37	740

Finally, our fourth test case (*bug.c*) is a C source file that causes an internal compiler error (ICE) in gcc 2.95.2. For this test case, the reduction criterion is to keep the failure-inducing code fragment in the source.[5] This test case has appeared in various papers in slightly different forms [9], [12], the variant that was presented in [3] is used here.

Information about the size of the test cases is given in Table 1. Thus, if we take lines as the units of configurations, we have input configurations in the range of 5 to 37, while with characters as units, the size of input configurations is between 33 and 740.

To evaluate the effect of the split factor on reduction, both line-based and character-based reductions have been performed on all test cases with the split factor following the Fibonacci sequence ([1, 1,]2, 3, 5, 8, 13, ...) as well as the sequence of powers of two ([1,]2, 4, 8, 16, ...), from two until it reached the size of the initial configuration (i.e., the number of lines or characters in the test input). Moreover, two different algorithm variants have been executed for each split factor. In the first variant, both the "reduce to subset" and "reduce to complement" steps of $ddmin_2^{\nu}$ were performed in syntactic order (i.e., the syntactically first subset of the test input was tested first, and then the algorithm iterated over the additional subsets deterministically in sequence, if necessary), and content-based caching was enabled [10]. The second variant also had caching enabled, but the "reduce to subset" step was omitted, and the complement tests were performed in backward syntactic order (i.e., the removal of the syntactically last subset of the test input was tested first, since previous experiences have shown that the order in which the elements of a configuration are investigated can affect performance [4], [13]). In the following paragraphs, we will refer to these variants as "with subset tests", and "without subset tests", respectively[6].

The combination of all test cases, both possible units of configuration (line or character), all split factor settings, and both algorithm variants gave 170 reduction sessions, for which the raw results are presented in Tables 2 and 3. In both tables, multiple lines belong to each test case: one line for the

[5]As gcc 2.95.2 is very hard to put in use on today's systems, a recent compiler is used to ensure that the reduced test case is valid C and substring search is used to ensure that the code fragment that crashed gcc 2.95.2 remains in the source.

[6]Results without caching are not presented, as no implementation of *ddmin* actually behaves this way in practice. The reason for this is that when the "reduce to complement" step is successful, *ddmin* tends to re-test the same subsets in the "reduce to subset" step over and over again. To overcome this performance overhead, even the original work [9] discussed the possibility of caching the test outcomes to avoid the repeated testing of recurring configurations.

TABLE 2. **Results of line-based reduction on test inputs from the literature.**

Test Input	Split Factor (ν)	With Subset Tests		Without Subset Tests	
		Steps	**Output Lines**	**Steps**	**Output Lines**
irrational.json	2	14 (100%)	4 (100%)	9 (100%)	4 (100%)
	3	14 (100%)	4 (100%)	10 (111%)	4 (100%)
	4	*13 (93%)*	4 (100%)	*8 (89%)*	4 (100%)
	5	*11 (79%)*	4 (100%)	*8 (89%)*	4 (100%)
issue1387.js	2	17 (100%)	3 (100%)	9 (100%)	3 (100%)
	3	17 (100%)	3 (100%)	*8 (89%)*	3 (100%)
	4	*16 (94%)*	3 (100%)	*8 (89%)*	3 (100%)
	5	*13 (76%)*	3 (100%)	*6 (67%)*	3 (100%)
helloworld.c	2	41 (100%)	7 (100%)	28 (100%)	7 (100%)
	3	*38 (93%)*	7 (100%)	*24 (86%)*	7 (100%)
	4	*34 (83%)*	7 (100%)	*18 (64%)*	7 (100%)
	5	*36 (88%)*	7 (100%)	*19 (68%)*	7 (100%)
	8	*32 (78%)*	7 (100%)	*23 (82%)*	7 (100%)
	13	*26 (63%)*	7 (100%)	*14 (50%)*	7 (100%)
bug.c	2	122 (100%)	15 (100%)	74 (100%)	15 (100%)
	3	144 (118%)	18 (120%)	95 (128%)	18 (120%)
	4	*99 (81%)*	15 (100%)	*55 (74%)*	15 (100%)
	5	142 (116%)	18 (120%)	*66 (89%)*	18 (120%)
	8	*120 (98%)*	18 (120%)	78 (105%)	18 (120%)
	13	*121 (99%)*	18 (120%)	*63 (85%)*	18 (120%)
	16	129 (106%)	18 (120%)	*66 (89%)*	18 (120%)
	21	131 (107%)	18 (120%)	*68 (92%)*	18 (120%)
	32	139 (114%)	18 (120%)	*64 (86%)*	18 (120%)
	34	140 (115%)	18 (120%)	*66 (89%)*	18 (120%)
	55	123 (101%)	18 (120%)	*59 (80%)*	18 (120%)

