Ghassan Shakib Misherghi
June 2007
Computer Science

Hierarchical Delta Debugging

Abstract

Inputs causing a program to fail are usually large and often contain information irrelevant to the failure. It thus helps debugging to simplify program inputs. The Delta Debugging algorithm is a general technique applicable to minimizing all failure-inducing inputs for more effective debugging. In this thesis, we present HDD, a simple but effective algorithm that significantly speeds up Delta Debugging and increases its output quality on tree structured inputs such as XML. Instead of treating the inputs as one flat atomic list, we apply Delta Debugging to the very structure of the data. In particular, we apply the original Delta Debugging algorithm to each level of a program's input, working from the coarsest to the finest levels. We are thus able to prune the large irrelevant portions of the input early. With the careful application of our algorithm to any particular input language, HDD creates only syntactically valid variations of the input, thus reducing the number of inconclusive configurations tested and accordingly the amount of time spent simplifying. We also demonstrate a general framework requiring only the input language's context-free grammar to automatically simplify input while ensuring that all test cases are syntactically valid. This framework proceeds by calculating correction strings of minimal length that can replace removed nodes from the input parse tree. We have implemented HDD and evaluated it on a number of real failure-inducing inputs from the GCC and Mozilla bugzilla databases. To demonstrate generality and the application of our framework, we have also evaluated it on a Java program injected with an error. Our Hierarchical Delta Debugging algorithm produces simpler outputs and takes orders of magnitude fewer

test cases than the original Delta Debugging algorithm. It is able to scale to inputs of considerable size that the original Delta Debugging algorithm cannot process in practice. We argue that HDD is an effective tool for automatic debugging of programs expecting structured inputs.

Professor Zhendong Su
Thesis Committee Chair

Hierarchical Delta Debugging

By

GHASSAN SHAKIB MISHERGHI
B.S. (University of California, Davis) 2004

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Zhendong Su, Chair

Professor Premkumar Devanbu

Professor Ronald Olsson

Committee in charge

2007

Acknowledgments

For the creation of this thesis and the course of my graduate studies, many individuals have been instrumental to my accomplishments. The support that they provided was a necessity to my individual research successes and my overall scholarly maturity. I thank them warmly for their patience, their valuable instruction, and their example.

I thank Professor Zhendong Su for being an exemplary adviser. Professor Su has been an attentive mentor, always making his advice available on even the shortest of notice. Beyond individual support, he has encouraged a community environment among students, ultimately guiding us to become better professionals. I have learned from him that working intelligently with awareness is a critical component for scientific contribution. He has very judiciously evaluated and suggested research ideas, offering much encouragement for those that were attainable and had potential. A graduate student can expect no better an adviser than one such as he.

I thank Professor Premkumar Devanbu for his consistent encouragement, support, and feedback. His instruction very contagiously instills a sense of intellectualism and academic appreciation. He also demonstrates a skill all students would be wise to learn: how to find humor in our work while maintaining a focused demeanor.

I thank Professor Ronald Olsson for his instruction and feedback. His emphasis on formalism and precision are requisite to a quality graduate education in computer science. Few instructors invest the level of detail and attention to their curriculum. His exceeding efforts did not go unnoticed; ultimately they raised my expectations for my own work.

I thank Earl Barr for his encouragement and camaraderie. Earl Barr very honestly showed me the reality of life for a graduate student. His ability to make a convincing argument for graduate studies despite this is a testament to his persuasive powers. Cynicism aside, I am very lucky to have made such a considerate friend.

Many others have also been kind and supportive to me. Of note are Professor Charles Martel, Professor Raju Pandey, Lingxiao Jiang, Christian Bird, and Gary Wassermann. I thank them and wish them the best of success in their endeavors. I hope those who I've forgotten to name will be kind enough to forgive me.

Contents

List of Figures

List of Tables

Abstract

Inputs causing a program to fail are usually large and often contain information irrelevant to the failure. It thus helps debugging to simplify program inputs. The Delta Debugging algorithm is a general technique applicable to minimizing all failure-inducing inputs for more effective debugging. In this thesis, we present HDD, a simple but effective algorithm that significantly speeds up Delta Debugging and increases its output quality on tree structured inputs such as XML. Instead of treating the inputs as one flat atomic list, we apply Delta Debugging to the very structure of the data. In particular, we apply the original Delta Debugging algorithm to each level of a program's input, working from the coarsest to the finest levels. We are thus able to prune the large irrelevant portions of the input early. With the careful application of our algorithm to any particular input language, HDD creates only syntactically valid variations of the input, thus reducing the number of inconclusive configurations tested and accordingly the amount of time spent simplifying. We also demonstrate a general framework requiring only the input language's context-free grammar to automatically simplify input while ensuring that all test cases are syntactically valid. This framework proceeds by calculating correction strings of minimal length that can replace removed nodes from the input parse tree. We have implemented HDD and evaluated it on a number of real failure-inducing inputs from the GCC and Mozilla bugzilla databases. To demonstrate generality and the application of our framework, we have also evaluated it on a Java program injected with an error. Our Hierarchical Delta Debugging algorithm produces simpler outputs and takes orders of magnitude fewer test cases than the original Delta Debugging algorithm. It is able to scale to inputs of considerable size that the original Delta Debugging algorithm cannot process in practice. We argue that HDD is an effective tool for automatic debugging of programs expecting structured inputs.

Professor Zhendong Su
Thesis Committee Chair

Chapter 1

Introduction

Programmers spend a significant amount of their time debugging programs. Studies consistently show that software maintenance typically requires more time than any other programming activity [16]. For an exhibited bug, programmers must determine which portions of a given test case induce a program failure. This search phase of the debugging process is slow and arduous. Once the program's errant behavior is finally understood, the bug is often quickly fixed.

Many times, a programmer is given a large test case that produces a failure. Reducing this test case simplifies the debugging process because there are fewer irrelevant details contained within the test case, allowing the programmer to focus on issues pertinent to the failure. Minimizing test cases has traditionally been left to humans.

1.1 Delta Debugging

Delta Debugging, a technique by Zeller and Hildebrandt, is an approach for automating test case minimization [26]. It consists of two algorithms:

Simplification: In this algorithm, the failure-inducing input is simplified by examining smaller configurations of the input. The algorithm recurses on the smaller failure configurations until it cannot produce a smaller configuration that still produces a failure; and

Isolation: This algorithm attempts to find a passing configuration such that with the

addition of some element it becomes a failing configuration. The algorithm works in both directions, finding bigger passing cases that are subsets of a failing case.

Isolation produces outputs which are less intuitive as a debugging aid for the programmer. The single element difference is not individually responsible for the failure, it merely guarantees that some symptom is exhibited. The programmer then has to sift through the potentially large failure-inducing configuration to determine what else may be responsible for the failure. Although isolation may generally be faster than simplification, for large test cases, it may lead to worse running times because of the time spent testing the large configurations. In this thesis, we will be mainly concerned with the first algorithm, simplification. More details on Delta Debugging are given in Section 3.1.

1.2 Hierarchical Delta Debugging

Input data is often structured hierarchically; however, Delta Debugging ignores input structure and may attempt many spurious input configurations. Our insight is that the existing input structure can be exploited to generate fewer input configurations and simpler test cases for more effective automated debugging. In this paper, we propose a *Hierarchical Delta Debugging* algorithm, HDD, to validate our hypothesis.

There are numerous examples of input data with recursive definitions. When input is nested and at least partially balanced, there is temptation to take advantage of this in our test case minimization algorithm. We present several examples of input data for which Hierarchical Delta Debugging is applicable. In the most general case, any data defined by a context-free grammar is a good candidate for Hierarchical Delta Debugging. If a context-free grammar is necessary for the definition of a particular language, a simple regular language cannot suffice. The data set is thus likely to be nested, giving us an advantage over standard Delta Debugging. We give below a few concrete scenarios where Hierarchical Delta Debugging may be applied:

Programming Languages: Programs can make use of a Hierarchical Delta Debugger when considered as input into a compiler or interpreter. If a large program causes a

failure in a specific compiler, the Hierarchical Delta Debugger can operate over the program's abstract syntax tree (AST). The minimum configuration is found at all levels of the AST. For example, the algorithm first finds the minimum configuration of classes, prototypes, global variables, and top level functions. It then recurses into methods, statements, and local declarations. Following this, it determines the minimum configuration of sub-statements and expressions. This process is performed to the lowest level of the AST. Declarations demonstrate that there are cases when some nodes depend on higher non-ancestral nodes within the AST. We will later present an approach to solve this problem.

HTML/XML: HTML and XML are also excellent candidates for Hierarchical Delta debugging. These languages, as generated both by humans and machines, are widely deployed. In both cases the inputs tend to be well nested. XML can benefit much from a syntactically aware algorithm because it adheres to a strict grammar. Furthermore, XML is meant to be general, thus a single implementation may be applicable to many program inputs.

Video Codecs (compressor/decompresser): Hierarchical Delta Debugging can be applied successfully to cases when data is of limited depth. Consider a simple video codec that crashes on a particular video sequence due to a subset of the frames not necessarily immediately neighboring one another. Suppose that the encoding scheme consists of several groups containing a single key frame and multiple delta frames. Our algorithm first selects the minimum configuration of groups in the failing sequence. It then recurses one level below to determine which delta frames are inducing the failure. Reaching the result is significantly faster than a flat implementation since we have removed the irrelevant groups first before considering their individual frames.

