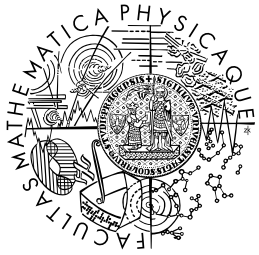

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Denis Leskovar

Automated Program Minimization With Preserving of Runtime Errors

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. RNDr. Pavel Parízek, Ph.D.
Study programme: Computer Science
Study branch: System Programming

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Dedication.

Title: Automated Program Minimization With Preserving of Runtime Errors

Author: Denis Leskovar

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Debugging of large programs is a difficult and time consuming task. Given a runtime error, the developer must first reproduce it. He then has to find the cause of the error and create a bugfix. This process can be made significantly more efficient by reducing the amount of code the developer has to look into. This paper introduces three different methodologies of automatically reducing a given program $\mathcal{P}$ into its minimal runnable subset $\mathcal{P}'$. The automatically generated program $\mathcal{P}'$ also has to result in the same runtime error as $\mathcal{P}$. The main focus of the reduction is on correctness when operating in a concrete application domain set by this study. Implementations of introduced methodologies written using the LLVM compiler infrastructure are then compared and classified. Performance is measured based on the statement count of the newly generated program and the speed at which the minimal variant was generated. Moreover, the limits of the three different approaches are investigated with respect to the general application domain. The paper concludes with an overview of the most general and efficient methodology.

Keywords: automated debugging, code analysis, syntax tree, statement reduction, clang

Contents

1. Introduction

Automation of routine tasks tied with software development has resulted in a tremendous increase in the productivity of software engineers. However, the task of debugging a program remains a mostly manual chore. Little progress is made due to the difficulty of reliably encountering logic-based runtime errors in the code, a task that, to this day, requires the developer's attention and supervision.

In this project, we attempt to tackle a specific issue concerning runtime error debugging. The problem can be summarized as program minimization with respect to a given runtime error. The following is the problem's definition.

Let program $\mathcal{P}$ contain a runtime error E that consistently occurs when $\mathcal{P}$ is run with arguments A. To find the cause of an error systematically, one might try removing unnecessary statements in the code, thus reducing the program's size. Let $\mathcal{P}'$ be a minimal variant of $\mathcal{P}$ such that $\mathcal{P}'$ results in the same error E as $\mathcal{P}$ when run with the same arguments. The program $\mathcal{P}'$ represents the smallest subset of $\mathcal{P}$ regarding code size while preserving the cause of the error in that subset. The task of program minimization finds $\mathcal{P}'$ for a given source code of $\mathcal{P}$.

Solving this problem leads to developers having to make less of an effort during their debugging sessions. Since they would be working with the smallest possible version of their source code, they would presumably spend less time finding the root cause in $\mathcal{P}'$ as opposed to $\mathcal{P}$. With this motivation, we attempt to suggest methods that perform program minimization automatically for code written in C and C++.

1.1 Goals

Having described the problem at hand, we designated our efforts into the following goals.

1. Research the existing techniques for automated debugging and source code size reduction.
2. Propose multiple approaches to solving the program-minimizing problem based on the previous findings. These proposals should include approaches that are accurate and practical.
3. Implement each approach for a concrete domain of input programs.
4. Compare the approaches by running a balanced set of benchmarks.

The motivation behind each suggested algorithm must be thoroughly explained. Moreover, all implementations should work on a specific domain of inputs.

1.2 Outline

The paper starts by describing existing techniques relevant to this project. Chapter 2 explains three debugging techniques used throughout the project. Having understood the essential ideas of automated debugging methods, we analyze frameworks and libraries to implement these ideas later.

In Chapter 3, we compare several compiler infrastructures and language recognition tools. The benefits and limitations of each framework are analyzed, and the best candidate is picked. Chapter 4 describes the candidate - Clang's LibTooling - in much more detail. The reader is introduced to the capabilities of the library. Constructs relevant to the implementation are highlighted and described as well.

Theoretical solutions to the program minimization problem are proposed in Chapter 5. A total of three approaches is presented. The section explains the motivation for each solution and well as its pseudocode. Chapter 6 weights the findings of the previous section and describes the implementation of the suggested algorithms.

The approaches are benchmarked in Chapter 7, and a comparison using several metrics is conducted. The paper concludes with a summary of the previous comparison's results.

2. Automated debugging techniques

TODO: Link relevant literature from Slicing of LLVM bitcode (muni.cz) and Bobox Runtime Optimization (cuni.cz)

Debugging can be described as the process of analyzing erroneous code to find the cause of those errors. Errors can also be of different natures. It can for example stem from poor design of the application. If that is not the case, then perhaps it comes from a rarely encountered input or a corner-case. The flaw might also be present in external code such as libraries or inappropriate usage of existing technologies.

It can be said with confidence that debugging is rarely an algorithmic approach. While the goal is clear, the process of debugging depends entirely on the programmer. It is typical that developers try to look for a root cause of an error by feeling what might be wrong. This works rather well in code the programmer is familiar with. However, in larger projects the developer did not create by himself, more sophisticated and reliable approaches are required. For example, one might add logging to the code being debugged, or perhaps create more tests that can narrow down the erroneous code.