results measured with each split factor setting. Then, each line contains two sets of results, one for each algorithm variant, showing both the number of executed testing steps and the size of the output configuration. The first lines (with $\nu = 2$) of the test inputs have been used as baselines (100%) to compare the results of higher split ratios against. To facilitate the reading of the tables, improvements are highlighted with italics and the best improvements with underlines.

Table 2 shows that increasing the split factor could yield faster results for both algorithm variants while giving the same number of output lines (i.e., the same output configuration size) as the baseline. Furthermore, for three out of the four test inputs (*irrational.json*, *issue1387.js*, and *helloworld.c*), $\nu > 2$ was at least as fast as $\nu = 2$ in almost all the cases (the exception being *irrational.json* at $\nu = 3$), and the highest split factor was the fastest. For *bug.c*, the "with subset tests" algorithm variant could not outperform the baseline except at $\nu \in \{4, 8, 13\}$. With the second variant, however, when subset tests were not executed, the results were more similar to the other test cases (but the best performance was still observed at the split factor of four, i.e., not at the highest split factor). In summary, for line-based reduction, the increase of the split factor could save 19–37% and 11–50% of the test steps of the "with subset tests" and "without subset tests" variants, respectively, in the best case.

In addition to the results above, there are two additional observations that can be made based on Table 2. First, the effect of the split factor on performance is not monotonic in practice, neither in general, nor for a given test case, nor for an algorithm variant: as the split factor increases, the number of executed tests sometimes increases, sometimes decreases. I.e., even if the theoretical upper and lower bounds of the number of executed tests decrease with an increasing split factor, this improvement does not necessarily manifest itself for every test case. Second, *bug.c* exemplifies that 1-minimal configurations are not unique. For that test case, some higher split factors yielded somewhat larger output configurations – however, it must be noted that even the larger outputs comply with the definition of 1-minimality.

The results of character-based reduction, shown in Table 3, partially align with the line-based results. Most notably, for both algorithm variants, and for all test cases, some higher split factor is always faster than the baseline. However, in line with the observation made above about non-monotonicity in practice, the fastest execution is not necessarily measured at the highest split factor, but sometimes at some quite different intermediate values (at $\nu = 5$ with subset tests and at $\nu = 3$ without subset tests for *irrational.json*, and at $\nu = 32$ with subset tests and at $\nu = 144$ without subset tests for *bug.c*). It is also true that for both variants, almost all higher split factors outperformed the baseline (with the exception of *irrational.json* at $\nu = 8$, at $\nu = 21$, and at $\nu = 32$, and *issue1387.js* at $\nu = 4$). Finally, the character-based reductions show the non-uniqueness of 1-minimality more prominently: *bug.c* is an outlier again, but now in a positive sense, as it gives a *smaller* output at $\nu = 3$ than the baseline, showing that the original approach of $\nu = 2$ is not necessarily the optimal parameterization either for performance or for size. As a summary of the

TABLE 3. Results of character-based reduction on test inputs from the literature.