UI Interactions: User interfaces also demonstrate a potential use for Hierarchical Delta Debugging. GUI applications may crash after complicated user sessions. The programmer benefits from a minimized session in that the failure can be reproduced and examined in a simpler form. UI interactions can be categorized into levels of tasks, *e.g.*, "open the document" and "print." These high level tasks can be composed of

other tasks such as "opening the print dialogue," "selecting the printer," "configuring the printer," and "selecting the pages." At the lowest level of this task hierarchy are the actual actions required to perform these tasks, including mouse movements and other events. With this hierarchy, one can apply Delta Debugging to find the minimum number of interactions that induce some error. Unfortunately, automating the creation of this hierarchy is difficult; a user must categorize various actions manually, perhaps inside the interaction recording program.

1.3 Main Contributions

In this thesis, we develop a general Hierarchical Delta Debugging algorithm to exploit hierarchical characteristics of program inputs. When attempting to find which portions of an input to prune, there is little reason to choose arbitrary points for division. Instead, we work along the boundaries of the tree from the top to the bottom. By limiting ourselves to one level at a time, we are examining smaller groups of nodes for minimization. This also exploits the relative independence of nodes on different branches of the tree.

Our algorithm produces only syntactically valid configurations. While this seems counter to the purpose of failure finding, it is indeed consistent. The goal of the algorithm is to determine a failure-inducing input that should be valid to the program. Parsing related issues are not the problem addressed by this thesis. If our input is not well-formed, techniques such as Delta Debugging should be employed to determine why the parser does not fail gracefully. In many cases, spurious test configurations that fail at the parse level before triggering our desired bug waste testing time.

The original Delta Debugging algorithm produces test cases that can be difficult to read by minimizing things such as identifiers, and removing section boundaries. This is often counter to the purpose of making the debugging process easier. Our approach can make the minimized input more closely resemble the original structure. A developer may find this easier to understand because of its similarity to a real test case.

We have implemented our algorithm and applied it in three settings: (1) bugs in the GCC compiler, (2) bugs in the Mozilla web browser for XML processing, (3) and

reduction of Java source files. On real failure-inducing inputs from the GCC and the Mozilla bugzilla databases, our algorithm generates orders of magnitude fewer test cases and produces simpler outputs than the original Delta Debugging algorithm, resulting in significantly faster minimization time. This evaluation confirms the effectiveness of our Hierarchical Delta Debugging algorithm when applied to programs accepting structured inputs.

1.4 Thesis Outline

The rest of the thesis is structured as follows. We first use a concrete example to explain the intuition of our Hierarchical Delta Debugging algorithm (Chapter 2). Next, we present our algorithm (Chapter 3), followed by an empirical evaluation to compare our technique to the original delta debugging algorithm (Chapter 4). Then, we present a general framework facilitating the preservation of syntactic validity (Chapter 5). Finally, we survey closely related work (Chapter 6), document our released tools (Chapter 7), and conclude (Chapter 8).

Chapter 2

Motivating Example

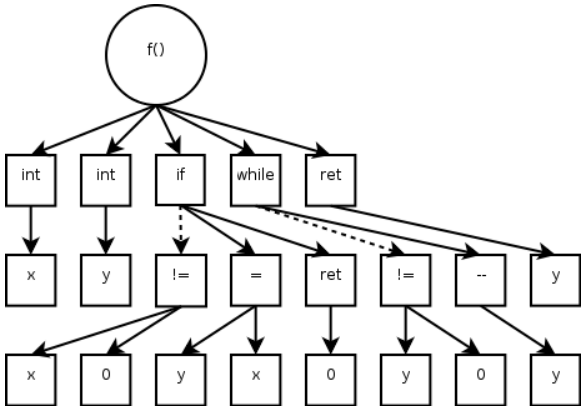
In this chapter, we demonstrate a contrived C program to illustrate our algorithm. Details of the algorithm will be given in Chapter 3.

The first step of our algorithm is to parse a failure-inducing input. Using the parse tree, we manipulate the input and generate new test cases. Consider the contrived program shown in Figure 2.1. The parsed input should be the AST of the program. Figure 2.2a shows the AST of the program in Figure 2.1. In this example, there is only one function causing the compiler to fail. Let us assume that the post-decrement operator, "`y--`," is the source of the failure.

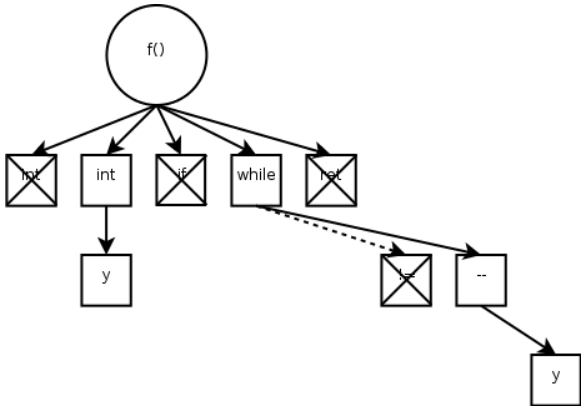
Our algorithm begins processing at the top level of the parse tree. We first determine the minimum configuration of functions, global variables, prototypes, and type definitions. There is only one top level form in our example, the function `f()`. Our algorithm will first try to exclude this function from the configuration. The function is required to induce the compiler error; it will thus be included in the top level's minimum configuration.

```
void f()
{
    int x;  int y;
    if (x!=0) { y= x; } else { return 0; }
    while (y!=0) { y--; }
    return y;
}
```

Figure 2.1: An example program.



(a) Figure 2.1's AST.



(b) The result of our algorithm on Figure 2.2a.

Figure 2.2: Applying Hierarchical Delta Debugging to the code in Figure 2.1.

Once the minimum configuration of a particular level is determined, we try the next level of the AST. Note that there may be multiple parents for all the nodes at a level. We use the standard Delta Debugging algorithm [26] to determine the minimum configuration of all these nodes. Each subtree could instead be treated individually, yielding an algorithm similar to a pre-order tree traversal. For many languages, our current approach is adequate, and so we leave this as future work.

Figure 2.2b shows the result of the Hierarchical Delta Debugging algorithm applied to the original AST in Figure 2.2a. The minimized AST corresponds to the following code:

```
f() {  int y;   while (0) { y--; }  }
```

We stated previously that the post-decrement operator was the source of the com-

piler failure. Note that the minimum nodes relevant to this error are chosen for any level. For example, if the variable `y` were not declared at the second level, our program would not type check and thus the compiler may not exhibit the failure. The `while` statement on the second level is of particular interest. The condition of the while statement was deemed irrelevant. Handling this case correctly is left to the tree manipulator; ours inserted an empty condition, `0`, in-place of the original condition. Some implementations may attempt to replace the loop with its body. There are several nodes in the AST that have required children. To produce syntactically valid inputs, we must handle removal of their children on a case by case basis, most likely by substituting in the smallest allowable syntactic fragment.

Chapter 3

Hierarchical Delta Debugging

3.1 Background on Delta Debugging

Before presenting HDD in detail, we explain the simplifying Delta Debugging algorithm [26], hereafter called ddmin, as it is integral to HDD. The input to the ddmin algorithm is a failure-inducing configuration, *i.e.*, a list of elements that causes a program to fail when given as input. The goal of the ddmin algorithm is to determine a subset of the input such that no one element can be removed from it while preserving the failure. Zeller and Hildebrandt call this a *1-minimal test case* [26]. The ddmin algorithm proceeds by executing the following steps to find a 1-minimal test case:

1. *Reduce to subset:* Split the current configuration into n partitions. Test each partition for failure. If a partition does induce the failure, then treat it as the current configuration and resume at Step 1.
2. *Reduce to complement:* Choose any single partition, omit it, and test with the remainder of the configuration. If any of the configurations induces the failure then treat it as the current configuration and resume at Step 1.
3. *Increase granularity:* Try splitting the current configuration into smaller partitions, $2n$ if possible, where n is the current number of partitions. Resume at Step 1 with the smaller partitions. If the configuration cannot be broken down into smaller partitions, the current configuration is 1-minimal and the algorithm terminates.

Each split is chosen to produce sub-partitions of similar size to remove as much as possible from the input. This characteristic of the ddmin algorithm makes it much like a binary search. If each split is successful, we can reduce the input by at least half each iteration (assuming "reduce to subset"). Unfortunately this is not likely to occur because the input is not split according to the structure of the failure-inducing configuration. Furthermore, the failure-inducing portions of the file may also be scattered throughout the file. Our approach addresses these issues better when simplifying structured inputs.

3.2 Algorithm Description

We are concerned with hierarchical inputs. Why not first determine the minimal failure-inducing configuration of coarse objects? Following this intuition, we can recursively minimize configurations starting from the top-most level. By limiting simplification to one level of the hierarchy at a time, we can prune data more intelligently. At each level we must try multiple configurations to determine the minimum failure-inducing configuration. This process employs the ddmin algorithm at each level.

Before a level is processed, we must first know how many nodes there are in a level, and name each node. Without addressing nodes we cannot compactly select a configuration for testing. Node naming is implemented by traversing the tree and assigning a unique identifier to each node at the level of interest. Nodes are labeled only before we process their levels.

