Enhancements of Automated Test Case Reduction

Summary of the Ph.D. Dissertation

by

Dániel Vince

Supervisor:
Dr. Ákos Kiss

Doctoral School of Informatics
Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged
2024

Introduction

Our modern lives are driven by software. It has become so obvious that we often do not even notice the programs around us; they collect and process data and then tell us to get up and walk around for a few minutes during our long office hours. In an ideal world, the software is perfect, the business logic fully covers the customer's expectations and produces the right output for every possible combination of inputs so it "cannot be fooled".

Unfortunately, we do not live in an ideal world. We also know from Murphy's Law that if something can go wrong, it will. Most likely at the worst possible time. And then, we will notice – or worse, remember – the software, and its failure. In the world of software development, this can set off a chain of events that ends with an engineer being given the noble task of fixing the bug.

The first challenge on this not-so-easy path is to find the relevant part of the input that is responsible for the error. If that input is "large" (or noisy), then this activity can consume valuable human time. Although it is possible to perform this task manually, we should automate it, at least to increase our own productivity. Consider a safety-critical project that is continuously tested by a random test case generator every single hour of the day, and then, automatically reports the bugs it finds. In such a case, it is really expensive to spend engineering time searching for the minimal replication package.

To reduce the cost, the discipline of automatic test case reduction researches algorithms that can reduce any kind of input to a smaller, some kind of "minimal" form, while maintaining a well-defined condition. The most well-known of these algorithms are minimizing Delta Debugging (DDMIN) and Hierarchical Delta Debugging (HDD), which are discussed in detail in this study. Several functional and non-functional properties and their improvements are detailed in the following booklet. Some of them result in smaller output, while others aim to reduce the memory footprint or the time required by the process.
The author identifies four main findings, which are listed below:

1. **Cache Optimizations**: Defined, implemented, and evaluated three cache optimizations for automatic test case reduction algorithms,

2. **Iterating the Minimizing Delta Debugging Algorithm**: Defined, implemented, and evaluated the fixed-point iteration of DDMIN that achieves smaller outputs,

3. **Parallel Optimizations of DDMIN***: Defined, implemented, and evaluated a new parallel variant of the DDMIN algorithm that can perform reduction faster than before,

4. **Extending Hierarchical Delta Debugging with Hoisting**: Defined, implemented, and evaluated the transformation-based minimization framework for hierarchical reduction, and an example transformation, the Hoisting.

1 Cache Optimizations

The main purpose of using cache memory in test case reduction algorithms is to avoid running the same test multiple times as the algorithm tries different configuration combinations. On the other hand, the reduction should somehow be completed, even if we run out of resources (RAM). Several techniques are available for cache replacement, the most widespread algorithms for it are Least Frequently Used (LFU), Most Frequently Used (MFU), and Least Recently Used (LRU), however, these classic techniques do not make use of the knowledge of the underlying algorithms, and evict elements from the cache that might be needed later. Therefore, the caching solutions of minimizing Delta Debugging (DDMIN) and Hierarchical Delta Debugging (HDD) were investigated, and based on the preliminary research, the "content-based" technique was chosen to work on, as it performed best with both algorithms.

The basic concept of DDMIN is that when it finds a *failing* configuration that results in a serialized test case of size n, it starts a new iteration with that to reduce it further. After that, configurations that result in test cases larger than n would not be tested, since the new iteration splits that configuration into smaller fragments. This observation can be written as follows, using the notation introduced in the publication of DDMIN [12]:

$$\begin{gathered} c_x, c_y \subseteq c_{\text{✗}} \\ \|.\| \colon \text{size of the serialized configuration} \\ \exists c_x \colon \mathit{test}(c_x) \rightarrow \text{✗ found} \\ \forall c_y \colon \|c_y\| > \|c_x\| \colon \text{ out of search space} \end{gathered} \tag{1}$$

It is known that after a failing configuration is found, its subsets would be reduced further via DDMIN, so theoretically, there is no chance of getting a *failing* outcome back from the cache. Suppose we have a configuration of size n, and before testing it, a cache lookup is performed. The cache may contain smaller entries, however, if a smaller entry ($m < n$) would be in the cache with a failing outcome, then the current state could not have occurred, since DDMIN would have split that m-sized entry into even smaller chunks. Therefore, when a cache hit occurs, we can be sure that it was the result of a *passing* test. Thus, the *first proposal*, as shown in (2), is to add only *passing* tests to the cache which may reduce the memory footprint of the minimization algorithm. Furthermore, cache lookups might be quicker since the queries are performed in a smaller search space. The *insert to cache* function inserts an element into the cache, while *serialize* performs the serialization of test cases as discussed in [3].

$$\begin{gathered} c_x \subseteq c_{\text{✗}} \\ \text{when } \exists c_x \colon \mathit{test}(c_x) \rightarrow \text{✓ found} \\ \mathit{insert\ to\ cache}\ (\ \mathit{serialize}(c_x)\ ) \end{gathered} \tag{2}$$

Another benefit of (1) is that when a failing test case is found, we can be sure that no cache entry corresponding to test cases larger than the currently found one will be queried during the remaining reduction process. Therefore, when a new failing test case is found, the entries that store the result of test cases larger than the currently investigated one can be evicted from the cache, as shown in (3). This eviction process will be referred to as the *second proposal*. The *delete from cache* function implements the removal of an item from the cache.

$$\begin{gathered} c_x, c_y \subseteq c_{\text{✗}} \\ \text{when } \exists c_x \colon \mathit{test}(c_x) \rightarrow \text{✗ found} \\ \forall c_y \colon \mathit{serialize}(c_y) \in \text{cache} \wedge \|c_y\| > \|c_x\| \colon \\ \mathit{delete\ from\ cache}\ (\ \mathit{serialize}(c_y)\ ) \end{gathered} \tag{3}$$

If the above-described proposals are applied, the cache will only contain *passing* tests and will be cleared after each successful reduction step. However, the lengths of the cached entries will vary, *i.e.*, they will be larger at the beginning of the reduction (proportional to the size of the initial failing test case) and will become smaller as the process progresses towards the 1-minimum. The *third proposal* is the following: the cache should not store the serialized contents of the configurations, but their transformed form as shown in (4).

$$\begin{gathered} c_x \subseteq c_{\text{✗}}, M \colon M \in \mathbb{N} \\ \mathit{transform}(c_x) \colon 2^{\mathbb{N}} \mapsto 2^M \text{ bijection} \\ \text{when } \exists c_x \colon \mathit{test}(c_x) \rightarrow \text{✓ found} \\ \mathit{insert\ to\ cache}\ (\mathit{transform}(\ \mathit{serialize}(c_x)\ )) \end{gathered} \tag{4}$$

The proposal is functional only if the transformation is bijective, *i.e.*, each test case has its own transformed form, each transformed element corresponds to exactly one test case, and unpaired elements are forbidden. From a practical point of view, the bijection is not possible, since an infinite set would have to be mapped to a finite one. Therefore, a sufficiently large M and a suitable *transform* function must be chosen to minimize the possibility of cache key collisions, *e.g.*, a *SHA-3-256* cryptographic hash function[1]. On the other hand, if the chosen M is too large, the desired positive effect on memory usage is lost.