Test Input	Split Factor (ν)	With Subset Tests		Without Subset Tests	
		Steps	Output Chars	Steps	Output Chars
irrational.json	2	95 (100%)	5 (100%)	66 (100%)	5 (100%)
	3	*60 (63%)*	5 (100%)	*34 (52%)*	5 (100%)
	4	*77 (81%)*	5 (100%)	*55 (83%)*	5 (100%)
	5	*56 (59%)*	7 (140%)	*41 (62%)*	7 (140%)
	8	*86 (91%)*	11 (220%)	71 (108%)	11 (220%)
	13	*76 (80%)*	7 (140%)	*43 (65%)*	7 (140%)
	16	*85 (89%)*	7 (140%)	*57 (86%)*	7 (140%)
	21	102 (107%)	11 (220%)	71 (108%)	11 (220%)
	32	*83 (87%)*	11 (220%)	73 (111%)	11 (220%)
	34	*71 (75%)*	11 (220%)	*63 (95%)*	11 (220%)
issue1387.js	2	255 (100%)	39 (100%)	173 (100%)	39 (100%)
	3	*197 (77%)*	39 (100%)	*121 (70%)*	39 (100%)
	4	258 (101%)	43 (110%)	*169 (98%)*	43 (110%)
	5	*168 (66%)*	39 (100%)	*110 (64%)*	39 (100%)
	8	*228 (89%)*	39 (100%)	*115 (66%)*	39 (100%)
	13	*160 (63%)*	43 (110%)	*126 (73%)*	43 (110%)
	16	*160 (63%)*	43 (110%)	*119 (69%)*	43 (110%)
	21	*176 (69%)*	43 (110%)	*129 (75%)*	43 (110%)
	32	*195 (76%)*	43 (110%)	*129 (75%)*	43 (110%)
	34	*193 (76%)*	43 (110%)	*129 (75%)*	43 (110%)
	55	*222 (87%)*	43 (110%)	*122 (71%)*	43 (110%)
	64	*224 (88%)*	43 (110%)	*130 (75%)*	43 (110%)
	89	*134 (53%)*	43 (110%)	*84 (49%)*	43 (110%)
helloworld.c	2	572 (100%)	81 (100%)	508 (100%)	81 (100%)
	3	*482 (84%)*	85 (105%)	*425 (84%)*	85 (105%)
	4	*440 (77%)*	92 (114%)	*411 (81%)*	92 (114%)
	5	*489 (85%)*	85 (105%)	*476 (94%)*	85 (105%)
	8	*395 (69%)*	81 (100%)	*337 (66%)*	81 (100%)
	13	*378 (66%)*	92 (114%)	*308 (61%)*	92 (114%)
	16	*383 (67%)*	92 (114%)	*311 (61%)*	92 (114%)
	21	*355 (62%)*	86 (106%)	*342 (67%)*	86 (106%)
	32	*357 (62%)*	92 (114%)	*367 (72%)*	92 (114%)
	34	*355 (62%)*	92 (114%)	*364 (72%)*	92 (114%)
	55	*378 (66%)*	92 (114%)	*338 (67%)*	92 (114%)
	64	*401 (70%)*	92 (114%)	*391 (77%)*	92 (114%)
	89	*400 (70%)*	92 (114%)	*357 (70%)*	92 (114%)
	128	*428 (75%)*	92 (114%)	*369 (73%)*	92 (114%)
	144	*442 (77%)*	92 (114%)	*383 (75%)*	92 (114%)
	233	*352 (62%)*	92 (114%)	*295 (58%)*	92 (114%)
bug.c	2	3119 (100%)	318 (100%)	2591 (100%)	317 (100%)
	3	*2284 (73%)*	*269 (85%)*	*1688 (65%)*	*269 (85%)*
	4	*2204 (71%)*	362 (114%)	*1700 (66%)*	361 (114%)
	5	*2458 (79%)*	320 (101%)	*1820 (70%)*	321 (101%)
	8	*2628 (84%)*	369 (116%)	*2027 (78%)*	368 (116%)
	13	*2270 (73%)*	345 (108%)	*1611 (62%)*	346 (109%)
	16	*2606 (84%)*	388 (122%)	*1795 (69%)*	389 (123%)
	21	*2035 (65%)*	330 (104%)	*1849 (71%)*	336 (106%)
	32	*1705 (55%)*	378 (119%)	*1729 (67%)*	372 (117%)
	34	*2080 (67%)*	408 (128%)	*1582 (61%)*	402 (127%)
	55	*2073 (66%)*	399 (125%)	*1742 (67%)*	393 (124%)
	64	*1838 (59%)*	380 (119%)	*1862 (72%)*	373 (118%)
	89	*1731 (55%)*	369 (116%)	*1526 (59%)*	368 (116%)
	128	*2322 (74%)*	393 (124%)	*1406 (54%)*	389 (123%)
	144	*2121 (68%)*	392 (123%)	*1316 (51%)*	389 (123%)
	233	*2405 (77%)*	390 (123%)	*1660 (64%)*	383 (121%)
	256	*2574 (83%)*	388 (122%)	*1779 (69%)*	389 (123%)
	377	*2188 (70%)*	378 (119%)	*1709 (66%)*	384 (121%)
	512	*2457 (79%)*	395 (124%)	*1844 (71%)*	397 (125%)
	610	*2585 (83%)*	402 (126%)	*1873 (72%)*	403 (127%)
	987	*2015 (65%)*	408 (128%)	*1548 (60%)*	402 (127%)