All of the mentioned techniques require either the knowledge of the code or enough time to write supporting code. Additional time might be spent looking through the logs and executing tests. Therefore, it is rather hard to tell beforehand how much time and resources debugging will take.

While most developers see debugging as a manual chore, there were numerous attempts at automating at least some parts of it during the last few decades. The rise in popularity of program analysis resulted in the developement of automated error checkers for popular programming languages.

SpotBugs[1], formerly known as FindBugs, is a free and platform-independent application for, as the name suggests, finding bugs. It works with the bytecode of JDK8 and newer, which indicates that source code is not required. SpotBugs uses static analysis to discover bug patterns. These patterns are sequences of code that might contain bugs. They include misused language features, misused API methods, and changes to source code invariants created during code maintenance. Java developers can use SpotBugs's static analysis in its GUI form or as a plugin for build tools.

Clang static analyzer[2] provides similar functionality to C, C++, and Objective-C programmers. The code written in these languages is parsed by the analyzer. A collection of code analyzing techniques is then applied to it. This process results in an automatic bug finding, similar to compiler warnings. These warnings, however, include runtime bugs as well. The analyzer can uncover many bugs, from simple faulty array indexing to guarding the stack address scope. Due to its extensibility and integration in tools and IDEs alike, the Clang static analyzer is popular amongst developers working with the C family of languages.

The functionality of the previous tool was extended in CodeChecker[3]. Code-

[1]SpotBugs can be found at `https://spotbugs.github.io/index.html`.

[2]The Clang static analyzer's homepage is `https://clang-analyzer.llvm.org/`.

[3]CodeChecker's information page is `https://codechecker.readthedocs.io/en/latest/`.

Checker serves as a wrapper for the Clang static analyzer and Clang-Tidy. Wrapping these two tools into a more sophisticated application helps with user-friendliness tremendously. Additionally, the wrapper also deals with false positives. Furthermore, it allows the user to visualize the result as HTML or save time by analyzing only relevant files.

Facebook's Infer[4] translates both the C family of languages and Java into a common intermediate language. It also utilizes compilation information for additional accuracy. The intermediate code is then analyzed one function at a time. During the analysis, Infer can uncover tedious bugs such as invalid memory address access and thread-safety violation.

While the tools mentioned above mainly cover only specific cases of potential bugs, such as out-of-range array indexing, they have proven themselves valuable for the developer. In the context of this work, techniques behind such checkers provide a helping hand when minimizing a program. Moreover, they do so with state-of-art performance.

The following sections will talk about the techniques behind such checkers and how they deal with automated debugging. Notably, they describe the motivation and notation of Delta debugging and static and dynamic slicing.

2.1 Delta debugging

Delta debugging is an iterative approach described by Zeller[1]. It has two primary goals for a given program and the program's failure-inducing input. The first is to simplify the input by keeping only those parts that lead to the failure. The second is to isolate a part of the input that guarantees the failure.

The first goal is especially relevant in the context of this project and will be described in more detail in this section. The simplifying algorithm, also known as the minimizing algorithm, reduces the size of a failure-inducing input. For a given program, a test case, and an input for that test case, it simplifies the test case's input. It assumes that each execution of the program has the following results: pass, fail, inconclusive.

Zeller and Hildebrandt[2] have presented the following definitions to be more precise with the terminology.

Definition 1 (Test case)**.** *Let $c_{\mathcal{F}}$ be a set of all changes $\delta_1, \ldots, \delta_n$ between a passing program's input $r_{\mathcal{P}}$ and a failing program's input $r_{\mathcal{F}}$ such that*

$$r_{\mathcal{F}} = (\delta_1(\delta_2(\ldots(\delta_n(r_{\mathcal{P}}))))).$$

We call a subset $c \subseteq c_{\mathcal{F}}$ a test case*.*

To understand the definition, we must first assume two inputs for the debugged programs. Say we have an input $r_{\mathcal{P}}$ with which the program terminates successfully. Let us consider that the passing input is trivial, i.e., empty. Now consider an input $r_{\mathcal{F}}$ that leads to a failure when the program is executed. The difference between these two inputs is what $c_{\mathcal{F}}$ represents.

The difference in the definition is decomposed into several more minor differences. In simple terms, one can think about the difference between $r_{\mathcal{P}}$ and $r_{\mathcal{F}}$

[4]General overview of Infer can be found at `https://fbinfer.com/`.

as the string $r_{\mathcal{F}}$ (since $r_{\mathcal{P}}$ is trivial). The decomposed differences $\delta_1, \ldots, \delta_n$ represent substrings of the string $r_{\mathcal{F}}$. When composed and applied to $r_{\mathcal{P}}$, $\delta_1, \ldots, \delta_n$ transform $r_{\mathcal{P}}$ into $r_{\mathcal{F}}$. Subsets of $\delta_1, \ldots, \delta_n$ are called test cases.

The goal of the minimizing algorithm is to find the minimal test case. The minimal test case can be interpreted as the smallest set of the failure-inducing input that still fails.

Definition 2 (Global minimum)**.** *A test case $c \subset c_{\mathcal{F}}$ is called a* global minimum *of $c_{\mathcal{F}}$ if $\forall c_i \subseteq c_{\mathcal{F}} : (|c_i| < |c| \implies c_i$ does not cause the program to fail.)*

The global minimum is practically impossible to compute. Since we are looking for a subset with specific properties, we must test all subsets. This results in exponential running time complexity. Instead, we can find a local minimum.