Results: With the optimizations combined, DDMIN requires 96% and HDD requires 85% less memory than the baseline implementation. In support of the scalability issue, the size of the input affected the results: the average improvement was 63% for smaller tests, while it was 99% for larger inputs. Furthermore, as a side effect, the reduction becomes faster by 10% with DDMIN. In our experiments, the result of the reductions did not change after the optimizations.

2 Iterating the Minimizing Delta Debugging Algorithm

Minimizing Delta Debugging is already more than twenty years old and still widely used because it works on any kind of input. Many approaches have tried to work smarter since DDMIN first appeared, but they typically needed some extra information about the structure of the test case, usually a grammar. This additional requirement can act as a blocker for some users of test case reducers: a grammar may not be readily available, and writing (or maintaining) one may not be a practical option. In such cases, the structure-unaware nature of DDMIN is proven to be very useful.

The program in Figure 1 is a variant of a classic example of program slicing [5]. It computes both the sum and the product of the first ten natural numbers in a single loop.

[1]https://csrc.nist.gov/projects/hash-functions/sha-3-project

Using slicing terminology, we can say that we want to compute the (so-called static backward) slice of this program with respect to the criterion (19, *prod*), thus creating a sub-program that does not contain statements that do not contribute to the value of *prod* at line 19.

```
int add(int a, int b)
{
    return a + b;
}
int mul(int a, int b)
{
    return a * b;
}
void main()
{
    int sum = 0;
    int prod = 1;
    for (int i = 1; i <= 10; i++)
    {
        sum = add(sum, i);
        prod = mul(prod, i);
    }
    printf("sum: %d\n", sum);
    printf("prod: %d\n", prod);
}
```

Figure 1: *Example C program that computes the sum and product of the first ten natural numbers, and the execution of DDMIN on it while keeping 10! on the output.*

This can be computed either by analyzing the control and data dependencies of the program – which is the classic slicing way – or by following the approach of observation-based slicing [1] that performs a systematic removal of code parts based on trial and error, similar to what DDMIN does on its input. In fact, even DDMIN can be applied to such tasks. The two things to keep in mind are that in such reduction scenarios, the inputs or test cases are also programs, and the interesting properties to preserve are not program failures (but it is still an ✗ that represents that the property is kept). Thus, we reformulate the classic slicing example as a test case minimization task, where the program in Figure 1 is the input (the lines are the units) and the testing function is given as

$$test(c) = \begin{cases} ✓ & \text{if } c \text{ is syntactically incorrect} \\ ✓ & \text{else if execution of } c \text{ does not terminate} \\ ✓ & \text{else if execution of } c \text{ does not print } \textit{prod: 3628800} \\ ✗ & \text{otherwise.} \end{cases}$$

The gray bars to the right of the program code show the progress of DDMIN from left to right. Each set of vertically aligned bars corresponds to a configuration of the algorithm and shows how that configuration is split into subsets. This example shows that DDMIN could "slice away" the lines of the *main* function that did not contribute to the calculation of *prod*. However, the algorithm could not remove the *add* function, because when the configuration no longer contained a call to it (at line 15), the granularity had already reached the line (*i.e.*, unit) level. But *add* could only be removed as a whole, not line by line, as removing any

```
int add(int a, int b)
{
    return a + b;
}
int mul(int a, int b)
{
    return a * b;
}
void main()
{
    int prod = 1;
    for (int i = 1; i <= 10; i++)
    {
        prod = mul(prod, i);
    }
    printf("prod: %d\n", prod);
}
```

Figure 2: *The output of DDMIN on the program of Figure 1, and the re-execution of DDMIN.*

single line would cause syntax errors. (This is one of the shortcomings of DDMIN that the grammar-based reducers wanted to fix.) So, DDMIN has produced a 1-minimal result (shown in Figure 2), but it is clearly not a global minimum. What we can realize when looking at this result is that we could run DDMIN again on this program with the same testing function as the first time, and we might be able to remove the superfluous *add* function as well. Again, the gray bars to the right of the program code show the progress of DDMIN, and indeed, the subsets of the second configuration aligned well with the structure of this input and allowed for further reduction. The result of the second execution of DDMIN is shown in Figure 3. This is the global optimum for this example, so further executions of DDMIN are not visualized.

```
int mul(int a, int b)
{
    return a * b;
}
void main()
{
    int prod = 1;
    for (int i = 1; i <= 10; i++)
    {
        prod = mul(prod, i);
    }
    printf("prod: %d\n", prod);
}
```

Figure 3: *The output of DDMIN on the program of Figure 2.*

Motivated by this example, we can formalize the intuition that DDMIN could be executed multiple times. Since it is not known *a priori* how many executions are needed for a given input, we propose to iterate DDMIN until a fixed point is reached. We will denote the fixed-point iteration of DDMIN as DDMIN* – following the notation used for HDD and

HDD* [4] – and define it as follows:

$$ddmin^*(c_{\text{✗}}) = \begin{cases} c'_{\text{✗}} & \text{if } c_{\text{✗}} = c'_{\text{✗}} \\ ddmin^*(c'_{\text{✗}}) & \text{otherwise} \end{cases}$$
$$\text{where } c'_{\text{✗}} = ddmin(c_{\text{✗}}).$$

Although the asterisk notation is the same for both algorithms and even its meaning is identical in both cases (*i.e.*, fixed-point iteration), its purpose is fundamentally different for HDD and DDMIN. A single execution of HDD has no minimality guarantees, only HDD* produces 1-tree-minimal results. However, even a single execution of DDMIN is guaranteed to give a 1-minimal result. The purpose of further iterations is to find an even better 1-minimum. (Re-executing DDMIN does not guarantee better results in all cases, only if the configuration aligns well with the structure of the input.)