Unparsing a configuration of the tree is required for each attempted test configuration. This is the most frequent operation executed by HDD. Our implementation traverses the tree, checking for node inclusion in the configuration before printing a particular node.

After determining the minimum configuration at a level, one approach to refining the input is to output the new configuration and re-parse it for further processing. This approach, though simple, is not ideal since we spend time re-parsing needlessly. We eliminate this step by providing primitives for tree pruning. This is implemented by removing all irrelevant nodes from a level. Following tree pruning we proceed to the next level. Figure 3.1 shows the first iteration of the Delta Debugging algorithm applied to the AST in Figure 2.2a.

Algorithm 1 The Hierarchical Delta Debugging Algorithm

```
1: procedure HDD(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:         level ← level + 1
8:         nodes ← TAGNODES(input_tree, level)
9:     end while
10: end procedure
```

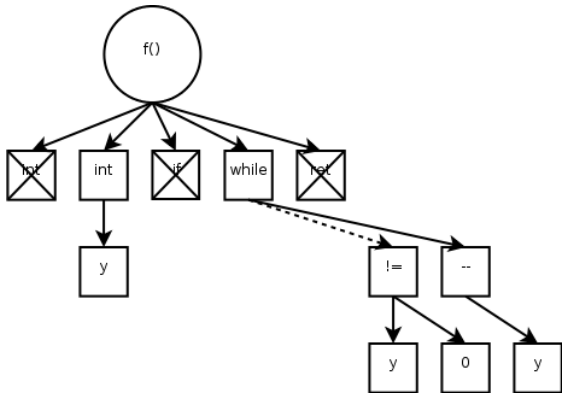


Figure 3.1: The AST after the first iteration of HDD.

Algorithm 1 shows HDD's pseudocode. Line 2 begins the process at the top level of the tree (where $level = 0$). We count and tag the nodes at the current level on lines 3 and 8 with an auxiliary function TAGNODES. If there are nodes at the current level (line 4), we try minimizing the nodes of that level using the standard Delta Debugging algorithm, DDMIN (line 5). Although not shown in the algorithm, ddmin must call a testing procedure to determine if a configuration induces the failure. This procedure must be given the input tree and the level currently under inspection. Line 6 removes the irrelevant nodes from the tree with the auxiliary function PRUNE. On line 7, we progress to the next level of the tree for processing.

3.3 Algorithm Complexity

We now discuss HDD's complexity with respect to the number of tests performed. Our HDD algorithm has the same worst-case complexity as ddmin, generating quadratic number of tests in the size of the original input. Our worst-case happens when the input is a flat list, essentially reducing HDD to one call to ddmin on the entire input.

For inputs with nested structures such as programs, we expect HDD to perform much better. We empirically validated this claim using a few real-world examples, and we delay that discussion to Chapter 4. In this section, we instead examine the performance of HDD with specific input characteristics. We consider the following two scenarios:

The Ideal Case: Suppose HDD is run on a balanced tree of size n with constant branching factor b such that for each parent *exactly one* child remains in the configuration. There are $\log_b n$ levels in this tree. At each level, we invoke ddmin, and thus require $O(b^2)$ tests. Consequently, we run $O(b^2 \log_b n)$ tests, or simply $O(\log n)$, because b is constant.

A More Realistic Case: The above scenario is too idealistic for many cases. Instead let us suppose that exactly m children are relevant from each relevant parent. While this is unlikely to occur in practice, we may consider m to be an upper bound on the number of children that are taken in all cases. We cannot choose more than b nodes from every parent; consequently we must have $m \leq b$. Intuitively, it is clear that the number of nodes to examine becomes fairly large further down the tree.

At the top level we examine the root of the tree. For any subsequent level, i, we examine b nodes for every included parent, or $b\left(m^{i-1}\right)$ nodes. We run ddmin at each level, therefore we try at most $O\left((b\left(m^{i-1}\right))^2\right)$ tests at level i. Summing over the

entire tree, we have:

$$\begin{aligned} & 1 + \sum_{i=1}^{log_b n} \left(b * m^{i-1}\right)^2 \\ = \;& 1 + b^2 \sum_{i=1}^{log_b n} \left(m^{i-1}\right)^2 \\ \leq \;& 1 + b^2 \left(\sum_{i=1}^{log_b n} m^{i-1}\right)^2 \\ = \;& 1 + b^2 \left(\frac{m^{log_b n} - 1}{m - 1}\right)^2, \text{ when } m \neq 1. \end{aligned}$$

Given m and b, we conclude that HDD runs worst-case:

$$O\left(\left(\frac{m^{log_b n} - 1}{m - 1}\right)^2\right) = O\left(m^{2log_b n}\right)$$

number of tests, when $m \neq 1$. Substituting 1 for m in the initial summation above, we have $1 + b^2 log_b n$. Thus, we run $O(log_b n)$ tests, which is consistent with the idealistic example above. If we choose all b nodes from each parent, *i.e.*, when $m = b$, we have $O(b^{2log_b n}) = O(n^2)$. Thus, we run absolute worst-case $O(n^2)$ tests. For a specific case in between, such as $b = 4$ and $m = 2$, we have $O(2^{2log_4 n}) = O(n)$.

Though the relative benefit gained by HDD depends on the shape of the input tree, we never perform asymptotically more tests than the original Delta Debugging algorithm. If the tree is well balanced, we can expect large portions of the original input to drop whenever we remove a node high in the tree. This property is what enables us to achieve better asymptotic bounds.

3.4 On Minimality

We now examine the problem of simplifying failure-inducing inputs from a more formal perspective. In particular, we investigate the quality of produced output from a simplifying algorithm. The most natural metric for a generated output is its size, and the obvious notion to consider is *minimality*: How small is the output? Zeller and Hildebrandt propose various definitions for minimality in the context of ddmin [26]. We introduce similar

definitions for trees instead of sequences of elements as used in their definitions. First, we define the meaning of tree simplification in terms of two predicates *simplify* and *simplify*$*$.

Definition 3.4.1 (simplify)**.** For any two trees T and T', the predicate $simplify(T, T')$ holds iff T' can be derived from T by removing a single node.

Definition 3.4.2 (simplify*)**.** For any two trees T and T', the predicate $simplify*(T, T')$ holds iff T' can be derived from T by removing zero or more nodes, and more precisely:

- $simplify*(T, T)$; and
- $simplify*(T, T') =$

$$\exists T'' \left(simplify(T, T'') \wedge simplify*(T'', T') \right)$$

Definition 3.4.3 (Global-Tree-Minimal)**.** Given program P and input tree T, T' is *global-tree-minimal* if $simplify*(T, T')$ and $P(T') = \text{fail}$, and for all T'' such that $simplify*(T, T'')$ and $P(T'') = \text{fail}$, it holds $|T'| \leq |T''|$.

Ideally, we would want an algorithm that finds a *global-tree-minimal* input that induces the failure. Unfortunately, we will show next that this is infeasible in general. Our notion of *global-tree-minimality* coincides with Zeller and Hildebrandt's notion of global-minimality, because a tree is merely a hierarchy of the actual input configuration. *Global-tree-minimality* is a very difficult problem computationally; we show that the problem is NP-complete. First, we formulate the decision version of this problem, GMT, where we treat the given program P as a constant-time oracle because we are interested in the number of generated test configurations.

Definition 3.4.4 (GMT)**.** *Given program P, a failure-inducing input T, and a positive integer K, is there a tree T' with $|T'| \leq K$, such that $simplify*(T, T')$ and $P(T') = fail$?*

Theorem 3.4.5. GMT is NP-complete.

Proof. GMT is clearly in NP, because given a GMT instance (P, T, K), we can guess a configuration S from T, and verify both $|S| \leq K$ and $P(S) = \text{fail}$, all in polynomial time.

To show that GMT is NP-hard, we reduce the hitting set problem (HS), which is known to be NP-complete [13], to GMT. Recall that in the hitting set problem, we are given a collection C of subsets of a finite set S and positive integer $K \leq |S|$, and the question is: Is there a subset $S' \subseteq S$ with $|S'| \leq K$ such that S' contains at least one element from each subset in C?

Given an instance (C, S, K) of HS, we construct an instance (P, T, K) of GMT as follows:

- $P(t) =$ fail iff for each $c \in C$, there exists a child s of t such that $s \in c$; and
- $T = \mathsf{root}(s_1, \ldots, s_n)$, where $\mathsf{root}(s_1, \ldots, s_n)$ denotes a tree with root as the root and $s_i \in S$ as its children.

This is clearly a polynomial time reduction. It is also straightforward to verify that (C, S, K) has a hitting set S' with $|S'| \leq K$ if and only if (P, T, K) has a failure-inducing input T' with $|T'| \leq K$. Here we measure the size of a tree as the number of children under the root node "root." □

Global-minimality is thus a difficult problem; neither HDD nor ddmin can claim to produce global-minimal configurations. Instead of attempting to achieve such an elusive goal, we will attain minimality with respect to the immediate "neighbors" of a failing configuration. With such a goal, we can merely examine all neighbors of the current failing configuration until none induce the error. Then we have attained a local-minimal input. Program failures tend to mirror this: Some portion of the input is relevant to the failure, the rest can be removed piece by piece.