character-based reduction, the parametric split factor helped to execute 38–47% and 42–51% fewer tests in the "with subset tests" and "without subset tests" variants in the best case, respectively.

B. TEST INPUTS FROM JavaScript FUZZING

In a second set of experiments, 13 test cases have been compiled from fuzzing sessions targeting the aforementioned JerryScript engine. The test inputs (i.e., JavaScript programs)

TABLE 4. Size of test inputs from JavaScript fuzzing.

Test Input	Lines	Characters
#3299	71	1901
#3361	55	1733
#3376	232	6323
#3408	66	2425
#3431	67	978
#3433	65	1005
#3437	232	6300
#3479	127	4597
#3483	20	444
#3506	137	3309
#3523	179	3626
#3534	87	1773
#3536	37	895

were generated by the Grammarinator fuzzer [14] and split to lines using a code reformatter. The thus created test scripts crash various development revisions of the engine with heap buffer overflows, stack buffer overflows, and assertion failures. The detected bugs have been reported in the issue tracker of the engine: the first column of Table 4 gives the issue IDs assigned to the reports (which detail the reproducibility of the problem by giving the affected engine revision, build steps, and observed failure, as well as a manually minimized version of the test case).

The size of the non-minimized test cases is also given in Table 4. In units of lines, the size of the inputs is in the range of 20 to 232. In terms of characters, the size of the test cases is between 444 and 6323. (However, this latter metric is solely given for reference as this second set of experiments only involves line-based reduction because of resource constraints.)

The results of the line-based reduction of these 13 test inputs are shown in Table 5 in a format similar to the one used in the previous subsection. The only difference is that results are not given for all split factors because of the large amount of data (the combination of test cases, split factor values, and algorithm variants gave 344 reduction sessions). The results for the lowest and highest split factors are given for each test input, but intermediate lines are only shown for the best performing split factors of each algorithm variant (if there even was a parameterization that was at least as good as $\nu = 2$ at all).

Again, these results partially align with the results seen in the previous subsection. For the ''with subset tests'' and ''without subset tests'' algorithm variants and for the majority of these test inputs, there is a split factor greater than two that can reach the performance or even outperform the baseline split factor of two. It were only the test inputs of #3299 and #3433 (for the ''with subset tests'' variant), and the test input of #3534 (for the ''without subset tests'' variant) for which every tried split factor was slower than the baseline. (Note, however, that because of the non-monotonic effect of the split factor on the test steps, there may be better parameterizations, even for these test inputs. But it was impractical to try every possible value for ν.) The best value for the split factor varied heavily, the fewest test steps were observed at $\nu \in \{3, 4, 5, 8, 16, 21, 55, 89, 128\}$ depending on the test input and the algorithm variant. These tests also show that a higher split factor can not only decrease the number of performed test steps, but in some cases, it can also result in a smaller output than the baseline (exemplified by issues #3376, #3408, #3506, and #3536, with 80% being the highest reduction observed). In summary, where improvement was possible, a higher split factor could save 1–84% and 4–63% of the test steps of the ''with subset tests'' and ''without subset tests'' algorithm variants.

C. ANSWERS TO THE RESEARCH QUESTIONS

The algorithms in Section III and the experimental results presented above help us answer the research questions raised at the beginning of this work.

(RQ1) What components of the original formalization of the ddmin algorithm are essential to its theoretical guarantees?

The trivial 1-minimizing algorithm, *onemin*, shows that, theoretically, it is only the ''reduce to complement'' step at the finest granularity that is essential. Nevertheless, the minimizing delta debugging algorithm with the generalized split factor, $ddmin^{\nu}$, shows that the granularity-related components of the formalization can be changed to improve performance without violating the theoretical guarantees.

(RQ2) Is the original formalization the fastest to give results, or are there other parameterizations that can be faster?