Results: First, the reduction of test cases was performed with character level granularity on a smaller test suite, the output became smaller by 68% on average. Then, the reduction was performed with line granularity, and the experiments show that DDMIN* can produce 48% smaller outputs on average (69% on a larger test suite and 19% on a smaller one). The price of this improvement is an increase in the number of steps, which was 66% on average. A "combined" two-pass reduction was then performed, where test cases were first reduced with line granularity, and then these intermediate results were reduced further with character granularity as fine-tuning. DDMIN* outperformed DDMIN even with this setting, and was able to further reduce inputs by an average of 46%. Surprisingly, some inputs could be reduced faster with DDMIN*, as the line-level reduction produces results in a reasonable number of steps, and then the character-level reduction can work further from this smaller input configuration. Encouraged by the promising results, we compared the output of DDMIN* with the output of HDD*, to see whether a structure-unaware algorithm could compete with a "more clever" one. In terms of required testing steps, the answer is simply no; however, in terms of size, DDMIN* brought the results much closer, from 9 times larger outputs (DDMIN) to only 3 times larger ones.

3 Parallel Optimizations of DDMIN*

We have investigated whether it would be possible to make DDMIN itself work faster without compromising its minimality guarantees. One technique that has already been proven useful for speeding up DDMIN is parallelization [2]. The question we sought the answer to was whether it was possible to make the *parallel DDMIN* even faster without losing the 1-minimal property of its output.

Hodován *et al.* [2] noticed that the original implementation of DDMIN used sequential loops to realize the "reduce to subset" and "reduce to complement" phases, however, the potential for parallelization is present in the formalization of DDMIN. Since the size of the initial configuration can be large for real inputs (and testing a configuration is considered to be an expensive part of the algorithm), they rewrote DDMIN to use parallel loops. As testing different configurations is independent, their proposal worked well in practice and achieved 75-80% less runtime in their experiments. Figure 4a shows how sequential loops

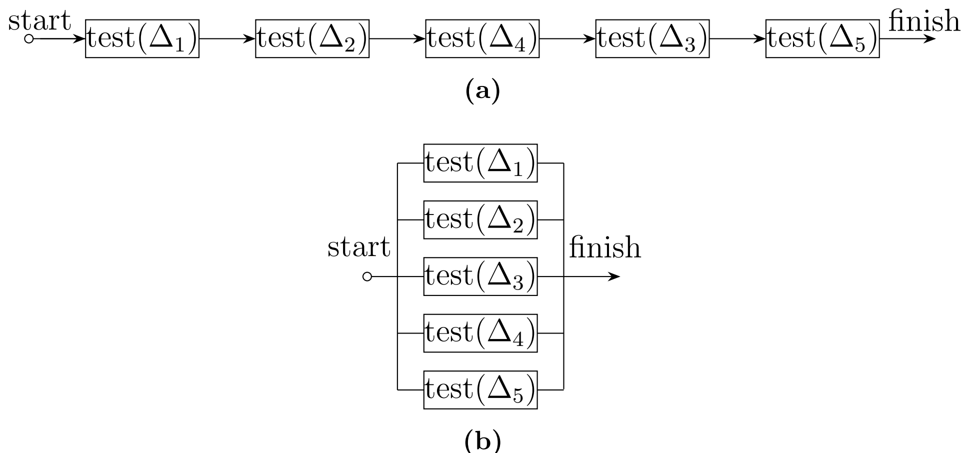


Figure 4: *(a) Sequential execution of "reduce to subset". (b) Parallel execution of "reduce to subset".*

iterate through five configurations: if we assume that each *test* takes the same amount of time t, then checking all of them takes $5t$. However, if the loops are implemented in a parallel way, as shown in Figure 4b, checking the configurations takes only t, which might bring a significant speedup in reduction. For the formal definition of their parallel DDMIN formulation, the reader is referred to [2]. Three assumptions were made regarding correctness and effectiveness, of which the following is relevant to this study: When a *fail* is found in a parallel loop, the other active loop bodies should be aborted even if their computation has not finished yet. This might cause computation results to be thrown away, but it does not violate the minimality guarantees of the algorithm.

Let j be the parallelization capabilities of the algorithm, *i.e.*, how many $test(\Delta_i)$ or $test(\nabla_i)$ jobs can be started concurrently (five was used in Figure 4). Let T be the *testing window* ($|T| = j$), *i.e.*, j tests are executed and j results (✔, ✘, or **?**) are produced. Let F denote the set of configurations with a *fail* outcome in T; if $|F| > 1$ then the behavior of parallel DDMIN [2] becomes unstable: it will choose among the interesting configurations based on which produced its *fail* outcome first. Therefore, different test reductions can yield different outcomes, which is not appropriate for carrying out reproducible experiments. (Note that the 1-minimality of the algorithm is not harmed, since multiple local minima might exist.)

The "reduce to subset" and "reduce to complement" phases iterate through configurations in a forward or backward syntactic order; it is known which configuration *should* be investigated first. To stabilize the algorithm, the following changes must be made: if a *fail* is found in T, then no new parallel loop bodies are started (no change), and the active *test* executions should be awaited (*i.e.*, computation results are not thrown away). If multiple *fail*s are found, the syntactic order must be taken into account when choosing which one to reduce further.

When multiple *fail*s are found in T and the algorithm chooses one of them based on the iterator, the results from other configurations are discarded, even if they could have been useful. This results in unnecessary test executions on configurations that have already been tested (and *fail*ed). The following strategy can help minimize the number of test executions:

$$greeDDy(c'_{✗}) = greeDDy_2(c'_{✗}, 2) \text{ where}$$

$$greeDDy_2(c'_{✗}, n) = \begin{cases} greeDDy_2(\bigcap_{i \in F} \overline{C}_i, max(n - |F|, 2)) & \text{if } |F| > 1 \wedge test(\bigcap_{i \in F} \overline{C}_i) = ✗ \text{ ("reduce greedily")} \\ greeDDy_2(\overline{C}_i, 2) & \text{else if } \exists i \in F \cdot \overline{C}_i \in \{\Delta_1, ..., \Delta_n\} \text{ ("reduce to subset")} \\ greeDDy_2(\overline{C}_i, max(n-1, 2)) & \text{else if } \exists i \in F \cdot \overline{C}_i \in \{\nabla_1, ..., \nabla_n\} \text{ ("reduce to complement")} \\ greeDDy_2(c'_{✗}, min(|c'_{✗}|, 2n)) & \text{else if } n < |c'_{✗}| \text{ ("increase granularity")} \\ c'_{✗} & \text{otherwise ("done").} \end{cases}$$

where $\overline{C}$ is a sequence of C, such that $\{\nabla_1, ..., \nabla_n\} \subseteq C \subseteq \{\Delta_1, ..., \Delta_n, \nabla_1, ..., \nabla_n\}$,
$\nabla_i = c'_{✗} - \Delta_i$, $c'_{✗} = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{✗}|/n$ holds.
F is a set of indices over $\overline{C}$ such that $\forall i \in F \cdot test(\overline{C}) = ✗$ and $F = \emptyset \iff \nexists \overline{C_i} \cdot test(\overline{C_i}) = ✗$.