Definition 3 (Local minimum)**.** *A test case $c \subset c_{\mathcal{F}}$ is called a* local minimum *of $c_{\mathcal{F}}$ if $\forall c_i \subseteq c : (c_i$ does not cause the program to fail.)*

The rule for a local minimum is that no test case's subset causes failure. Unlike the global minimum, the local minimum is not the smallest input variant. However, it still preserves an interesting property. All elements of the local minimum are significant to producing the failure. In other words, no element can be removed. Calculating a local minimum is also an exponentially complex operation. To be more efficient, we need to deploy approximations.

Definition 4 (n-minimality)**.** *A test case $c \subset c_{\mathcal{F}}$ is* n-minimal *if $\forall c_i \subseteq c : (|c| - |c_i| \leq n \implies c_i$ does not cause the program to fail.)*

This approximation dictates how throughout the element removal will be. The larger the n in n-minimality is, the smaller the output will be. Delta debugging is generally interested in 1-minimal test cases, i.e., removing any element results in passing a test case. Though, testing for 1-minimality might take more time than necessary. The minimizing algorithm utilizes binary search to reduce the number of its iterations.

The algorithm attempts to increase its chances of finding a failing subset by using a following modification. It tests the binary search's partitions as well as their complements. By testing small subsets (partitions split by the binary search), the algorithm reduces its chances of achieving a smaller failing test case. On the other hand, testing larger subsets (complements of those partitions) improves the chances of finding a failing test case. While testing larger subsets increases the chances of getting a result, it is considerably slower.

The simplified algorithm description seen in figure 2.1 splits the test case into n even-sized partitions and their respective complements. These partitions are tested first, followed by all complements. The testing can result in three different outcomes. If all tests pass correctly, the granularity, i.e., n, is doubled, and the test case is split into more even-sized partitions. On the other hand, if a partition fails a test, the granularity is reset to its initial value. Additionally, the partition now becomes the test case. If neither of the two mentioned scenarios happens, then a partition's complement must have failed to pass a test. This case results in the granularity being decreased, and the test case is set to the failure causing complement. These three steps repeat iteratively, updating the test case and splitting it systematically with different granularities. Once the granularity

Input: $\sigma \ldots$ the test's input string.
Output: The reduced failure-causing substring.

1: $n \leftarrow 2$
2: Split the string σ into $\alpha_1, \ldots, \alpha_n$ of equal size.
3: For each α_i, calculate its complement β_i.
4: Run tests on $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$.
5: **if** all tests passed **then**
6: $\quad n \leftarrow 2 * n$
7: $\quad$ **if** $n > |\sigma|$ **then return** the most recent failure-causing substring.
8: $\quad$ **else**
9: $\quad\quad$ goto (2).
10: $\quad$ **end if**
11: **else if** $\alpha_i failed$ **then**
12: $\quad n \leftarrow 2$.
13: $\quad \sigma \leftarrow \alpha_i$.
14: $\quad$ **if** $|\sigma| == 1$ **then return** σ.
15: $\quad$ **else**
16: $\quad\quad$ goto (2).
17: $\quad$ **end if**
18: **else**
$\quad \triangleright \beta_i$ failed.
19: $\quad \sigma \leftarrow \beta_i$.
20: $\quad n \leftarrow n - 1$.
21: $\quad$ goto (2).
22: **end if**

Figure 2.1: Minimizing Delta Debugging Algorithm.

is greater than the test cases's size, the most recent failure-inducing snippet is returned. The same case holds when the test case is of size 1, i.e., it cannot be further divided.

For the sake of this project, we can quickly transform test case minimization into source code minimization:

1. We consider the source code as the input of the algorithm.
2. We compile and execute that input in an appropriate execution environment.
3. We test each iteration on whether it contains the desired runtime error.

The details on the usage of Delta debugging are described in section 5.2.

2.2 Static slicing

Program slicing, formalized more than three decades ago, is a branch of program analysis that studies program semantics. It systematically observes and alters the program's control-flow and data-flow for a given statement and variable in the code. The goal of slicing is to create a slice of a program, i.e., a series of parts

of the program that could potentially impact the control and data flow at some given point in that program. The direction from which the target statement is approached divides slicing methods into two groups. Firstly, forward slicing uncovers parts of the code that might be affected by the targeted statement and variable. Secondly, and much more common, backward slicing computes parts of the program that impacts the targeted statement.

The first introduced slicing method was static backward slicing. And with it came brand new formalism concerning program analysis. Specifically for static slicing methods, definitions for the target 4 statement and variable needed to be written. Weiser [3] defined a slice with respect to criterion C as a part of a program that potentially affects given variables in a given point.

Definition 5 (Static slicing criterion)**.** *Let $\mathcal{P}$ be a program consisting of program points $P = p_1, \ldots, p_n$ and variables $V = v_1, \ldots, v_m$. Any pair $C = (p_i, V')$, such that $p_i \in P$, $V' \subseteq V$, and $\forall v_i \in V' : v_i$ is present in p_i, is called a* slicing criterion*.*

Slicing is the process of finding such a part of a program. Suggested approaches neglected any execution information and focused solely on observations made by analyzing the code.