From a theoretical perspective, the complexity analysis of $ddmin^{\nu}$ shows that changing (increasing) the here-introduced split factor parameter has a positive effect on both the worst and best-case behavior of the algorithm. From a practical point of view, the experimental results also show that the split factor parameter has a positive, although non-monotonic, effect on the performance of the reduction in the majority of the cases. In the above described experiments, up to 84% of the test executions could be eliminated. The experimental results also confirm our earlier findings [4] that the omission of the ''reduce to subset'' step of delta debugging can speed up the reduction process for most of the test cases.

(RQ3) Does the original formalization give the smallest result, or are there other parameterizations that can yield a smaller output?

According to the experimental results, the original formalization often gives the smallest results, but not always. For some test cases, changing the split factor parameter could decrease the output size along with the number of test steps. In one case, a surprisingly high output size reduction was observed (80%). Note, however, that theoretically all outputs are 1-minimal.

(RQ4) Is there a best parameterization for size or for performance?

Because of the non-monotonic effect of the split factor in practice, the best value for ν is not always the highest possible value (as possibly suggested by the complexity analysis of $ddmin^{\nu}$). At the moment, there seems to be no fixed value or

TABLE 5. Results of line-based reduction on test inputs from JavaScript fuzzing.

Test Input	Split Factor (ν)	With Subset Tests		Without Subset Tests	
		Steps	Output Lines	Steps	Output Lines
#3299	2	157 (100%)	15 (100%)	91 (100%)	15 (100%)
	8	170 (108%)	16 (107%)	*87 (96%)*	16 (107%)
	89	170 (108%)	25 (167%)	96 (105%)	25 (167%)
#3361	2	67 (100%)	5 (100%)	37 (100%)	5 (100%)
	4	*66 (99%)*	5 (100%)	*34 (92%)*	5 (100%)
	55	139 (207%)	14 (280%)	69 (186%)	14 (280%)
#3376	2	493 (100%)	34 (100%)	355 (100%)	34 (100%)
	3	*291 (59%)*	*29 (85%)*	*202 (57%)*	*29 (85%)*
	233	684 (139%)	80 (235%)	392 (110%)	80 (235%)
#3408	2	202 (100%)	20 (100%)	139 (100%)	20 (100%)
	3	*90 (45%)*	*7 (35%)*	*51 (37%)*	*7 (35%)*
	5	*33 (16%)*	*4 (20%)*	*108 (78%)*	22 (110%)
	89	*135 (67%)*	24 (120%)	*90 (65%)*	24 (120%)
#3431	2	108 (100%)	12 (100%)	66 (100%)	12 (100%)
	3	*77 (71%)*	12 (100%)	99 (150%)	19 (158%)
	4	*86 (80%)*	12 (100%)	*53 (80%)*	12 (100%)
	89	167 (155%)	26 (217%)	92 (139%)	26 (217%)
#3433	2	71 (100%)	5 (100%)	40 (100%)	5 (100%)
	8	72 (101%)	5 (100%)	40 (100%)	5 (100%)
	89	117 (165%)	8 (160%)	73 (183%)	8 (160%)
#3437	2	184 (100%)	1 (100%)	230 (100%)	25 (100%)
	4	*169 (92%)*	1 (100%)	*191 (83%)*	37 (148%)
	89	*62 (34%)*	1 (100%)	322 (140%)	61 (244%)
	233	*100 (54%)*	1 (100%)	309 (134%)	79 (316%)
#3479	2	466 (100%)	33 (100%)	228 (100%)	33 (100%)
	16	*310 (67%)*	36 (109%)	*150 (66%)*	36 (109%)
	128	*308 (66%)*	44 (133%)	*171 (75%)*	44 (133%)
#3483	2	19 (100%)	3 (100%)	33 (100%)	6 (100%)
	4	*15 (79%)*	3 (100%)	*27 (82%)*	6 (100%)
	5	38 (200%)	6 (200%)	*26 (79%)*	6 (100%)
	21	44 (232%)	6 (200%)	*26 (79%)*	6 (100%)
#3506	2	44 (100%)	3 (100%)	193 (100%)	30 (100%)
	3	50 (114%)	3 (100%)	*120 (62%)*	*19 (63%)*
	8	*39 (89%)*	3 (100%)	*141 (73%)*	30 (100%)
	144	273 (620%)	48 (1600%)	*184 (95%)*	48 (160%)
#3523	2	464 (100%)	42 (100%)	321 (100%)	43 (100%)
	16	*318 (69%)*	59 (140%)	*231 (72%)*	69 (160%)
	21	*312 (67%)*	66 (157%)	*238 (74%)*	66 (153%)
	233	*354 (76%)*	84 (200%)	*262 (82%)*	84 (195%)
#3534	2	36 (100%)	4 (100%)	25 (100%)	4 (100%)
	8	*26 (72%)*	4 (100%)	125 (500%)	27 (675%)
	89	167 (464%)	43 (1075%)	129 (516%)	43 (1075%)
#3536	2	128 (100%)	15 (100%)	83 (100%)	15 (100%)
	5	*75 (59%)*	*13 (87%)*	*52 (63%)*	*13 (87%)*
	55	*84 (66%)*	17 (113%)	*49 (59%)*	17 (113%)