Figure 5: *GreeDDy: the greedy extension of minimizing Delta Debugging.*

If a testing window has multiple *fail*s, then it is worth trying to combine the configurations that yielded them and check whether those combinations also result in a *fail*. If yes, several test executions are saved in one step. If no, then select the first *fail* (based on the syntactic order) and try to combine the other interesting configurations one by one. In this case, test steps can also be saved since only configurations with a *fail* outcome are retested in the next parallel loop iteration, instead of the whole testing window. Figure 5 formalizes our proposed optimization using the notations discussed earlier.

Results: The modified parallel DDMIN – called *GreeDDy* – is presented, evaluated on a subset of a publicly available dataset and found that *GreeDDy** could save 31% of the testing steps of DDMIN* which resulted in 40% less runtime.

4 Extending Hierarchical Delta Debugging with Hoisting

Although HDD and its variants perform better on structured inputs than DDMIN, there is still room for improvement. Several improvements have already been proposed, often by preprocessing the tree representation that HDD is working on, *e.g.*, by hiding some tokens from HDD to reduce the number of nodes that have to be considered, by collapsing multiple nodes into one for the same reason, or by rotating recursive structures of the tree to reduce its height. However, these transformations do not change the core structure of the tree, *i.e.*, the test case serialized from the preprocessed tree will still be the same as the original input. Because of this, and because of the way HDD variants work, an HDD-reduced test case (even if it is 1-tree-minimal) may contain structural elements that a human expert would still remove.

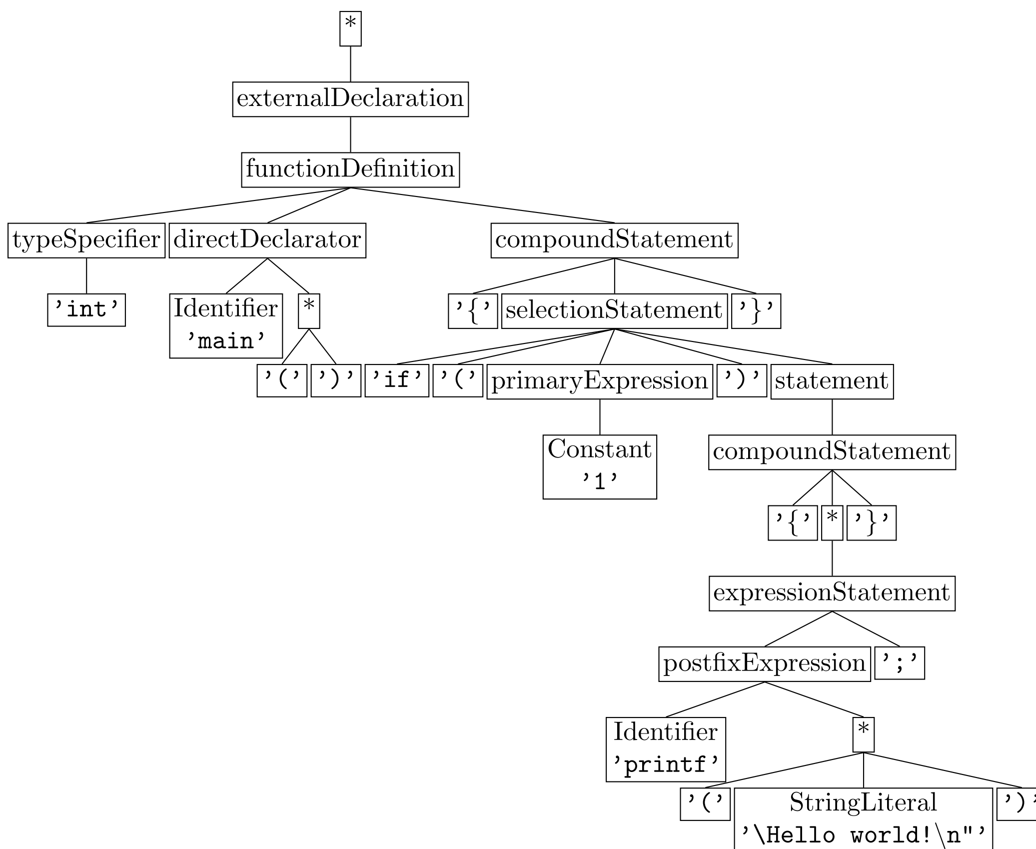
A simple example of this suboptimal structure-preserving behavior is shown in Figure 6. The C program in Figure 6a prints the classic "Hello world!" message, and the printing is wrapped in an *if* statement where the predicate always evaluates to true. If we take this program as a test case and define the printing of the "Hello world!" message as interesting, then we can try and minimize it. (This is an example where the interesting property of the test case is *not* a program failure.) Figure 6b shows the parse tree of the program, generated by a parser using a context-free grammar of the C programming language and

preprocessed for compactness (most notably, squeezing and recursion flattening have been applied). Unfortunately, none of the HDD-based algorithms can reduce this test case further as removing any of the nodes would either yield a syntactically incorrect test case or one that does not print the message, making it uninteresting.

There are recurring structures in the parse tree, subtrees rooted at nodes with identical labels, denoting the derivation of the same non-terminal of the grammar. The assumption of hoisting is that one such subtree can be replaced by another without losing syntactic correctness, and that subtrees whose roots are in an ancestor-descendant relationship are good candidates for reduction. In the tree in Figure 6b, there is a pair of such subtrees, those rooted at nodes labeled as *compoundStatement*. Figure 7a shows a transformed tree where the descendant subtree is hoisted to replace all the structures that enclosed it. When this tree is serialized in the form of a C program (see Figure 7b), it becomes apparent that this transformation was indeed useful in this case and we have a smaller but still interesting test case. The testing function must confirm (or reject) whether such a transformation keeps the

```
int main() {
  if (1) {
    printf("Hello world!\n");
  }
}
```

(a) *A "Hello World" program in C.*



(b) *Parse tree of the "Hello World" program.*

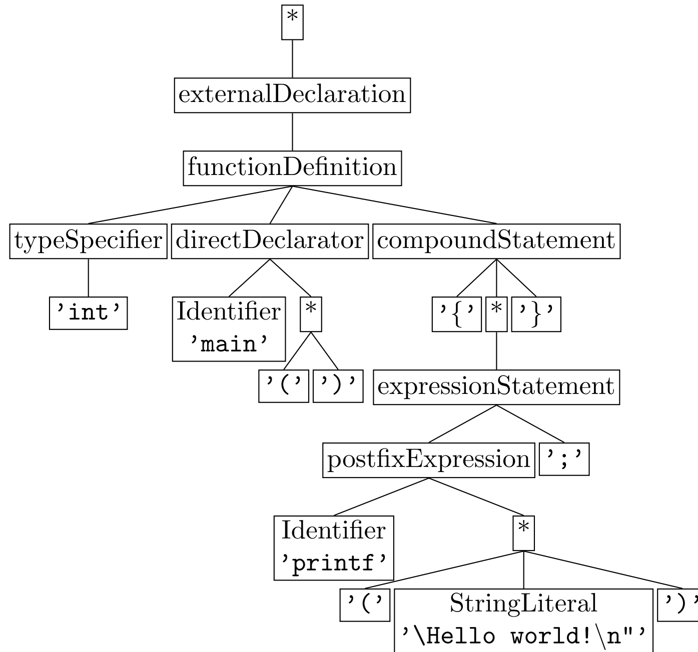
Figure 6: *An overly complicated "Hello World" program in C and its parse tree.*

resulting test case interesting.