One can imagine that the size of a static slice would be much smaller than the original program. That would be the case in modular code that rarely interacts between its components. An example of such code would be heavy parallel applications and computational tasks. However, in programs with aggressive use of branching, it is not so. Since static slicing considers statements that **might** impact the criterion, it leaves otherwise useless branches in the slice, thus negating the potential decrease in size.

In listing 2.1, we can see the code of a simple program. It loads a value a, which then alters the control-flow of the code. Meanwhile, it iterates through a printing loop. The intriguing part, however, is the output of the $write(x)$ command on line 42. Let the criterion be $C = (write(x)_{42}, \{x\})$. The value of x on that line is changed in the branching part of the program, which entirely depends on the value of a. Since a is unknown, no significant code reduction can be made. The static slice with respect to C, seen in listing 2.2, still contains all of the branching statements. Note that the independent printing loop is gone.

Later that year, K. J. Ottenstein and L. M. Ottenstein [4] restated the problem as a reachability search in the program dependence graph (PDG). PDG represents statements in the code as vertices and data and control dependencies as oriented edges. Additionally, edges induce a partial ordering on the vertices. In order to preserve the semantics of the program, statements must be executed according to this ordering.

Edges are, therefore, of two types. First, the control dependency edge specifies that an incoming vertex's execution depends on the outgoing one's execution. Second, the data flow dependence edge suggests that a variable appearing in both the outgoing and incoming edge share a variable, the value of which depends on the order of the vertices execution.

Once the PDG is built, slices can be extracted in linear time with respect to the number of vertices.

Figure 2.3 shows a PDG that was extracted using an AST Slicer. Nodes of the graph contain the same statements as seen in the code. Frameworks that

Listing 2.1: Simple branching program.

```
1  #include<iostream>
2
3  void write(int x)
4  {
5    std::cout << x << "\n";
6  }
7
8  int read()
9  {
10   int x;
11   std::cin >> x;
12
13   return x;
14 }
15
16 int main(void)
17 {
18   int x = 1;
19   int a = read();
20
21   for (int i = 0;
22   i < 0xffff; i++)
23   {
24     write(i);
25   }
26
27   if ((a % 2) == 0)
28   {
29     if (a != 0)
30     {
31       x *= -1;
32     }
33     else
34     {
35       x = 0;
36     }
37   }
38   else
39   {
40     x++;
41   }
42
43   write(x);
44
45   return 0;
46 }
```

Listing 2.2: Static slice of the simple branching program.

```
#include<iostream>                1
                                  2
void write(int x)                 3
{                                 4
  std::cout << x << "\n";         5
}                                 6
                                  7
int read()                        8
{                                 9
  int x;                          10
  std::cin >> x;                  11
                                  12
  return x;                       13
}                                 14
                                  15
int main(void)                    16
{                                 17
  int x = 1;                      18
  int a = read();                 19
                                  20
                                  21
                                  22
                                  23
                                  24
                                  25
                                  26
  if ((a % 2) == 0)               27
  {                               28
    if (a != 0)                   29
    {                             30
      x *= -1;                    31
    }                             32
    else                          33
    {                             34
      x = 0;                      35
    }                             36
  }                               37
  else                            38
  {                               39
    x++;                          40
  }                               41
                                  42
  write(x);                       43
                                  44
  return 0;                       45
}                                 46
```

Figure 2.2: An illustration of the difference static slicing makes. The source code on the left is the original program, the code on the right is its static slice w.r.t. $C = (write(x)_{42}, \{x\})$.