obvious formula that could give the best split factor (size or performance-wise), yet.

V. RELATED WORK

One of the first and most influential works in the field of test case reduction is the delta debugging approach (*ddmin*) introduced by Zeller [2], Zeller and Hildebrandt [12], Hildebrandt and Zeller [9]. It can be applied to arbitrary inputs 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 parallelization or configuration reordering [4], while keeping its guarantee of 1-minimality. Other approaches used static and dynamic analysis, or slicing to discover semantic relationships in the code under inspection and reduce the search space where the failure is located [15], [16].

To lower the number of syntactically broken intermediate test cases, Miserghi and Su used context-free grammars to preprocess the test cases [3]. They converted the textual inputs into a tree representation and applied the *ddmin* algorithm to the levels of the tree. With this approach, called Hierarchical Delta Debugging (HDD), they could remove parts that aligned with syntactic unit boundaries. Although it substantially improved delta debugging 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, Miserghi analyzed the input grammar to decide which node can be completely removed and which should be replaced with a minimal, but syntactically correct string [17]. 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 using extended context-free grammars (eCFGs) for tree building [18]. 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 [11]. They also realized that by analyzing the grammars to help avoid superfluous removals and by using a new caching approach, they could speed up reduction even further [10]. Moreover, they experimented with a variant of HDD, called HDDr or recursive HDD, that applied *ddmin* to the children of one node at a time, traversing the tree either in a depth-first or breadth-first way [13].

Tree-based test case reduction does not necessarily have to mean subtree removal. Bruno suggested to use hoisting as an alternative transformation in his framework called SIMP [19], 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 [5].

Sun *et al.* combine the above mentioned techniques in their Perses framework [7]. 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 maintain an ordered worklist of the pending nodes to be reduced. The ordering of the worklist follows the number of tokens that a certain node contains. When reducing, they apply *ddmin* to the quantified nodes and hoisting to the non-quantified ones. Pardis [20], from Gharachorlu *et al.*, is built upon the idea of Perses, but it uses a different approach to prioritize the ordering of the worklist.

Herfert *et al.* [21] 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.* used delta debugging as one possible method in their C-Reduce test case reducer tool for C/C++ sources [22]. This system contains various source-to-source transformator plugins – line-based delta debugging among others – to mimic the steps that a human would have taken. They also applied language-specific transformations based on the semantics obtained by the Clang compiler. Regehr also experimented with running the plugins of C-Reduce in parallel, the write-up about this work is available on his blog [23].

All the above-mentioned works targeted textual failure-inducing inputs, but test case reduction has a much broader application area. Scott *et al.* [24] minimized faulty event sequences of distributed systems. Brummayer and Biere [25] used delta debugging in order to minimize failure-inducing satisfiability modulo theories (SMT) solver formulas. Hammoudi *et al.* [26] adapted delta debugging to be applicable to bug-inducing web application event sequences. Clapp *et al.* [27] aimed at reducing faulty Android graphical user interface (GUI) event sequences with an improved *ddmin* variant called ND3MIN. SimplyDroid [28] also targeted Android GUI event minimization, but it represented input events as a hierarchy tree and applied HDD and two new variants for reduction. Delta debugging was also used to reduce unit tests [15], [29] or even unit test suites [30]. Binkley *et al.* used similar but non-(H)DD-based algorithms in their observation-based slicing approaches [6], [8].