To formalize the ideas motivated and described above, the notations and terminology of DDMIN have been extended to introduce transformation-based minimization. In the context of DDMIN, a test case is always composed of a subset containing elements of the initial configuration. The testing function is also defined only for the subsets of $c_{\text{✗}}$. However, the outcome of a program composed of a set of elements must be determined, even if some of them are not part of the initial configuration. In the case of hoisting, when an element (a node) is replaced by another element (another node further down the hierarchy), which is part of the tree, but is not a member of the initial set. Therefore, the definitions of DDMIN are generalized as follows.

Let D denote the set of all potential test case elements, and let $\delta \in D$ denote an element of this set, *i.e.*, a test case element. A test case or configuration is denoted by $c \subseteq D$. A testing function $test : 2^D \to \{\text{✗}, \text{✓}, ?\}$ shall determine for any test case whether it produces the failure in question. The initial failing configuration is denoted by $c_{\text{✗}} = \{\delta_1, \ldots, \delta_n\} \subseteq D$, and $test(c_{\text{✗}}) = \text{✗}$ holds. Since $c_{\text{✗}}$ is a subset of a potentially larger set D, we allow *transformations* that can not only remove but also *replace* elements in the configuration. The following definitions and notations are used for transformations:

A function $t : D \to D$ is a transformation of test case elements, and the identity transformation is $id_D : D \to D; \delta \mapsto \delta$. The application of a transformation to configurations



(a) *Parse tree minimized with hoisting applied to keep printing the "Hello world!" message.*

```
int main() {
  printf("Hello world!\n");
}
```

(b) *The C program serialized from the minimized tree.*

Figure 7: *Motivational Example for Hoisting.*

Let D denote the set of all potential test case elements, and let $\delta \in D$ denote one element of that set.
Let *test* and $c_✗ = \{\delta_1, \dots, \delta_n\} \subseteq D$, and $test(c_✗) = ✗$ holds.
Let τ and $\|\cdot\|$ be given such that $\forall \delta \in D \cdot \forall \delta' \in \tau(\delta) \cdot \|\delta'\| < \|\delta\|$ holds.
The goal is to find $t_✗ = tmin^\tau(c_✗)$ such that $test(\bar{t}_✗(c_✗)) = ✗$ and $t_✗$ is 1-maximal.
The *transformation-based minimizing algorithm* $tmin^\tau(c)$ is

$$tmin^\tau(c_✗) = tmin_2^\tau(c_✗, id_D) \text{ where}$$

$$tmin_2^\tau(c_✗, t'_✗) = \begin{cases} tmin_2^\tau(c_✗, t'_✗[\delta \mapsto \delta']) & \text{if } \exists \delta \in c_✗ \cdot \exists \delta' \in \tau(t'_✗(\delta)) \cdot test(\bar{t'_✗}[\delta \mapsto \delta'](c_✗)) = ✗ \\ t'_✗ & \text{otherwise.} \end{cases}$$

The recursion invariant (and thus precondition) for $tmin_2^\tau$ is $test(\bar{t'_✗}(c_✗)) = ✗$.

Figure 8: *The Transformation-based Minimizing Algorithm.*

is defined as $\bar{t} : 2^D \to 2^D; c \mapsto \{t(\delta) : \delta \in c\}$ (*e.g.*, $\overline{id}_D(c_✗) = c_✗$).

And a transformation that is derived from another transformation by changing the mapping of one test case element is defined as

$$t[\delta' \mapsto \delta''] : D \to D; \delta \mapsto \begin{cases} \delta'' & \text{if } \delta = \delta' \\ t(\delta) & \text{otherwise.} \end{cases}$$

The transformations that could be applied were quite straightforward in the presented example. There was only one *compoundStatement* that could potentially replace its parent. In a general case, a test case element may have multiple replacement candidates (or none at all). This is formalized by a function $\tau : D \to 2^D$ that maps test case elements to their transformed candidates.

Finally, since test cases are not necessarily subsets of the initial failing configuration, minimality can no longer be defined in terms of the subset relation. Thus, a $\|\cdot\|$ measure is expected to exist on the set D. If all transformation candidates in τ potentially reduce the size of a configuration according to the measure $\|\cdot\|$, *i.e.*, $\forall \delta \in D \cdot \forall \delta' \in \tau(\delta) \cdot \|\delta'\| < \|\delta\|$ holds, then in order to minimize the test case, the replacements applied to the elements of the initial configuration must be maximized (even transitively) while ensuring that the so-transformed test case remains interesting. Just as it is true for DDMIN that searching for the global optimum is impractical, it is also true for transformation-based minimization. Therefore, the goal is to find a local optimum, a *1-maximal* transformation $t_✗$ such that $\forall \delta \in c_✗ \cdot \forall \delta' \in \tau(t_✗(\delta)) \cdot test(\bar{t}_✗[\delta \mapsto \delta'](c_✗)) \neq ✗$ holds.

Figure 8 formalizes the transformation-based minimizing algorithm TMIN$^\tau$, worded in the likeness of DDMIN.

The transformation-based minimization algorithm provides a framework for formulating hoisting as a transformation of tree nodes. More precisely, those nodes in the tree representation of the input that can act as replacement candidates for their ancestors must be defined. The formula in Figure 9, $\chi(n)$, is one possible way to define these candidates, *i.e.*, the hoistable descendants of a node n. $\chi(n)$ is given in terms of two auxiliary functions, of which *children*(n) is trivial, giving the direct descendants of a node, whereas *compatible*(n, n') leaves some space for interpretation. In an extreme case, any two nodes could be considered compatible, but that would rarely be useful. If the tree representation of the input is built using a context-free grammar, then a natural interpretation is to regard identically labeled nodes (*i.e.*, subtrees of derivations of the same non-terminal of the grammar) as compatible.

$$\chi(n) = \bigcup_{n' \in children(n)} \chi'(n, n')$$

$$\chi'(n, n') = \begin{cases} \{n'\} & \text{if } compatible(n, n') \\ \bigcup_{n'' \in children(n')} \chi'(n, n'') & \text{otherwise} \end{cases}$$

Figure 9: *$\chi(n)$, the potentially hoistable descendants of node n.*

A basic measure for nodes of a tree is based on the size of their subtrees, *i.e.*, the number assigned by the measure to a node n equals the number of nodes in the subtree of n. It is obvious that all transformation candidates returned by $\chi(n)$ reduce the size of the configuration according to this measure, as expected by the definition of TMIN.