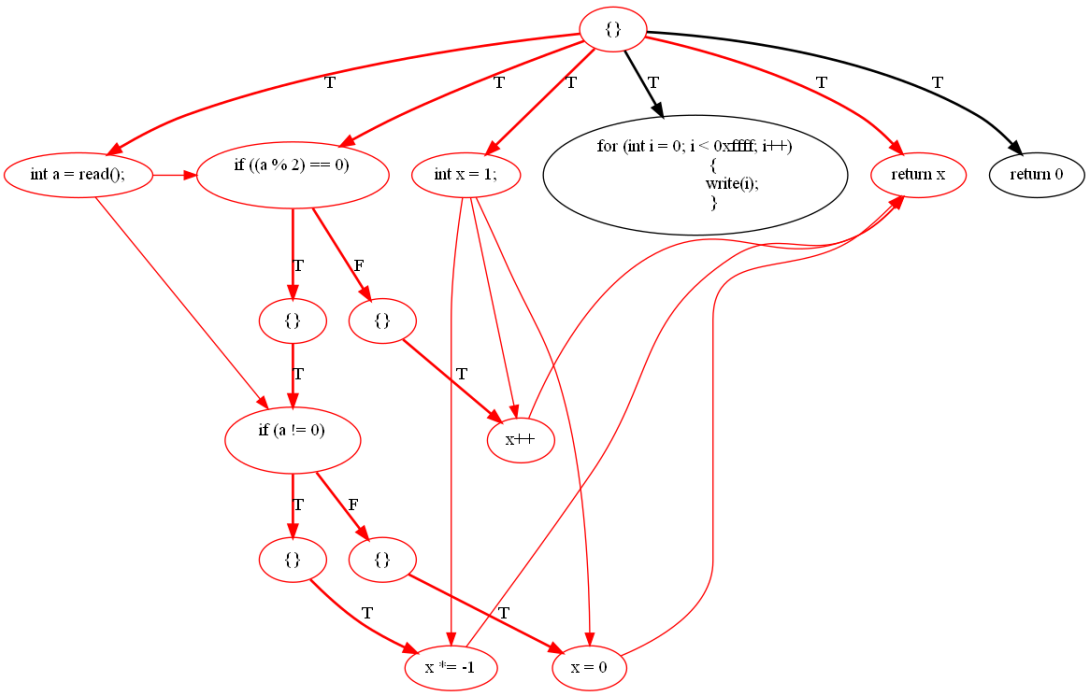

Figure 2.3: Sliced PDG. The graph was created from the source code shown in listing 2.1. Red edges indicate the sliced part of the program w.r.t. $C = (write(x)_{42}, \{x\})$.

achieve such mapping between the code and the internal control and data flows allow developers to create slicing tools much more easily. One such framework is the LLVM/Clang Tooling library, which will be talked about later. The tool is available at `https://github.com/dwat3r/slicer`.

However, one can find many potential issues and obstacles when performing data flow analysis. Omitting the interprocedural slicing, as it is not relevant in this projects's context, one is left with pointers and unstructured control flow. While the latter is rarely used in single-threaded modern programming, the same cannot be said about the former.

Pointers require us to extend the syntactic data flow analysis into a pointer or points-to analysis, which should be performed first. It is necessary to keep track of where pointers may point to (or must point to, in case their address is not reassigned) during the execution. From this knowledge, other data flow edges must be created or changed to accommodate the fact when the outgoing vertex mayhap writes into a memory location possibly used by the incoming vertex.

The analogical approach is then used for control dependency analysis since pointers might alter control flow as well. This change to control flow happens, namely when functions are called using function pointers.

The main advantage of static slicing is that it does not require any run-time information. As program execution can be expensive both time-wise and resource-wise, static slicing offers program comprehension at a low cost. Because static slicing discovers program statements that can affect certain variables, it can remove dead code and be used for program segmentation.

Furthermore, static slicing is used for testing software quality, maintenance,

and test, all of which are relevant to this project.

2.3 Dynamic slicing

While the idea of building a program slice prevails, dynamic slicing drastically differs from static slicing in terms of input and the way it is processed.

Korel [5] described a slicing approach that took into consideration information regarding a program's concrete execution. As opposed to static slicing, which builds a slice for any execution, dynamic slicing builds a slice for a given execution of a program. Using information available during a run of the program results in a typically much smaller slice.

```
1  #include<iostream>
2
3  void write(int x)
4  {
5    std::cout << x << "\n";
6  }
7
8  int read()
9  {
10   int x;
11   std::cin >> x;
12
13   return x;
14 }
15
16 int main(void)
17 {
18   int x = 1;
19   int a = read();
20
21   x = 0;
22
23   write(x);
24
25   return 0;
26 }
```

Figure 2.4: Dynamic slice of the simple branching program seen in listing 2.1 w.r.t. $C = (write(x)_{42}, \{x\}, \{2\})$.

This decrease in size is mainly due to removing unnecessary branching of control statements and unexecuted statements in general. The slicing criterion now contains a set of the program's arguments in addition to the previous information. The location of the criterion's statement is also specified to avoid vagueness in the execution history.

The criterion is therefore defined as follows.

Definition 6 (Dynamic slicing criterion)**.** *Let $\mathcal{H} = (s_{x1}, \ldots, s_{xn})$ be an execution history of a program $\mathcal{P} = (\{s_1, \ldots, s_m\}, V)$, where s_i denotes a statement and V is a set of variables $v_1, \ldots, v_k$. Any triple $C = (h_i, V', \{a_1, \ldots, a_j\})$, such that $h_i \in \mathcal{H}$, $V' \subseteq V$, $\forall v_i \in V' : v_i$ is present in h_i, and $\{a_1, \ldots, a_j\}$ is the input of the program, is called a* slicing criterion*.*

The example listing 2.4 was computed from the original listing 2.1. The criterion was set to $C = (write(x)_{42}, \{x\}, \{2\})$. Since the dynamic slicer witnessed the program's execution, it could precisely reduce the code to only those statements that were executed. the result is a significantly smaller slice than the static slice shown in listing 2.2. Note that branching statements are gone.

Since dynamic slicing requires the user to run the program, it is typically used in cases where the execution with a fixed input happens regardless. Such cases include debugging and testing. For debugging, dynamic slices must reflect the subsequent restriction: a program and its slices must follow the same execution paths.

2.4 Summary

While the described program minimizing and debugging approaches have been formulated more than two decades ago, there have not been nearly enough successful attempts at implementing them.

With each approach having its clear positives and negatives, it would be interesting to see how they handle program minimization. When cleverly used, a combination of these methods might result in a reasonably fast and inexpensive algorithm for the reduction of program size.

3. Compilers and analysis tools

In the previous chapter, the reader was introduced to a branch of program analysis. The techniques discussed above focused on both the static and runtime side of program analysis.