Definition 3.4.6 (1-Tree-Minimal)**.** Given a program P and input tree T, T' is *1-tree-minimal* if *simplify*$^*(T, T')$ and $P(T')$ produces a failure, and for all S such that *simplify*(T', S), $P(S) \neq$ fail.

Before evaluating HDD's output with respect to such a property, let us discuss minimality with ddmin. DDmin's behavior at the finest granularity is what allows it to assure 1-minimality: it tries all individual elements of the configuration alone, or tries to remove single elements from the current configuration until no single element can be

removed. By definition, 1-minimality is assured since the algorithm terminated and thus there is no single element that can be removed. Our approach constrains ddmin to subsets of the failure-inducing configuration. When we call ddmin to remove nodes at a specific level, we may enable nodes at some other level of the tree to be removed.

Our algorithm will not re-attempt to remove nodes on levels higher than the current level, hence HDD may not always produce 1-minimal configurations. Any algorithm which does not produce a 1-minimal configuration does not produce a *1-tree-minimal* configuration either, since the elements in the configuration must also be nodes in the tree. We now demonstrate several alternate algorithms derived from HDD that satisfy 1-minimality or *1-tree-minimality*.

Suppose all internal nodes of the input tree represent only collections of other nodes and do not contribute any of the actual elements that makeup the program's failure-inducing configuration. Consider non-terminals from context-free grammars as a good example of this. If we ensure that all leaf nodes are put on the last level, then HDD's final call to ddmin will include all elements in the configuration. This guarantees that the final result is 1-minimal with respect to these individual elements, by merely relying on ddmin to do so. By considering trees of this type, HDD alone is sufficient for 1-minimality. This approach does not adequately take advantage of the hierarchical nature of HDD: the final output may not be *1-tree-minimal*. For example, consider a C program inducing some failure in a C compiler. Assume that the compound statement "{}" is inside the aforementioned program and is irrelevant to inducing the failure. Removing just one of the contained braces causes a parse error, potentially preventing the compiler from inducing the failure. Removing the entire statement from the program may still induce the failure. If a tree has a single parent node for the compound statement, then a *1-tree-minimal* input cannot contain the compound statement.

We now present an algorithm, HDD+, consisting of a greedy phase and a final *1-tree-minimality* assurance phase. First we perform the greedy phase: a call to HDD on the input configuration tree. This phase will attempt to trim the tree as best it can without consideration for *1-tree-minimality*. The second optimality oriented phase is symmetric with the final step of ddmin. It will try to remove individual nodes from the tree one by one in

a BFS-like manner. It will repeat the attempt on the entire tree continually as long as the previous iteration successfully removed at least a single node. This algorithm produces a *1-tree-minimal* configuration by definition: the final input tree does not have a single node that can be removed, otherwise the algorithm would not have terminated. In the worst case, the entire algorithm generates $O(n^2)$ number of test inputs on trees of size n. As we have previously shown, HDD generates worst-case $O(n^2)$ test inputs. Each iteration of the second phase requires at most n tests, and since it removes at least one element from the tree each iteration, it cannot iterate more than n times. It follows that the second phase also generates worst-case $O(n^2)$ test inputs, affirming our analysis.

We present another algorithm for *1-tree-minimality*, HDD*. This algorithm repeatedly calls HDD until a single call fails to remove a single element from the input tree. Since ddmin will attempt to remove every node in the tree individually at least once, HDD will attempt to remove every node in the tree individually at least once. If HDD cannot remove a single node from the tree, the tree is *1-tree-minimal*. It is possible that each iteration of this algorithm removes only one node from the tree, and since each iteration generates worst-case $O(n^2)$ test inputs, it follows that HDD* generates worst-case $O(n^3)$ test inputs. We find this is very unlikely in practice. Implementing HDD* is simple with an already working HDD implementation, so for comparison we included it in our empirical results. We hope by our evaluation to demonstrate two conclusions: (1) worst-case analysis for HDD, ddmin, and HDD* does not reflect what happens in practice, and (2) 1-minimality is not necessarily the best criteria for input minimization.

Proposition 3.4.7. Both HDD+ and HDD* produce *1-tree-minimal* configurations.

We suspect HDD and HDD* may produce output close to the global minimum for practical settings. We leave this as future work to validate this claim.

Chapter 4

Empirical Evaluation

We have evaluated HDD in two settings: bugs in the GCC compiler and bugs in XML processing of the Mozilla web browser. In this chapter, we discuss our experience with the algorithm.

4.1 The C Programming Language

We created a tree processing module for the C programming language in order to test various GCC failure-inducing programs. Our AST simplifier was implemented as an extension to Elsa [18], a GLR parser for C++. We augmented Elsa's AST nodes with methods and data to facilitate reduction. There was little inheritance among AST nodes in the Elsa framework, forcing us to duplicate functionality more than we should have. For example, an `if` statement has specific variables for each of its three children: the condition, the `then` branch, and the `else` branch. A `for` loop, on the other hand, has entirely different variables for its children: an `initializer`, a `condition`, a `post-incrementor`, and a `body`. Although the process of printing and pruning is similar for both of these statements, their implementation could not be adequately factored. This lack of generality made implementation tedious. We examine a general approach to language simplification in Chapter 5.

We now apply HDD to a concrete example. To simplify comparison, we choose the same program demonstrated in Zeller and Hildebrandt's work on delta debugging [26].

```
double mult( double z[], int n )
{
    int i;
    int j;
    for (j= 0; j< n; j++) {
        i= i+j+1;
        z[i]=z[i]*(z[0]+0);
    }
    return z[n];
}

int copy(double to[], double from[], int count)
{
    int n= (count+7)/8;
    switch (count%8) do {
        case 0: *to++ = *from++;
        case 7: *to++ = *from++;
        case 6: *to++ = *from++;
        case 5: *to++ = *from++;
        case 4: *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
    } while (--n > 0);
    return (int)mult(to,2);
}

int main( int argc, char *argv[] )
{
    double x[20], y[20];
    double *px= x;

    while (px < x + 20)
        *px++ = (px-x)*(20+1.0);

    return copy(y,x,20);
}
```

Figure 4.1: A program that crashes GCC-2.95.2.

Figure 4.1 shows a program that causes GCC-2.95.2 to fatally abort, even though the poorly written program is valid C code. The problem lies within the `for` loop of the function `mult()`, where an incorrect floating point optimization causes GCC to crash.

The Hierarchical Delta Debugging algorithm first determines that the only relevant

```
double mult(double *z,
int n)
{
  int i;
  int j;
  for (j=0;j<n;j++) {
    i=i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
  return z[n];
}
```

(a) The minimized 1st level.

```
mult(double *z, int n)
{
  int i;
  int j;
  for (j=0;j<n;j++) {
    i=i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
}
```

(b) The minimized 2nd level.

```
mult(double *z, int n)
{
  int i;
  int j;
  for (;;) {
    i=i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
}
```

(c) The final output.

Figure 4.2: HDD applied on various levels of the code in Figure 4.1.

function is `mult()` by calling ddmin on the first level of the code's AST (Figure 4.2a). Next, it removes the return type, `double`, and the `return` statement because they are also irrelevant for the bug (Figure 4.2b). At the next level, the algorithm determines that the loop initializer, condition, and post-incrementor are not necessary to induce the failure (Figure 4.2c). The algorithm then unsuccessfully attempts to simplify the expressions inside the two assignment statements. It cannot produce a smaller configuration from these expressions because they are integral to inducing the failure, and HDD terminates with the program shown in Figure 4.2c. Reaching this output required 86 test cases, significantly fewer than the 680 tests performed by ddmin.

To further test the performance of HDD, we applied it to other real programs from the GCC bugzilla database (`http://gcc.gnu.org/bugzilla`). Our examples exhibited a number of characteristics, each highlighting interesting results. We selected the first three examples based only on our ability to reproduce failures within our limited testing environment. Table 4.1 shows our empirical results. All experiments were run on an Intel Pentium 4 (Xeon) with 2GB of RAM running Linux 2.6.17. For each program in the table, we list size, bug report number from the bugzilla database, and the number of tests and final sizes as computed by ddmin, HDD, and HDD*. We use a token as our unit of size because a token abstracts details such as identifier length and whitespace. Our algorithm

File	size (tokens)	report (id)	algorithm	tests (#)	size (tokens)	time (sec)	space (KB)
bug.c	277	[26]	ddmin	680	53	14	8536
			HDD	86	51	3	2822
			HDD*	164	51	5	3489
boom7.c	420	663	ddmin	3727	102	121	204652
			HDD	144	57	2	2924
			HDD*	304	19	4	3489
cache.c	25011	1060	ddmin	1743	62	56	1406484
			HDD	191	61	10	3674
			HDD*	327	58	12	3698
cache-min.c	145	1060	ddmin	1074	71	16	21999
			HDD	114	59	1	2863
			HDD*	182	59	2	3485

Table 4.1: Experimental results. All tests were performed against GCC version 2.95.2.

could have minimized such characteristics, but doing so is unlikely to help the developer. We now discuss the result for each test case:

bug.c: The first program, bug.c, is the same example we demonstrated in Figure 4.1. HDD ran almost an order of magnitude fewer tests than ddmin—a common result in our evaluation. HDD's output size was nearly identical to that of ddmin, though HDD's is slightly smaller. The output from HDD is shown in Figure 4.2c. For comparison, we show ddmin's output below (formatted for easier reading):

```
t(double z[], int) {
  int i;
  int j;
  for(;;j++) {
    i=i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
}
```