The efficiency of reduction can be improved with additional information. The authors of Penumbra [31] used dynamic tainting to identify failure-relevant inputs. Wang [32] optimized event trace minimization by specifying constraints on events and failures. Lin *et al.* [33] used lightweight user-feedback information to guide the recognition of suspicious traces.

VI. SUMMARY

In this paper, we have focused on the minimizing delta debugging algorithm (*ddmin*), which – despite its age – is still one of the most important works in the field of automated test case reduction. One criticism that is sometimes raised against *ddmin* is that it can take too long for it to reach the granularity where it can perform actual reduction. However, after investigation, it became clear that there is a parameterization possibility of *ddmin*, not yet analyzed or utilized: there is a split factor built into the algorithm that affects the granularity used during reduction, which – although originally defined as the constant value of two – may be varied without losing the guarantee of 1-minimal results. A parametric algorithm variant has been formalized, $ddmin^{\nu}$, and it has been shown to be able to express multiple existing approaches, e.g., both the original *ddmin* and the trivial 1-minimizing algorithm, *onemin*. Its complexity analysis has revealed that the theoretical worst and best-case behavior improves with the increase of the split factor parameter. A large number of reduction sessions have been conducted, and it has been found that by choosing the best split factor, the reduction could be sped up by eliminating up to 84% of the test steps. However, it has also been found that determining the best value for the split factor is non-trivial in practice, i.e., there is no known formula for it for the time being.

Whether the best-performing split factor can always be found using a formula or guessed with sufficient accuracy using some heuristics, is left for a future work. Additionally, the work presented here has raised further research questions related to algorithms built on top of *ddmin*. Thus, we have plans to conduct a similar experiment with hierarchical delta debugging. It will be interesting to see what effect the split factor can have on tree-structured configurations, which help to create subset partitions better aligned with the input structure. Finally, it could also be investigated in follow-up research whether other test case reduction algorithms not directly related to *ddmin* could adopt the concept of a generalized split factor.

ACKNOWLEDGMENT

The author is grateful to R. for the introduction to this field of research, to T. for all the discussions, and to M. for being a continuous motivation during the course of the work.

REFERENCES

[1] A. Takanen, J. DeMott, C. Miller, and A. Kettunen, *Fuzzing for Software Security Testing and Quality Assurance*, 2nd ed. Norwood, MA, USA: Artech House, 2018.

[2] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Proc. 7th Eur. Softw. Eng. Conf. Held Jointly With, 7th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (ESEC/FSE)*, in Lecture Notes in Computer Science, vol. 1687. Berlin, Germany: Springer, Sep. 1999, pp. 253–267.

[3] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proc. 28th Int. Conf. Softw. Eng. (ICSE)*, May 2006, pp. 142–151.

[4] R. Hodován and Á. Kiss, "Practical improvements to the minimizing delta debugging algorithm," in *Proc. 11th Int. Joint Conf. Softw. Technol.*, vol. 1, 2016, pp. 241–248.

[5] K. Morton and N. Bruno, "FlexMin: A flexible tool for automatic bug isolation in DBMS software," in *Proc. 4th Int. Workshop Test. Database Syst. (DBTest)*, 2011, pp. 1:1–1:6.

[6] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "ORBS: Language-independent program slicing," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2014, pp. 109–120.

[7] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*, 2018, pp. 361–371.

[8] D. Binkley, N. Gold, S. Islam, J. Krinke, and S. Yoo, "Tree-oriented vs. line-oriented observation-based slicing," in *Proc. IEEE 17th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2017, pp. 21–30.

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

[10] R. Hodovan, A. Kiss, and T. Gyimothy, "Tree preprocessing and test outcome caching for efficient hierarchical delta debugging," in *Proc. IEEE/ACM 12th Int. Workshop Automat. Softw. Test. (AST)*, May 2017, pp. 23–29.

[11] R. Hodován, A. Kiss, and T. Gyimóthy, "Coarse hierarchical delta debugging," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 194–203.

[12] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, Aug. 2000, pp. 135–145.