Regardless of whether these approaches have been implemented, it was required to find a suitable tool for source code manipulation for two reasons. First, any external tool output might require altering the input source code based on its output. Second, if implementing any code reducing algorithm would have to occur, one would need a sophisticated code modifying framework.

Due to these reasons, an analysis of compilers and tools for C and C++ was conducted. The goal of the analysis is to pick the most practical tool available. Required criteria include frequent upkeep of the framework, an existing user base, and the ability to manipulate some abstract representation of the code.

The representation boiled down to an abstract syntax tree (AST). AST embodies the syntactic structure of the code, regardless of the code's language. A vertex of an AST represents a construct of the code while not being concrete with the details of the code's programming language. This generality is perfect for C and C++'s chosen domain, as both languages only differ syntax-wise in minor details.

Below are the findings concerning the most important candidates.

3.1 GCC

A well-known C and C++ compiler, the GNU Compiler Collection [6] is an extensive open source project. As popular as GCC is, it does not provide the features an analysis-tool-building developer needs.

For the sake of building such tools, a compiler front end is used. Due to an old design, it is difficult to work with either the front end or the back end of GCC alone. Besides, the compiler implicitly makes optimizations that destroy any parallels between the source code and the AST. Therefore, the AST has to be treated as an entirely different object rather than an abstraction of the code. Most of the compiler's source code representation is unintuitive and hard to pick up for anyone not actively contributing to GCC. Figure 3.1 showcases the unfriendliness rather well. Compared to figure 2.3, which is an output of a tool built using LLVM and Clang, GCC's mapping between the source code and the internal representation does not hold up.

As far as AST manipulation is concerned, the compiler allows the user to dump the structure into a text representation. However, due to the difficulties mentioned above, it can hardly be used.

These issues result in a seldom-used variant that offers nearly no developer-friendly features. An upside is that GCC allows the user to visualize the AST. However, that is hardly a useful feature in the context of this project.

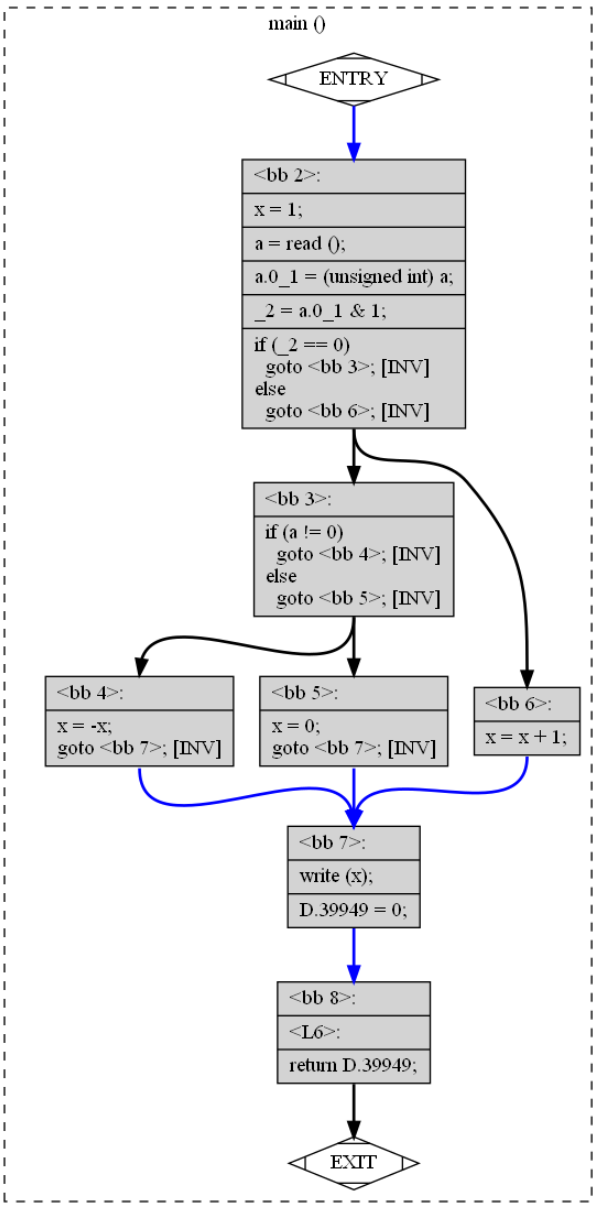

Figure 3.1: GCC AST Dump. This figure showcases the AST representation of listing 2.1 as dumped by GCC. Note that it is not easily comprehendable.

3.2 Clang

Thanks to LLVM [7], the widespread compiler infrastructure, the Clang project [8] has provided a compiler front end not only for C and C++ but also for CUDA, OpenCL, and other mainstream programming languages. The extend of Clang as a compiler front end is so vast that it covers both the C++ standard and the unofficial GNU++ dialect.

The project does not include just the front end but also a static analyzer and several code analysis tools, which are now commonly used in IDE's as syntax and semantic checks.

This description of Clang foreshadows its friendliness to analysis tool developers. The fact that the front end runs on a common intermediate language also indicates that openly working with abstract code representations is supported.