Now, with the help of TMIN$^{\chi}$, a hierarchical algorithm called Hoist can be introduced, which works its way through the tree from the root to the leaves, using TMIN$^{\chi}$ to find the hoisting transformations at each level. Candidates found by TMIN$^{\chi}$ are prioritized by their distance to the ancestor, with further nodes getting higher priority. The pseudocode of the algorithm is shown in Figure 10a. The structure of Hoist is similar to HDD: both contain a loop to iterate through the levels of a tree and inside the loop, both perform a minimization step (TMIN$^{\chi}$ vs. DDMIN) and the application of its result to the tree (via the *transform* and *prune* auxiliary functions).

Hoist can achieve reduction on its own, although it is expected to work best if used in combination with HDD, *e.g.*, by using Hoist as a preprocessing step. However, inspired by the similarities between the variants of these two algorithms, they can be combined as well. *E.g.,* the bodies of the loops can be interlaced, performing both the DDMIN and TMIN$^{\chi}$-based minimization at each iteration. One way to formulate this idea is shown in Figure 11, where HDD and Hoist are interlaced in the algorithm named HDDH.

Because of the similarities between HDD variants and the Hoist algorithm, a recursive, a coarse, and a coarse recursive variant of the hoisting algorithm can be defined. These are given in Figures 10b, 10c, and 10d, and are named Hoist$^{\mathrm{r}}$, Coarse Hoist, and Coarse Hoist$^{\mathrm{r}}$, respectively. Similarly, we can create new combined algorithms from HDD$^{\mathrm{r}}$, and Hoist$^{\mathrm{r}}$ (HDDH$^{\mathrm{r}}$), from Coarse HDD, and Coarse Hoist (Coarse HDDH), and from HDD$^{\mathrm{r}}$ and Coarse Hoist (Coarse HDDH$^{\mathrm{r}}$). These combinations are trivial following the example of HDDH, therefore, they are not shown to avoid unnecessary repetition.

Results: On real-world inputs, hoisting combined with HDD gives generally smaller, or at least as small outputs as HDD alone. Bigger outputs are rare. Minimized test cases can be as small as ⅕ of the output given by traditional HDD. The effects of hoisting to HDD and HDD$^{\mathrm{r}}$ are similar: the majority of the test cases could be reduced further with hoisting. Coarse HDD and Coarse HDD$^{\mathrm{r}}$ show similar patterns to the non-coarse variants with respect to the output size: test cases could be reduced further with hoisting. However, hoisting had no effect on the Coarse HDDH and Coarse HDDH$^{\mathrm{r}}$ algorithm variants, furthermore, algorithms performed the reduction exactly the same way when hoisting was a preprocessing step. The effect of hoisting on the efficiency of the reduction highly depends on the height of the input

```
1  procedure Hoist(input_tree)
2      level ← 0
3      nodes ← tagNodes(input_tree, level)
4      while nodes ≠ ∅ do
5          hoisting ← TMINᵡ(nodes)
6          transform(input_tree, level, hoisting)
7          level ← level + 1
8          nodes ← tagNodes(input_tree, level)
9      end while
10 end procedure
```

(a) *Hoisting.*

```
1  procedure Hoistʳ(root_node)
2      queue ← ⟨root_node⟩
3      while queue ≠ ⟨⟩ do
4          current_node ← pop(queue)
5          nodes ← tagChildren(current_node)
6          hoisting ← TMINᵡ(nodes)
7          transformChildren(current_node, hoisting)
8          append(queue, tagChildren(current_node))
9      end while
10 end procedure
```

(b) *Recursive Hoisting.*

```
1  procedure CoarseHoist(input_tree)
2      level ← 0
3      nodes ← tagNodes(input_tree, level)
4      while nodes ≠ ∅ do
5          nodes ← filterEmptyPhiNodes(nodes)
6          if nodes ≠ ∅ then
7              hoisting ← TMINᵡ(nodes)
8              transform(input_tree, level, hoisting)
9          end if
10         level ← level + 1
11         nodes ← tagNodes(input_tree, level)
12     end while
13 end procedure
```

(c) *Coarse Hoisting.*

```
1  procedure CoarseHoistʳ(root_node)
2      queue ← ⟨root_node⟩
3      while queue ≠ ⟨⟩ do
4          current_node ← pop(queue)
5          nodes ← tagChildren(current_node)
6          nodes ← filterEmptyPhiNodes(nodes)
7          if nodes ≠ ∅ then
8              hoisting ← TMINᵡ(nodes)
9              transformChildren(current_node, hoisting)
10         end if
11         append(queue, tagChildren(current_node))
12     end while
13 end procedure
```

(d) *Recursive Coarse Hoisting.*

Figure 10: *Proposed Hoisting algorithm and its variants.*

```
1  procedure HDDH(input_tree)
2      level ← 0
3      nodes ← tagNodes(input_tree, level)
4      while nodes ≠ ∅ do
5          minconfig ← DDMIN(nodes)
6          prune(input_tree, level, minconfig)
7          hoisting ← TMINᵡ(minconfig)
8          transform(input_tree, level, hoisting)
9          level ← level + 1
10         nodes ← tagNodes(input_tree, level)
11     end while
12 end procedure
```

Figure 11: *The Hierarchical Delta Debugging and Hoisting algorithm.*

tree. If the height of the tree is small (below 50), hoisting increases the required testing steps. However, if the height of the tree is big enough (above 150), test cases can be reduced faster with hoisting. The Hoist* + HDDH* and Hoist^{r}* + HDDH^{r}* algorithm variants produced the smallest output among the tested ones, and the Coarse variants performed the reduction requiring the fewest steps (at the cost of bigger outputs).

Table 1: *Summary of thesis topics and corresponding publications*

	[11]	[6]	[9]	[10]	[7]	[8]
1	•					
2		•	•			
3				•		
4					•	•

The Author's Contributions

The author had a decisive role in the design, implementation and evaluation of a significant proportion of the above presented findings.

1. **Cache Optimizations**: The author analyzed the state-of-the-part caching solutions that have been used in reduction, then designed and implemented three optimizations for reducing the memory footprint of the reduction. Then, he evaluated the effects of proposals with different reduction approaches, on multiple test suites.