In this example, the main distinction was the removal of the loop terminator, "`j++,`" something ddmin failed to do. In ddmin's case, there are two possible ways to remove the expression "`j++`" from the loop without modifying the loop itself. One possibility is for the post-increment operator to fall on the correct boundary when ddmin operates at the granularity of two characters. The other is for the entire expression to fall

on a correct boundary at the granularity of three or more characters, assuming the introduction of whitespace. Indeed, ddmin's output is very sensitive to whitespace. Although only slightly better, HDD's output could have been shortened if our implementation had attempted to remove parameters from function signatures. A quick comparison with ddmin's output shows that the parameter is not necessary, though it failed to remove the parameter altogether:

```
t(double z[],int) { ...
```

It is interesting to note that ddmin was able to remove a parameter name from the function definition while still inducing the bug in GCC. This is because GCC version 2.95.2 leniently proceeds to the code generation and optimization phases despite noticing the error.

boom7.c: The program boom7.c is relatively small in size and contains many variable declarations with one deep expression that induces the bug. HDD ran more than an order of magnitude fewer tests than ddmin, and the output was also significantly smaller—about half the size of ddmin's output. Early versions of our implementation failed to minimize the program significantly, but as we added support for minimizing unary expressions properly, HDD was able to produce a small failure inducing statement. The output from ddmin is less informative; it is not clear which portions correspond to the original. The output is not even parseable at points, for example:

```
(long int)(signed short)(var0=9>(var1=var1=8))""""
```

Compiling ddmin's output under a newer version of GCC produces a syntax error.

HDD* minimized the file even further than HDD. On the first iteration, HDD dramatically reduced an expression crucial for the failure, leaving many variables without references in the program. Since the original variable references were at deeper levels than the variable declarations themselves, attempting to remove them in the first iteration caused a type-checking error that prevented the compiler from failing. The second iteration was then able to remove the irrelevant variables since they were no longer needed.

cache.c: Program cache.c is relatively large, mainly due to the high number of header files included by the preprocessor. This characteristic is less than ideal for our approach since the resulting AST is flat, with many function prototypes and a few relevant functions following. The ddmin algorithm was quite capable of scaling with respect to the program size since most of the input is not pertinent. Even so, HDD still ran more than an order of magnitude fewer tests with an output size just under that of ddmin's. The most significant contributor was HDD's ability to unravel a heavily nested and parenthetical expression. Had our implementation removed parameters from function signatures or semi-colons after case statements, it would have removed five additional tokens. Not surprisingly, HDD* was able to remove a single unreferenced variable, yielding a slightly smaller program than HDD.

cache-min.c: The synthetic example, cache-min.c, exhibits a characteristic that affirms our approach. The program is a nearly minimized version of cache.c with parentheses introduced in all sub-expressions. It may seem strange that a nearly minimized input requires nearly the same number of test cases for the ddmin algorithm, but a nearly minimized program actually induces poor behavior in ddmin. This is because the algorithm tries in vain to remove large portions of the program. Although HDD is also limited by this, HDD constrains the number of nodes examined at one time. If a tree is fairly well balanced and of significant depth, we scale quite well in comparison. The output from cache-min.c was slightly more minimized than cache.c because all references to the variable "`line`" were dropped and thus HDD was able to eliminate the declaration "`int line;`". This is the same declaration that previously allowed HDD* to minimize cache.c, beyond what HDD was able to do.

4.2 XML

XML seems intuitively to be an interesting application for HDD because, unlike C, it is general, very strict, and often deeply nested. These characteristics make XML an ideal candidate for HDD. In this section, we experimentally confirm this.

Because of XML's generality, a single implementation of HDD can be applied

File	size (lines)	report (id)	algorithm	tests (#)	size (lines)	time (sec)	space (KB)
ms-tour.xsl	433	248258	ddmin	failed	failed	failed	failed
			ddmin-line	1092	92	485	27234
			HDD	124	8	86	6307
			HDD*	167	8	116	6332
uiwrapper.xsl	66	207358	ddmin	5757	46	1735	389480
			ddmin-line	277	43	117	3977
			HDD	105	15	63	3936
			HDD*	143	15	85	4059

Table 4.2: Experimental results for the XML study.

uniformly in many application domains. The process of our experimentation is evidence of this: we first implemented the XML tree processing module for HDD and then searched for XML document types on which to experiment. Our implementation is very simple; it is written in less than 150 lines of Python code using the DOM API. Although it may not always produce documents that are valid with respect to a specific document type definition (DTD), the generated documents are always well-formed.

While compilers and web browsers may tolerate bad inputs, XML parsers do not. This strictness is beneficial, even when users create XML documents directly, because it forces users to conform before facing mysterious errors or vendor lock-in. Because XML parsers are more strict, ddmin is more likely to produce documents that fail to parse, and is thus less likely to succeed. Consider that all tags present in a document must also have their closing tags to be processed. Removing one tag without the other will not induce the failure, even if the tags are both irrelevant. Furthermore, tags themselves must have matched angle brackets. Any configuration that produces a boundary covering an odd number of angle brackets will be frivolous.

Extensible Stylesheet Language Transformation (XSLT) is an XML document type that is used to transform other XML documents into another form. XSLT has a notorious history of inducing bugs; as such it was a good starting point for our bug search. The documents in Table 4.2 are the first XML documents with which we were able to reproduce errors in Mozilla. All experiments were performed on an Intel Pentium 4 (Xeon) with 2GB of RAM running Linux 2.6.17. The Mozilla bugzilla database (`http://bugzilla.mozilla.`

`org/`) gives such an example. It contains a bug entry for ms-tour.xsl (id 248258), a complex transformation that attempts to generate a knights tour of a chess board. The Mozilla web browser crashes with a segmentation fault upon processing this document. The file is comprised of 16500 characters and 433 lines of XML. Table 4.2 summarizes our results. In particular, we performed the following tests on this file:

ddmin: The ddmin algorithm failed to minimize this program. At larger granularities, ddmin did not remove significant portions of the document aside from those inside large comments. At smaller granularities, ddmin's cache of tested configurations ended up exhausting memory on our evaluation system as well as another with 4GB of RAM. With the cache disabled, ddmin retests duplicate configurations. Based on ddmin's diagnostic output, it was clear after one day of testing that we had not completed a significant percentage of total testing. We concluded that ddmin could not simplify this data under normal circumstances.

ddmin-line: To simplify the matter, we ended up minimizing the document on a line-by-line basis instead. Tags were generally isolated on individual lines. Accordingly, all possible boundaries respect angle bracket openings and closings, though they may not respect open and close tags. By treating lines or tags atomically, ddmin will not process tag attributes, but since we evaluate ddmin's performance by the number of lines it produces, this is not necessary.

As expected, ddmin had trouble significantly reducing the file. The output was 92 lines long and took 1092 tests to produce. On examination, the output seems unsatisfactory: it is still quite complex and includes many consecutive open and close tags that are irrelevant to inducing the failure. It may seem strange that ddmin was unable to remove them, but one must remember that the output is indeed 1-minimal since removing any one tag will not produce a parseable document. These tag pairs were not removed because they either contained tags that were not removed until later granularities, or they were on an odd-numbered line.

HDD and HDD*: Table 4.2 shows that HDD was more successful than ddmin in minimizing this input file. The output is only 8 lines and took 129 tests to produce. We

observe again that HDD takes an order of magnitude fewer tests, and manages to simplify the failure-inducing configuration to a concise one. The difference in output size for ddmin and HDD is very significant, with a factor of ten. Repeated application of HDD did not yield a smaller configuration. Still, HDD* did not incur a significant cost in terms of number of additional test cases.

We found another file that is small enough for ddmin to process at the character level. The XSLT file uiwrapperauto.xsl proved small enough for a direct comparison. Since ddmin was able to simplify the file successfully, we separated the tags from ddmin's output onto separate lines, to facilitate comparison. Our experiment showed that ddmin executed a factor of 30 more tests than ddmin-line, and nearly equivalent output. It was, however, able to shorten a few of the actual tag attributes. HDD also attempted to exclude attributes, though ddmin-line did not. Because this file is smaller, the overall results of this experiment are less drastic than ms-tour.xsl.

Almost paradoxically, we are unable to say HDD produces 1-minimal inputs, but HDD in our experience seems to be more effective than ddmin. The reason for HDD's success is quite simple: HDD can intelligently handle input languages which are context-free, while ddmin is suited for regular languages.

Chapter 5

Ensuring Syntax Validity of Test Cases

In Chapter 4, we described how some nodes of the input tree must be replaced with correction strings upon removal and demonstrated how conditional expressions could be replaced with "0". This *ad hoc* replacement is tedious for developers wishing to adapt HDD to their input domain. In this chapter, we describe a general framework to generate tree simplifying tools for context-free languages such that only syntactically valid variations of the input are produced. The purpose of providing this framework is to make correct simplifications on input trees automatically. The framework utilizes an algorithm that takes a language grammar as input and determines a minimal-length string for each nonterminal in the grammar. With this information, a parse tree simplifying tool is generated to operate on parse trees corresponding to the language grammar such that any removed node is replaced with a subtree representing a string of minimal length that preserves the syntactic validity of the entire tree. We implemented this algorithm by utilizing the YACC parser generator to facilitate parse tree construction.