There are three most notable interfaces for customizing Clang. Firstly, the LibClang interface allows the users to write comprehend-able high-level code with limited functionality. On the other hand, LibTooling gives the user much more control at the cost of a steep learning curve. Lastly, the Plugins interface features similar difficulty as LibTooling with a more specific goal. Plugins are used with the Clang compiler and can be run as a front-end action when called during compilation.

3.3 ANTLR

A less typical way of extracting an AST from a source file is by using grammar recognition. ANTLR [9], which stands for Another Tool for Language Recognition, is a free parser generator that generates both a lexer and a parser based on a given grammar. Additionally, ANTLR can also generate a tree parser. Tree parsers are helpful in processing ASTs.

The tool is generally used to read data formats, process expressions of various query languages, and even parse source code written in complex programming languages. It can be used to generate a syntax tree and walk through it using a visitor. ANTLR is based on the LL parser, which parses the input from left to right, performing its leftmost derivation.

To create a parser or a syntax tree of code written in a programming language, ANTLR requires the complete grammar of that language. Some programming languages, namely C and C++, have an ambiguous syntax that is hard to parse based solely on its grammar. Due to ANTLR's high popularity, many grammars have already been written for it. As far as C++ is concerned, its C++14 standard's grammar is the most recent one available.

Writing grammar for newer standards or creating a custom one for both C and C++ would be unnecessarily burdensome for this project. This statement holds, especially when considering other tools mentioned above.

The most recent release, ANTLR 4, added more options for grammar rules. Most notably, it supports direct left recursion. However, that still might not be enough to choose it over other tools.

3.4 DMS

Similar to ANTLR, the DMS Software Reengineering Toolkit [10] features a parser generator. The tool is proprietary software created by Semantic Designs. Besides the mentioned parser generator, it features an entire toolkit for creating custom software analysis. This toolkit is used mainly for reliable refactoring, duplicate code detection, and migration of the source code's programming language.

The parser generator part takes a grammar and produces a parser. This parser then constructs abstract syntax trees for provided source code. Additionally, created ASTs can be converted back to source code using prettyprinters. The parser saves additional information about provided source files, such as comments and formatting. It can then recreate the file accurately.

DMS provides a grammar for a large number of programming languages, including C and C++. The language support, however, is not always up-to-date. The newest supported C++ standard is still the older C++17. These complicated grammars' ambiguity is avoided using a generalized left-to-right parser, which performs the rightmost derivation (GLR). Since DMS provides refactoring ability as well, it allows for transformation rules in the grammar.

Another helpful feature of the toolkit is control flow and data flow analysis. Analyzing control flow and data flow, generating their graphs, and performing the points-to analysis (also supported by DMS) is practical when considering static slicing (section 1.2).

It should be noted that some of the free, open-source tools mentioned above do a better job of being a so-called 'software analysis toolkit' than DMS does.

3.5 Summary

The chapter highlighted a spectrum of tools, ranging from language recognizers to compilers.

It would seem that parsing source code written in multiple programming languages into an abstract representation requires a common intermediate language, in which the representation is stored. Having an intermediate language is not always possible for several reasons, including licensing and old architecture. The compiler giant GCC seems to suffer from precisely that. Additionally, since the Clang project is being contributed to regularly, resulting in as many as five releases per year, it pulls in a more significant developer community.

Therefore, Clang is the favorite source code altering tool for this project. In the following chapter, the relevant parts of the Clang project will be broken down and explained.

4. Clang LibTooling

The previous chapter described tools and environments that were taken into consideration for this project. The utmost importance was given to the ease of use, availability, and active community. As the reader might have guessed from the summary, the LLVM/Clang suite stood out as the best candidate.

Clang is a language front-end. With high compilation performance, low memory footprint, and modifiable code base, it quickly and flexibly converts source code to LLVM intermediate code representation. The front-end supports languages and frameworks such as C/C++, Objective C/C++, CUDA, OpenCL, OpenMP, RenderScript, and HIP. This support is crucial for this thesis since the project aims to support both C and C++. The LLVM Core then handles the optimization and IR synthesis, supporting a plethora of popular CPUs.

Clang is widely used for its warnings and error checks, both very helpful and outstanding compared to competing compilers. Furthermore, Clang offers an extensive tooling infrastructure through which tools such as clang-tidy were developed. A relatively well-documented tooling API written in C++ helps programmers create their tools easily. However, not all developers share the same skill set. Some programmers require complicated additional features, while others prefer an easy-to-use interface. The tooling API has been split into multiple libraries and frameworks, including Plugins and LibClang. Explaining the two mentioned libraries is necessary. It is essential to show their capabilities before introducing LibTooling. LibTooling is the tooling library ultimately chosen for this project.

Plugins. The library intuitively called Plugins is used for plugin development. The library is linked dynamically, resulting in relatively small tools. Plugins are launched at compilation and offer compilation control as well as access to the AST.

More specifically, Plugins allow performing an extra custom front-end action during compilation. The functionality is generally similar to that of LibTooling, which will be talked about later. However, unlike a standalone tool, Plugins cannot do any tasks before and after the analysis (and compilation). When creating a plugin, one can choose from a selection of `FrontendAction` classes to inherit. If, for example, the plugin should work with the AST, the `ASTFrontendAction` can be inherited. Doing so also allows overriding the `ParseArgs` method, in which the plugin's command line handling is specified.