[13] Á. Kiss, R. Hodován, and T. Gyimóthy, "HDDr: A recursive variant of the hierarchical delta debugging algorithm," in *Proc. 9th ACM SIGSOFT Int. Workshop Automating TEST Case Design, Selection, Eval. (A-TEST)*, 2018, pp. 16–22.

[14] R. Hodován, A. Kiss, and T. Gyimóthy, "Grammarinator: A grammar-based open source fuzzer," in *Proc. 9th ACM SIGSOFT Int. Workshop Automating Test Case Design, Selection, Evaluation (A-TEST)*, Nov. 2018, pp. 45–48. [Online]. Available: https://github.com/renatahodovan/grammarinator.

[15] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2007, pp. 417–420.

[16] M. Burger and A. Zeller, "Minimizing reproduction of software failures," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2011, pp. 221–231.

[17] G. S. Misherghi, "Hierarchical delta debugging," M.S. thesis, Univ. California, Davis, Davis, CA, USA, Jun. 2007.

[18] R. Hodován and Á. Kiss, "Modernizing hierarchical delta debugging," in *Proc. 7th Int. Workshop Automating Test Case Design, Selection, Eval. (A-TEST)*, 2016, pp. 31–37.

[19] N. Bruno, "Minimizing database repros using language grammars," in *Proc. 13th Int. Conf. Extending Database Technol. (EDBT)*, Mar. 2010, pp. 382–393.

[20] G. Gharachorlu and N. Sumner, "Pardis: Priority aware test case reduction," in *Proc. 22nd Int. Conf. Fundam. Approaches Softw. Eng. (FASE)*, in Lecture Notes in Computer Science, vol. 11424. Cham, Switzerland: Springer, Apr. 2019, pp. 409–426.

[21] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree-structured test inputs," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Oct. 2017, pp. 861–871.

[22] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2012, pp. 335–346.

[23] J. Regehr. (Jul. 2012). *Parallelizing Delta Debugging*. Accessed: Sep. 27, 2019. [Online]. Available: https://blog.regehr.org/archives/749

[24] C. Scott, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker, "Minimizing faulty executions of distributed systems," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2016, pp. 291–309.

[25] R. Brummayer and A. Biere, "Fuzzing and delta-debugging SMT solvers," in *Proc. 7th Int. Workshop Satisfiability Modulo Theories (SMT)*, 2009, pp. 1–5.

[26] M. Hammoudi, A. Alakeel, B. Burg, G. Bae, and G. Rothermel, "Facilitating debugging of Web applications through recording reduction," *Empirical Softw. Eng.*, vol. 23, no. 6, p. 3821, May 2017.

[27] L. Clapp, O. Bastani, S. Anand, and A. Aiken, "Minimizing GUI event traces," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2016, pp. 422–434.

[28] B. Jiang, Y. Wu, T. Li, and W. K. Chan, "SimplyDroid: Efficient event sequence simplification for Android application," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Oct. 2017, pp. 297–307.

[29] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *Proc. 16th IEEE Int. Symp. Softw. Rel. Eng. (ISSRE)*, 2005, pp. 267–276.

[30] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction for quick testing," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation (ICST)*, Mar. 2014, pp. 243–252.

[31] J. Clause and A. Orso, "Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting," in *Proc. 18th Int. Symp. Softw. Test. Anal. (ISSTA)*, 2009, pp. 249–260.

[32] J. Wang, "Constraint-based event trace reduction," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2016, pp. 1106–1108.

[33] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong, "Feedback-based debugging," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 393–403.

AKOS KISS was born in Mohács, Hungary, in 1977. He received the M.Sc. and Ph.D. degrees in computer science from the University of Szeged, Szeged, Hungary, in 2000 and 2009, respectively.

From 2000 to 2001, he was a Research Assistant with the Research Group on Artificial Intelligence, University of Szeged, and the Hungarian Academy of Sciences. From 2004 to 2009, he was an Assistant Lecturer with the Department of Software Engineering, University of Szeged, where he has been an Assistant Professor, since 2010. He is the author of more than 40 publications and of two patents. His research interests include program analysis, program slicing, random (a.k.a. fuzz) testing, and test case reduction, with a strong emphasis on embedded systems and the IoT.

Dr. Kiss has been the member of the Hungarian John von Neumann Computer Society (NJSZT), since 2005, and the Secretary of its Csongrád County Organization, since 2006.