5.1 Minimal-Length Strings

We now address the problem of finding a minimal-length string for each nonterminal in a language grammar. Recall that a context-free grammar is formally defined as

$G = (V, T, P, S)$, where V is the set of nonterminals; T is the set of terminals; P is the set of productions; and S is the starting nonterminal. The language of G is the set of strings that can be derived from the starting nonterminal S. Notice that like the starting nonterminal, all other nonterminals can also derive a set of strings. The algorithmic task is to determine a minimal-length string from this set of derivable strings for each nonterminal in the grammar.

We define a partial order, $\sqsubseteq$, on strings to describe precisely our notion of minimal-length strings. The empty string, denoted as ϵ, satisfies that $\forall s \in T^* \cdot \epsilon \sqsubseteq s$. The top string, denoted as $\top$, satisfies that $\forall s \in T^* \cdot s \sqsubseteq \top$. Let $s_1, s_2 \in T^*$, then $s_1 \sqsubseteq s_2$ if and only if $|s_1| < |s_2|$. A minimal-length string m for nonterminal n must satisfy that $\forall s \in L_n \cdot m \sqsubseteq s$.

Our problem exhibits an optimal substructure, indicating the potential for a dynamic programming algorithm. The optimal substructure allows that a minimal-length string for some nonterminal can be used to find a minimal-length string in productions referencing that nonterminal. More precisely, consider a minimal-length string s of some nonterminal n. A minimal-length string for any production containing n can have s in place of the string derived by n. If this were not the case, then some string of shorter length than s would be derived from n, which would contradict that s is a minimal-length string for n.

In order to find a dynamic algorithm we need to determine an order for evaluating nonterminals. If there is immediate recursion in productions, *i.e.*, a production for a nonterminal that is referenced within its rule, we could descend into an infinite recursion. It is apparent though, that evaluating this immediate recursive production can either concatenate to the string or leave the string as is. Such productions will thus never occur in the smallest derivation tree for a minimal-length string. Consider instead that there can be more indirect recursions, where two or more nonterminals form a cycle. The minimal-length strings of some of the nonterminals within the cycle may require a production that references some other nonterminal in the cycle. It is also the case that none may require any of the others. If all of the nonterminals derive at least one finite length string, it cannot be the case that all of the nonterminals within this cycle make use of productions that reference some other nonterminal within the cycle. But even with this restriction, we still cannot determine which minimal strings should be calculated first.

The key to solving this problem is instead to iteratively consider solutions whose derivation trees have a bounded height. Terminals, with height 0, have a minimal-length string containing only themselves. Nonterminals must always use productions, and so their derivation tree must have a height of at least 1. Nonterminals with a minimal-length string that uses a production referencing some other nonterminal must have a derivation tree height of at least 2. With this inductive reasoning in mind, we define a recursive formula, Φ, to compute minimal-length strings of nonterminals and terminals as

$$\begin{aligned}
\Phi_0[e] &= \begin{cases} e & \text{if } e \in T \\ \top & \text{if } e \in V \end{cases} \\
\Phi_i[e] &= \mathsf{min}\left(\{\Phi_{i-1}[e]\} \bigcup \left\{s | (e,r) \in P, s = \Phi_{i-1}[r_1]\Phi_{i-1}[r_2]\ldots\Phi_{i-1}[r_{|r|}]\right\}\right).
\end{aligned}$$

We define the result of string concatenation with $\top$ to be $\top$ and min to be with respect to our partial order $\sqsubseteq$.

The question that remains, however, is the number of iterations required by this formula before guaranteeing that minimal-length strings have been found for all nonterminals that have them. One solution to this problem is to wait until a fixed-point is reached, *i.e.*, $\Phi_i = \Phi_{i-1}$. In such a condition, no additional production can be applied to shorten the length of one of the minimal-length strings, and so there can be no reduction by iterating further. Though this condition is sufficient for termination and also that it is demonstrable that termination is guaranteed, for the sake of worst-case analysis, we must produce an upper bound on the number of iterations required. We now show that the upper bound of the derivation tree height of a minimal length string for any nonterminal is $|V|$.

Theorem 5.1.1. *Let s be a minimal-length string derivable from nonterminal n. There exists a derivation tree for s that has height less than or equal to $|V|$.*

Proof. Suppose that a minimal-length string s for nonterminal n requires a derivation tree of height greater than $|V|$. It follows that there is some path to a leaf of the derivation tree with length greater than $|V|$. Since each non-leaf node of the path is the application of one production, it follows that there must be more than $|V|$ derivations along the path. Each derivation makes use of a production for some nonterminal, and so there must be

some nonterminal p which is used twice along this path (by the pigeon hole principle). Replacing the higher p derivation subtree with the lower p derivation subtree produces a valid derivation tree for nonterminal n. If the new derivation tree yields a smaller string than s, then a contradiction of the hypothesis arises. If the new derivation tree does not yield a smaller string, we can apply the same reduction to all paths greater than $|V|$ until we have a derivation tree that has height less than or equal to $|V|$. This also is a contradiction of the hypothesis, and so there must always exist some derivation tree of height no greater than $|V|$ for all minimum-length strings of all nonterminals. □

Theorem 5.1.1 tells us that all minimal-length strings will be found in $|V|$ iterations. As a corollary of this, it is easy to see that all nonterminals for which we find no minimal-length string in $|V|$ iterations derives no strings. Since only $|V|$ iterations are required, our algorithm runs worst-case in $O(|V||P|)$ time; in essence it is a worst-case quadratic algorithm on the size of the language grammar.

In many cases, a minimal-length string is not unique. In our formulation of the algorithm, we have left min to select one string when there may be many of the same length. This raises the potential to apply heuristics to select among them. In our application, we believe this is not important, as the minimal-length strings are used in portions of the parse tree that are irrelevant to inducing a failure. Although it is possible that a particular minimal-length string may happen to facilitate simplification, we expect this to be rare enough that it would not merit a heuristic selection algorithm or the attempt of multiple minimal-length strings.

5.2 Parse Tree Simplification

We now present the general framework to produce tree simplifying tools suitable for use by HDD. Given only a context-free grammar, the framework generates a parser, a parse tree builder, minimal-length strings for nonterminals (using the aforementioned algorithm), and tree simplifying routines that respond to requests from the HDD algorithm.

Figure 5.1 diagrams the components of this framework more precisely. First the language grammar is sent to an annotator that inserts tree construction code to be used by

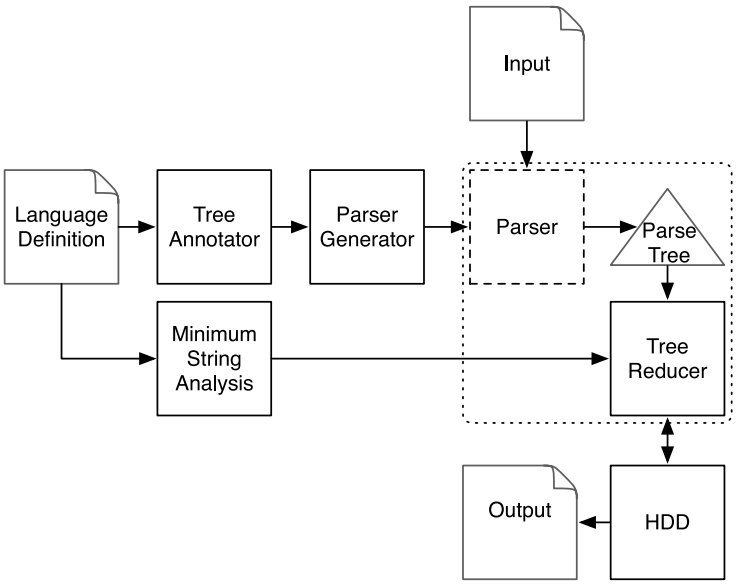


Figure 5.1: A language simplification framework.

the parser generator. The language grammar is also input to the minimum string analysis algorithm. The parser generator produces a parser which operates on input conforming to the language grammar. The parse trees are then simplified by the Hierarchical Delta Debugging algorithm.

Due to the availability of parser generators, we decided to use existing tools. We choose YACC despite its limitation to LALR grammars because of its ubiquity and the availability of many language grammars. Other parser generators are capable of parsing arbitrary context-free grammars, such as Elkhound [18].

The tree construction annotator must generate actions to build parse tree nodes as productions are parsed. The node should contain information indicating the nonterminal or terminal it represents, as well as the children that should be placed beneath it. As an example, the production `E := E + E` is annotated with the YACC code fragment:

```
{
  $$= Node("E");
  addChild($$,$1);
  addChild($$,$2);
  addChild($$,$3);
}
```

The parse trees are constructed in a general representation so that a single tree simplifying implementation can be applied to parse trees for all languages. Since HDD

```
E := E * E
E := E / E
E := E + E
E := E - E
E := ( E )
E := N
```

Figure 5.2: The grammar for our simple arithmetic language.

requires node labels, temporary removal, and permanent removal, nodes are augmented with this information. Removal need not alter the tree structure, and may instead change node states which are consulted during unparsing. During unparsing, when a removed node is encountered, its minimal-length string is produced instead of traversing its subtree to generate the final string. This general solution is far simpler than the hand-written C AST simplifying implementation developed earlier. The logic for removal and replacement is kept on the nodes in question rather than their parents, while previously, nodes had to check for the presence of children and produce specific correction strings in their place.