2. **Iterating the Minimizing Delta Debugging Algorithm**: The author designed and prototyped the fixed-point iteration of the DDMIN algorithm.

3. **Parallel Optimizations of DDMIN***: The author analyzed the weaknesses of the parallel DDMIN, then designed, implemented and evaluated a solution to it.

4. **Extending Hierarchical Delta Debugging with Hoisting**: The author investigated the structure of the inputs (abstract syntax trees) of HDD searching for optimization possibilities. He found that identically labeled nodes can be replaced without loosing the syntactic correctness. Then, he prototyped the transformation-based minimization framework, implemented the hoisting as an example transformation, and evaluated it on publicly available test suites.

Furthermore, the supporting replication package has been published at the time of each publication. The author has been responsible for the redesign and implementation of the algorithms that stand their ground in the world of open-source. The publications related to the thesis points are the following:

[11] **Dániel Vince** and Ákos Kiss. Cache Optimizations for Test Case Reduction. In *Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS 2022)*, pages 442-453, Guangzhou, China (Virtual), December 2022. IEEE.

[6] **Dániel Vince**. Iterating the Minimizing Delta Debugging Algorithm. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST'22)*, pages 57-60, Singapore, November 2022. ACM.

[9] **Dániel Vince** and Ákos Kiss. Evaluation of the fixed-point iteration of minimizing delta debugging. In *Journal of Software: Evolution and Process*, 2024. Wiley.

[10] **Dániel Vince** and Ákos Kiss. GreeDDy: Accelerate Parallel DDMIN. In *Proceedings of the 15th ACM International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST '24)*, pages 1-4, Vienna, Austria, September 2024. ACM.

[7] **Dániel Vince**, Renáta Hodován, Daniella Bársony, and Ákos Kiss. Extending Hierarchical Delta Debugging with Hoisting. In *Proceedings of the 2nd ACM/IEEE International Conference on Automation of Software Test (AST 2021)*, pages 60-69, Madrid, Spain (Virtual), May 2021. IEEE.

[8] **Dániel Vince**, Renáta Hodován, Daniella Bársony, and Ákos Kiss. The effect of hoisting on variants of Hierarchical Delta Debugging. In *Journal of Software: Evolution and Process*, 34(11):e2483:1-e2483:26, November 2022. Wiley.

Summary

When a new bug is found in software, it is usually reported in the project's bug tracking system with a test case. At the end of the process, it is sent to an engineer who reproduces it and then fixes it. In the debugging process, it does matter how many irrelevant parts are in a test case; the noisier it is, the more valuable time is spent filtering out the parts that actually caused the bug.

Fortunately, this does not need to be done manually; algorithms are available and became important when the random test case generation (fuzz-testing) became popular. This thesis discusses two major algorithms of automatic test case reduction, minimizing Delta Debugging (DDMIN) and Hierarchical Delta Debugging (HDD), and their optimization possibilities. The study can be divided into two main parts: either making the output of the algorithm smaller or making the reduction process more lightweight.

Reduction algorithms do not aim to find a global optimum; finding one would take too long from a practical point of view. However, the property of a local optimum is that a better one can potentially be found. This feature is used in both algorithms. For DDMin, we proposed the use of a so-called fixed-point iteration: as long as the algorithm can reduce its input, it is repeated, and the input of the next iteration is the output of the current one. Experiments show that there are at least two iterations on the datasets used, and on average the output is 68% smaller with character-based reduction. For HDD, transformations have been proposed on the tree structure that forms its input. It is easy to construct an example that cannot be reduced by the prune-based HDD algorithm. This problem has been investigated and a hoisting transformation has been proposed that can replace one node of the tree with another if the two nodes have identical labels and are in an ancestor-descendant relationship.

When running these algorithms, it is possible for multiple configurations to result in the same serialized test case, therefore, the same test executions are checked multiple times. To avoid this duplication, a cache is used to store the generated test cases and their results. However, the cache itself can consume a lot of memory, which is not good either. In order to optimize this memory consumption, the study proposes three solutions, the combined use of which achieved a 96% improvement for DDMIN and an 85% improvement for HDD. A technique that has already been proven useful for speeding up DDMIN is parallelization, but the published concepts still have some room for improvement: The stability issues of the algorithm are described, then a potential solution is outlined. A greedy approach is proposed that reduces the number of required test runs by 31% and the runtime by 40%.

Acknowledgments

Firstly, I would like to thank Dr. Ákos Kiss, my supervisor, for his professional help and unique opinions during my PhD studies. I will not forget my colleagues who gave valuable feedback on the manuscript: Dr. Dombi József Dániel, Zsolt Borbély, and Edit Szűcs. I would like to express my gratitude for the continuous support of my family. Then, I would like to thank Amanda, my wife, who supported me all the way, even though I spent the evenings quietly on the sofa, writing this text. Finally, I am really grateful to Noel, my son, whose arrival put a hard deadline to my writing.

One or more research papers, the result of which were used in this thesis, were partially supported by

- GINOP-2.3.2-15-2016-00037: the EU-supported Hungarian national grant,
- NKFIH-1279-2/2020: of the Ministry for Innovation and Technology, Hungary,
- TKP2021-NVA-09: implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme,
- ÚNKP-22-3-SZTE-469 and ÚNKP-23-3-SZTE-536: New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

References

[1] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. ORBS: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 109–120. ACM, November 2014.

[2] Renáta Hodován and Ákos 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*, pages 241–248. SciTePress, July 2016.

[3] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Tree preprocessing and test outcome caching for efficient hierarchical delta debugging. In *Proceedings of the 12th IEEE/ACM International Workshop on Automation of Software Testing (AST 2017)*, pages 23–29. IEEE, May 2017.

[4] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pages 142–151. ACM, May 2006.

[5] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[6] Dániel Vince. Iterating the minimizing delta debugging algorithm. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST'22)*, pages 57–60. ACM, November 2022.

[7] Dániel Vince, Renáta Hodován, Daniella Bársony, and Ákos Kiss. Extending hierarchical delta debugging with hoisting. In *Proceedings of the 2nd ACM/IEEE International Conference on Automation of Software Test (AST 2021)*, pages 60–69. IEEE, May 2021.

[8] Dániel Vince, Renáta Hodován, Daniella Bársony, and Ákos Kiss. The effect of hoisting on variants of hierarchical delta debugging. *Journal of Software: Evolution and Process*, 34(11):e2483:1–e2483:26, November 2022.

[9] Dániel Vince and Ákos Kiss. Evaluation of the fixed-point iteration of minimizing delta debugging. *Journal of Software: Evolution and Process*, (n/a):e2702.