5.3 An Example Arithmetic Language

We now demonstrate how this framework would operate on a simple language for infix arithmetic. Let us assume the language is defined as in Figure 5.2.

Our grammar is excessively simple, and does not handle standard precedence of the arithmetic operators. Still, let us suppose that we have implemented a calculator for this language, and that a user has discovered an expression, `((1+(2*3))/(2-2))+(3*5)`, inducing the calculator to failure. As the developer of the calculator, we wish to find out why the expression fails, so we may chose to employ Hierarchical Delta Debugging and our general framework to facilitate this. To do so, we would simply provide the grammar to our framework to generate a tool for simplifying inputs for our language. Following that we would provide the failing expression as input, as well as a test harness to determine when simplified expressions exhibit the same symptom. We show the parse tree for the failing expression in Figure 5.3.

The minimal-length string for `E`, the one nonterminal of the language grammar, is `N`. We have not yet described how to create strings from terminals. One solution is to

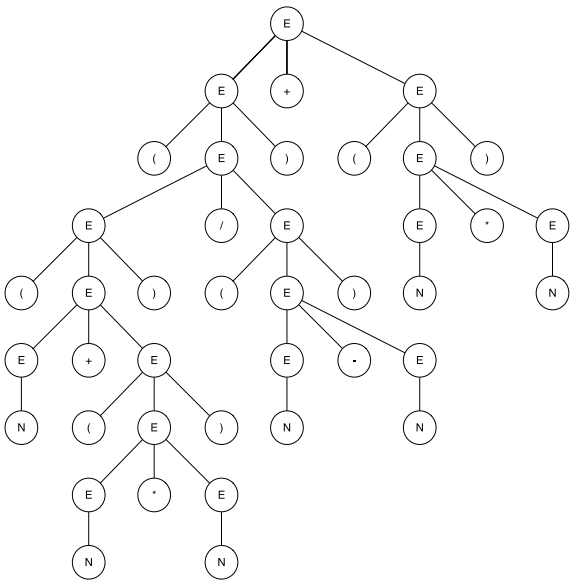


Figure 5.3: The parse tree for our example expression.

apply replacement rules to simplify regular expressions. Quantified repetition expressions, such as in `e*`, `e+`, and `e?`, can be replaced with the minimum number of occurrences of the subexpression. Character sets, such as in `[a-z]`, can be replaced with an arbitrary character in the set. Disjunction, such as in `e|f`, can be replaced with the minimum string from the reduced form of both sides. Alternatively, one could represent the regular language as a context-free grammar, and use the algorithm described in Section 5.1.

Returning to our example, let us assume that the minimal input produced is as shown in Figure 5.4, ultimately yielding a final expression of `(1 / (2 - 2)) + 1`. This resulting expression is simpler, though it is not *global-tree-minimal*. Suppose, for example, that the failure was induced by trying to evaluate a division by zero. There would be smaller *global-tree-minimal* expressions, such as `1 / (2 - 2)`. Though not *global-tree-minimal*, the output produced by our framework is particularly useful, because it determined the quotient was irrelevant early on. Without our syntax replacements, removing the quotient would have yielded a syntax error, thus requiring its presence until more fine grained simplification on it proceeds. Still, there is room for improvement on HDD with respect to parse trees. We see in this example that the path to the root of the tree from the relevant portions of the expression included an irrelevant operation, "`+ 1`". For future work, we plan to solve this

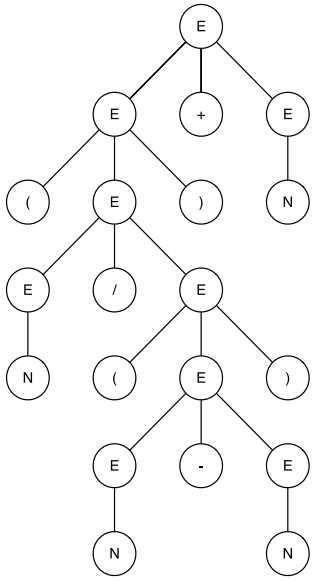


Figure 5.4: The simplified parse tree for our example expression.

File	**size** (chars)	**algorithm**	**tests** (#)	**size** (chars)	**time** (sec)	**space** (KB)
SHA1.java	433	ddmin	14817	652	10330	5373759
		HDD	88	41	49	60715

Table 5.1: Experimental results for the Java study.

problem by replacing nonterminals with ancestors of the same kind.

5.4 Empirical Results

In this section, we will describe a simple evaluation to confirm the correctness of our syntax validity algorithm as well as to measure its performance in a syntactic context. To do this we produced a concrete tool implementing HDD and the minimal string analysis algorithms described in this thesis. The tools was used to generate an automatic Java simplifier. We used an existing Java grammar that was included in GCC version 3.4.

We performed our evaluation on a Java implementation of SHA1 [2], a one-way hashing algorithm used in cryptography (though it is now deprecated). In this source file, we modified an expression with many arithmetic operations so that it contained a division by zero. We then setup a testing harness that would invoke the Java compiler on test inputs

```
/**/public final class S
extends e{//
{e
b[]=((5))+W[5];B((B));D=((E))+((D));C((C)(C));E=((5))+(C)[5];B=((5))+(B)[6]
;A((A)(E));B=((5)|(5))+f()+W[8];D=((D))+f;A=((5))+(E)[3];D((0)|(0));A=((5))
+(D)[4];C=((0)|(5))+f()+((5)|(5))+(D)[4];C((C));E=((A))+f();B((B)(B));D=((5
)|(5))+f()+W[6];A=((5))+((0));A=((5))+(D)[9];C=((C))+(C)[0];B((B));C=((D))+
f()+W[7];E((E)(E));B=((5))+f()+W[8];D=((D)(D))+f()+W[9];C=((C)(5))+(C)[0];B
((B));C=((5))+(A)[7];E=((5))+(A)[8];D=((5))+(E)[9];C((C)(C));E=((5))+f()+W[
0];B=((5))+(C)[1];A((A)(A));C=((5))+f()+W[2];E((0));E=((5))+(D)((0));D=((E)
)+(A)[6];A((0));B=((C))[8];D((0));A=((5))+(C)((0))/0;t[0]=t[1]=(D);}g
t="";/**/t()throws t{{}}}
```

Figure 5.5: The final output of Delta Debugging on SHA1.java.

```
final  class  a    {  ;  a a ( )  {  ;  a  =  ( 0  )  /  0  ;  ;  ;  ;  ;
;  ;  }  ;  ;  ;  ;  ;  ;  }
```

Figure 5.6: The final output of the Hierarchical Delta Debugging on SHA1.java.

to check for syntax errors and scan the test inputs for the division by zero to determine if the "failure" would still be induced. For this evaluation, we ignore semantic errors.

The results of our evaluation is shown in Table 5.1. All experiments were run on an Intel Core 2 Duo (Xeon) with 8GB of RAM running Linux kernel 2.6.20. The original source file has 13269 characters (excluding whitespace). The final outputs from both ddmin and HDD are shown in Figures 5.5 and 5.6 respectively. HDD, including the syntax preserving algorithm, minimized the source file to 41 non-whitespace characters in 88 tests. As expected, none of the HDD test cases had syntax errors. ddmin required 14817 tests, of which 13371 had syntax errors. ddmin required more than 5 GB of RAM, due to the large number of unique combinations attempted. The final output size for ddmin was 652 non-whitespace characters. As we experienced previously, ddmin had the most difficulty simplifying deep arithmetic expressions with many parenthesis. HDD, performed well, though there are spurious empty statements and declarations within the output. The reason for this is as mentioned in the previous section: the context from the relevant nodes to the root of the tree always remains. Since the language grammar uses recursion to represent lists, the parse tree contain these deeply nested nodes that become entrenched in the final output.

Chapter 6

Related Work

The most relevant work to our own is Zeller's seminal work on Delta Debugging [25,26]. Zeller hints at intelligent partition choices in [25]. The idea would significantly reduce the number of test cases run by the algorithm, though it falls short of realizing our motivational insight. Partition boundaries should be chosen with respect to a specific granularity of the input. It follows intuitively that we should start with the coarsest granularity and move towards the finest. Zeller and Hildebrandt briefly mention applying Delta Debugging to specific domains including a short reference to context-free grammars [26]. Sterling and Olsson's recent work on "program chipping" [19] is also related, and addresses a similar problem for Java. It uses simple tree manipulation techniques to produce test cases, while ours is based directly on Delta Debugging. The relative strength of both approaches is yet to be explored.

Rather than examining the input to a program, program slicing [20, 22] can be used to isolate relevant portions of a program that are necessary to yield some result. Program slicing can be performed either statically or dynamically (with respect to one concrete run) [4]. These techniques ease debugging by removing irrelevant portions of the failing program [3, 23]. It is worth noting that both Delta Debugging and program slicing can be used cooperatively. By first minimizing the input, we may significantly simplify the program slice and trace. These two techniques are not necessarily competitors; they can be complementary.