[10] Dániel Vince and Ákos Kiss. Greeddy: Accelerate parallel ddmin.

[11] Dániel Vince and Ákos Kiss. Cache optimizations for test case reduction. In *Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS 2022)*, pages 442–453. IEEE, December 2022.

[12] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

Összefoglaló

Amikor egy új hibát találnak egy szoftverben, azt bejelentik a projekt hibakezelő rendszerében. A folyamat végén valaki reprodukálja és kijavítja azt A hibakeresés során nem mindegy, hogy mennyi irreleváns rész van egy tesztesetben, minél zajosabb, annál több értékes idő megy el a hibát kiváltó részek megkeresésével.

Szerencsére ezt nem kell manuálisan elvégezni, algoritmusok állnak rendelkezésünkre. A dolgozat az automatikus tesztesetredukció két elterjedt algoritmusát, a "minimizing Delta Debugging"-ot (DDMIN) és a "Hierarchical Delta Debugging"-ot (HDD), és az ezekhez kapcsolódó optimalizációkat tárgyalja. A dolgozat két nagyobb részre bontható: vagy az algoritmus kimenetét teszi kisebbé, vagy a redukció folyamatát könnyedebbé.

A redukciós algoritmusok nem globális optimum elérésére törekszenek, a lokális optimum tulajdonsága, hogy potenciálisan mindig lehet jobbat találni. Ez a tulajdonság került kihasználásra. A DDMIN esetében egy fixpont iteráció használatát javasoltuk: amíg egy algoritmus redukálni képes, addig iteráljuk azt; a következő iteráció bemenete az előző eredménye. Legalább két iteráció mindig lefut és átlagosan 68%-kal lett kisebb a kimenet karakteralapú redukciót használva. HDD-nél a bemenetet képző fán lettek transzformációk javasolva. Javasoltuk a Hoist transzformációt, ami a fa egy csomópontját képes helyettesíteni egy másikkal abban az esetben, ha a két csomópont címkéje megegyezik és ős-leszármazott viszonyban vannak egymással.

Az algoritmusok futása során előfordul, hogy több konfiguráció ugyanazt a tesztesetet eredményezi, így többször újra kell futtatni őket. Ezt oldja meg a gyorsítótár alkalmazása, ami tárolja a teszteseteket és azok eredményeit, a duplikált futtatásokat kiküszöbölve. A gyorsítótár viszont sok memóriát használhat fel. A memóriafogyasztás csökkentésére több megoldást is javasol a tanulmány, melyek együttesen 96%-os javulást értek el a DDMIN-nél és 85%-os javulást a HDD-nél. Az futásidő csökkentésére jó opció lehet a párhuzamosítás, melynek koncepcióiban volt optimalizálható elem: stabilizációs problémák kerültek ismertetésre, majd egy megoldás került felvázolásra. Javasolva lett egy mohó megközelítés, mely csökkentette a szükséges tesztvégrehajtások számát 31%-kal és a futásidőt 40%-kal.

Nyilatkozat

Vince Dániel "**Enhancements of Automated Test Case Reduction**" című PhD disszertációjában a következő eredményekben **Vince Dániel** hozzájárulása volt a meghatározó:

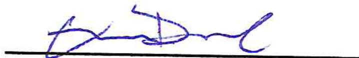
- **Cache Optimizations**: A szerző elemezte a redukcióban alkalmazott korszerű gyorsítótárazási megoldásokat, majd három optimalizálást tervezett és valósított meg a redukció memóriafelhasználásának csökkentése érdekében. Ezt követően kiértékelte a különböző redukciós megközelítésű javaslatok hatásait több teszthalmazon.
- **Iterating the Minimizing Delta Debugging Algorithm**: A szerző megtervezte és prototípusként elkészítette a DDMIN algoritmus fixpontos iterációját.
- **Parallel Optimizations of DDMIN***: A szerző elemezte a párhuzamos DDMIN gyengeségeit, majd megtervezte, megvalósította és kiértékelte a megoldást.
- **Extending Hierarchical Delta Debugging with Hoisting**: A szerző a HDD bemeneteinek (absztrakt szintaxisfák) szerkezetét vizsgálta, optimalizálási lehetőségeket keresve. Megállapította, hogy az azonos címkével ellátott csomópontok a szintaktikai helyesség elvesztése nélkül cserélhetők. Ezután elkészítette a transzformáció-alapú minimalizálási keretrendszer prototípusát, példatranszformációként megvalósította az emelést, majd nyilvánosan elérhető tesztcsomagokon kiértékelte.

A következő felsorolásban lévő publikációk tartoznak ehhez a tézisponthoz:

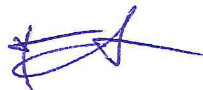
1. **Dániel Vince** and Ákos Kiss. Cache Optimizations for Test Case Reduction. In *Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS 2022)*, pages 442-453, Guangzhou, China (Virtual), December 2022. IEEE.
2. **Dániel Vince**. Iterating the Minimizing Delta Debugging Algorithm. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST'22)*, pages 57-60, Singapore, November 2022. ACM.
3. **Dániel Vince** and Ákos Kiss. Evaluation of the fixed-point iteration of minimizing delta debugging. In *Journal of Software: Evolution and Process*, 2024. Wiley.
4. **Dániel Vince** and Ákos Kiss. GreeDDy: Accelerate Parallel DDMIN. In *Proceedings of the 15th ACM International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST '24)*, pages 1-4, Vienna, Austria, September 2024. ACM.
5. **Dániel Vince**, Renáta Hodován, Daniella Bársony, and Ákos Kiss. Extending Hierarchical Delta Debugging with Hoisting. In *Proceedings of the 2nd ACM/IEEE International Conference on Automation of Software Test (AST 2021)*, pages 60-69, Madrid, Spain (Virtual), May 2021. IEEE.
6. **Dániel Vince**, Renáta Hodován, Daniella Bársony, and Ákos Kiss. The effect of hoisting on variants of Hierarchical Delta Debugging. In *Journal of Software: Evolution and Process*, 34(11):e2483:1-e2483:26, November 2022. Wiley.

Ezek az eredmények **Vince Dániel** PhD disszertációján kívül más tudományos fokozat megszerzésére nem használhatók fel.

Szeged, 2024.12.02



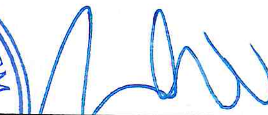
Vince Dániel
jelölt



Dr. Kiss Ákos
témavezető

Az Informatika Doktori Iskola vezetője kijelenti, hogy jelen nyilatkozatot minden társszerzőhöz eljuttatta, és azzal szemben egyetlen társszerző sem emelt kifogást.

Szeged, 2024. 12.12.



Dr. Jelasity Márk
doktori iskola vezető