PSE [17] is a static analysis technique for diagnosing program failures. It can be viewed as a program slicing technique, however it is more precise because of its consideration of error conditions. A motivating example is dereferencing a NULL value. It is similar to Das's earlier work, ESP, a symbolic dataflow analysis engine [10].

Bug isolation is related to simplifying failure-inducing input. Rather than focusing on minimizing the input, it focuses on finding the cause underlying the failure. Delta Debugging the program state space is used as the mechanism for this technique [8,25]. The algorithm attempts to locate the state differences between passing and failing runs. This determines the relevant variables and values that infect the program to failure. A similar technique is applied to multi-threaded applications [7]. Instead of focusing on state, the technique examines the thread schedule differences of passing and failing runs.

Whalley's work on isolating failure-inducing optimizations is also related to our work [24]. His approach automatically isolates errors in the *vpo* compiler system. The search is performed on the sequence of optimizations performed by *vpo* to find the *first improving* optimization that causes incorrect output. The approach is fairly domain specific.

Liblit *et al.* use a sampling technique to reduce the runtime overhead of collecting successful and failing runs [14]. They also propose to use statistical learning techniques to infer the failures from many sampled runs [14, 15].

Work in error explanation for static analysis relates to our approach. Many tools produce error traces when a program violates its specification. However, understanding the error and locating the cause is usually left to the user. Several techniques have been developed to address this problem. Ball *et al.* suggest an approach to localize error causes [5]. Their idea is to find transitions in the error trace that do not appear in correct traces. Groce and Visser suggest a different approach for the same problem [12]. Given an error trace, they compute other error traces leading to the same assertion violation to compare with traces preserving the assertion.

Chapter 7

Implementation of our HDD Tool

In this chapter, we document our implementation of HDD. We have implemented HDD in the Python programming language. Users can apply HDD to minimize failure-inducing inputs by providing an object behaving as a test harness as well as a reducible tree.

An `HDDTest` object is responsible for invoking the inspected program and examining the return code and output from its execution. This test harness determines if a given test case continues to manifest the same symptom. The object must conform to the same interface as the `HDDTest` class. A user may subclass `HDDTest` for convenience, though this is not necessary in a duck-typed language such as Python. The object must have a method named `test`, taking one argument containing the string representing the test case. The method must return the result of the test case, which is one of: PASS, indicating that the failing symptom was not present; FAIL, indicating that the symptom was manifested.

Optionally, a user can use the HDDTest class directly by providing three arguments to its constructor: the file suffix for generated test cases (*e.g.*, `.java` for Java source files), the command to invoke the test (with the special format specifier `%s` indicating where the new test file should placed as an argument), and, optionally, a string that must be present in the program's output for failures. If the user does not provide a failure string, only the command's return code is used. For our Java example, the HDDTest object is constructed as `HDDTest('.java', './test.sh %s', 'DIVIDE BY ZERO')`.

The user must also provide a tree object that HDD will attempt to simplify. The tree should provide several methods:

`setLevels()`, a method to mark nodes with their level;

`unparse()`, a method to produce a string representing the current tree;

`tag(level)`, a method to tag all nodes on the given level with increasing identification numbers, starting with 0;

`setRemove(level,tokeep)`, a method that marks nodes as temporarily removed (and are thus ignored during unparsing) at the given level if they are not in the list `tokeep`.

`clearRemove()`, a method that removes the temporary removed state from all nodes in the tree;

`commitRemove(level,tokeep)`, a method to mark nodes at the given level and not in the list `tokeep` as permanently removed from the tree.

If the user wishes to exploit our automatic tree simplifying framework (with the aforementioned syntax validity algorithm), the user need only produce a language specification. The language specification is split into two sections separated by the characters `%%`: the context-free grammar, and the lexical definition. The context-free grammar conforms exactly to those of YACC. All actions in the YACC productions are removed automatically (and ultimately replaced with tree building actions by the tool). The lexical definition is a slight deviation from existing tools such as LEX. First the user specifies the token name, then a regular expression, and finally a minimal-length string recognized by the token. These strings could have been deduced automatically as described in Section 5.3, though we didn't find this as necessary as for context-free grammars, as most tokens contain only one string (such as keywords), and the others can be determined by the user. Figure 7.1 shows a portion of the language grammar for the Java programming language that was used in our evaluation. The context-free grammar was taken from GCC version 3.4. Notice that users can specify definitions of the `SKIP` token to describe sequences of characters that should not be considered by the lexer.

```
goal:  compilation_unit {} ;
...
compilation_unit:
            {$$ = NULL;}
|       package_declaration
|       import_declarations
|       type_declarations
...
class_declaration:
        modifiers CLASS_TK identifier super interfaces
                { create_class ($1, $3, $4, $5); }
        class_body
                {;}
|       CLASS_TK identifier super interfaces
                { create_class (0, $2, $3, $4); }
        class_body
                {;}
...
%%

SKIP: [|"/*"([^\*]|\*[^\/])*"*/"|];
SKIP: [|"//"[^\n]*\n|];
PUBLIC_TK : [|"public"|] [|public|];

PROTECTED_TK : [|"protected"|] [|protected|];
STATIC_TK : [|"static"|] [|static|];
FINAL_TK : [|"final"|] [|final|];
SYNCHRONIZED_TK : [|"synchronized"|] [|synchronized|];
VOLATILE_TK : [|"volatile"|] [|volatile|];
...
```

Figure 7.1: A selection from the Java language grammar used in our evaluation.

One produces language parse tree builders by running the language specification through the provided `HDDParserBuilder.py` tool. After processing the definition, our tool generates new files that are in turn used by both LEX and YACC. It also generates a Python source file, containing information about the language, such as the minimal-length strings for all nonterminals used in the grammar. The parse tree builder must be made available to HDD in order to simplify the trees. We use SWIG [1], a tool for facilitating the interoperability of multiple languages, to take the resulting native machine-code parser and make it available to our Python HDD implementation.

As an example of how to use HDD to simplify a failure-inducing input, we have

```
import j
from HDD import HDD,HDDTree,HDDTest

tree= j.parse( 'SHA1.java' )

tester= HDDTest( '.java', './test.sh %s', 'DIVIDE ZERO' )

HDD(tree,tester)

min= open('min.java','w')
min.write( tree.unparse() )
min.close()
```

Figure 7.2: The HDD driver used in our evaluation.

shown the Python program used in our Java evaluation in Figure 7.2. The HDD interface is simple. After running the program, the user can inspect `min.java` to see how HDD performed. The default `HDDTest` object dumps all test files into the directory `./test`. The user can examine the `./test` directory to see how many tests were run and what the individual test cases were. We have found it useful to examine these test cases for evaluating the strengths and pitfalls of input minimization tools.

As a reference, we have bundled our Java example with the HDD tool, so that users can refer to it when applying HDD to other languages.

Chapter 8

Conclusions

In this thesis, we have presented HDD, a Hierarchical Delta Debugging algorithm that exploits input structure to minimize failure-inducing inputs. We have evaluated the algorithm on some real-world examples. Our empirical evaluation confirms that Hierarchical Delta Debugging can reduce the number of generated tests and at the same time produce smaller output than the original Delta Debugging algorithm. Although some simplicity is lost, if projects have many non-trivial bug reports, the required work to implement the necessary tree simplifier is worthwhile. In these scenarios, input minimization without structural knowledge of the data would not scale. With our general framework for automatically generating parse tree builders and simplifiers, the amount of labor to adopt HDD is negligible provided that a context-free grammar for the input language is available. We leave two candidate simplification algorithms as future work: one that constrains Delta Debugging to branches of the input tree as opposed to our current level-wide approach; and another that attempts to promote nonterminals to higher ancestors of the same kind.

Bibliography

[1] SWIG: Simplified Wrapper and Interface Generator. `http://www.swig.org/`.

[2] The Cryptix Foundation. `http://www.cryptix.org/`.

[3] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993.

[4] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[5] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–105, 2003.

[6] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pages 73–82, 2004.

[7] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 210–220, 2002.

[8] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, pages 342–351, 2005.

[9] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for Java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 528–550, 2005.

[10] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[11] Premkumar T. Devanbu. GENOA - a customizable, front-end-retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology*, 8(2):177–212, 1999.

[12] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[13] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, NY, 1972.

[14] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 141–154, 2003.

[15] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 15–26, 2005.

[16] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[17] Roman Manevich, Manu Sridharan, and Stephen Adams. PSE: Explaining program failures via postmortem static analysis. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pages 63–72, 2004.

[18] Scott McPeak. Elkhound: A GLR parser generator and Elsa: The Elkhound-based C/C++ parser. `http://www.cs.berkeley.edu/~smcpeak/elkhound/`.

[19] Chad Sterling and Ronald A. Olsson. Automated bug isolation via program chipping. In *Proceedings of the Sixth International Symposium on Automated Debugging and Analysis-Driven Debugging*, pages 23–32, 2005. extended version to appear in SP&E 2007.

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

[21] Daniel Weise and Roger F. Crew. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 156–165, 1993.

[22] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[23] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[24] David B. Whalley. Automatic isolation of compiler errors. *ACM Transactions on Programming Languages and Systems*, 16(5):1648–1659, 1994.

[25] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1–10, 2002.

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