Due to dynamic loading, the wanted plugin must be added to a plugin registry inside the code. The plugin is then loaded from the registry by specifying the `-load` command or `-fplugin` on the command line when running clang. The plugin takes those arguments from the command line that are prefixed by `-Xclang`.

LibClang. Another framework, LibClang, offers a simple C and Python API for quick tool writing. Unlike Plugins and LibTooling, which will be mentioned later, the code base of LibClang is stable. This stability implies that tools written using LibClang do not require upkeep with every new LLVM/Clang release. Overall,

the framework and tools written using it are high-level and are easily readable.

LibTooling. The most feature woven set of libraries is LibTooling. Unlike Plugins, LibTooling [11] allows the developer to build standalone Clang tools. This robust framework is written in C++ and has an active community of contributors. One can find many manuals and tutorials online. However, with each contribution to LibTooling and each release of Clang, there is a chance that older tools will not support the newer LibTooling API. That is the reason why countless tools written using this framework do not run in modern environments. Programmers who use LibTooling cannot expect compatibility in upcoming releases. On the bright side, the libraries of LibTooling allow a plethora of source code modifications, AST traversals, and access to the compiler's internals.

The set of features supplied by LibTooling is immense. The following sections describe notable features used during the implementation of this project. The reader should get a better idea of how a tool is built and what LibTooling offers during the development process. Important concepts, such as providing the correct input to the tool in the form of a compilation database, traversing the AST, and modifying source code inside the tooling environment, are described below. These concepts will be referenced further in the text.

4.1 Compilation databases

To accurately and faithfully recreate a compilation, tools created using LibTooling require a compilation database (CD) [12] for a given input project.

The motivation behind a CD is simple. If a source file uses unusual include paths that need to be provided using the `-I` compiler command, it cannot be reliably compiled. Similarly, if the file contains macros and lacks definitions, its content can drastically change when the definitions are present. In the latter case, definitions are provided to the compiler with the `-D` command. Such compiler commands, options, and flags are usually defined in a build system. At least, that is the recommended practice for larger projects. Having a build system is similar to having a CD. It is clear which file is compiled with which options.

Clang expects a CD in the JSON format and looks for the file specifically named `compile_commands.json` in the current or parent directories. The JSON file contains entries for source files. Each entry contains a directory, a file name, and a compilation command. Multiple entries for a single source file are also valid. Such a case can arise when performing repeated compilation.

As previously mentioned, having a build system helps. Build tools such as CMake and Ninja can be used to generate a CD. If the project is not using any of the compatible build tools, the user can either make a CD manually or use an external tool. One such tool is Build EAR available at `https://github.com/rizsotto/Bear`.

Tools created using LibTooling do not always require compilation databases to run. For simple projects, they can take the `--` argument that separates the tool's arguments from the project's compilation arguments. One can interpret the arguments following `--` as a temporary compilation database.

4.2 Clang AST

The abstract syntax tree used in the Clang front-end [13] is different from the typical AST. It saves and carries more data, namely context. For example, it contains additional information to map source code to nodes and capture semantics. This chapter describes the Clang node type hierarchy, the tree's representation in memory, and different ways of traversing an AST.

4.2.1 Node types

Clang AST's nodes belong to a vast class hierarchy. This hierarchy contains classes that represent every supported source code construct. Nodes are of four different types: statements (`Stmt`), declarations (`Decl`), specific declaration context (`DeclContext`), and types (`Type`). However, in the APIs mentioned above, the nodes do not share a common ancestor.

The children of `Type` represent all available types. The goal is to give each type in the source code a canonical type, i.e., a type stripped of any typedef names. Canonical types are used for type comparison, while non-canonical types give complete information during diagnostics. The `Decl` hierarchy's goal is to have a class for each type of declaration or definition. These declarations vary, and the children cover specific cases such as function, structure, and enum declarations. Some declarations, such as function and namespace declarations, capture additional data in `DeclContext`'s children. The final node type, `Stmt`, represents a single statement. It has subclasses for loops, control statements, compound statements, and more. Additionally, expressions (`Expr`) also belong to the `Stmt` hierarchy.

Figure 4.1 shows a part of the class hierarchy. The entire class diagram cannot be shown as there are over a thousand different classes[1]. The topmost node, the root, of a concrete Clang AST is called the translation unit declaration (`TranslationUnitDecl`). Edges between nodes are simplified, as each node stores a container of its children.

Listing 4.2 contains a short program written in C++. The source code was provided to a Clang tool clang-check, which dumped the abstract syntax subtree of a given function. In this case, the filter was set to the `main` function. The AST dump visualizes the subtree using ASCII characters and node information. Nodes entries start with their type names. Each node also carries its address, source location, and description. Note that the root of the subtree is of type `FunctionDecl`. The usual root `TranslationUnitDecl` is absent due to the function filter being applied.

4.2.2 Representation

The Clang AST attempts to represent the source code as faithfully as possible. It can be said that Clang's AST is closer to C, C++, and Objective-C code and grammar than other ASTs. To achieve the best accuracy in reproducing a source code file, it must save additional data besides the AST. This supplementary data

[1]The class hierarchy is shown in Clang's Doxygen documentation. An example of the Stmt hierarchy can be found at `https://clang.llvm.org/doxygen/classclang_1_1Stmt.html`.

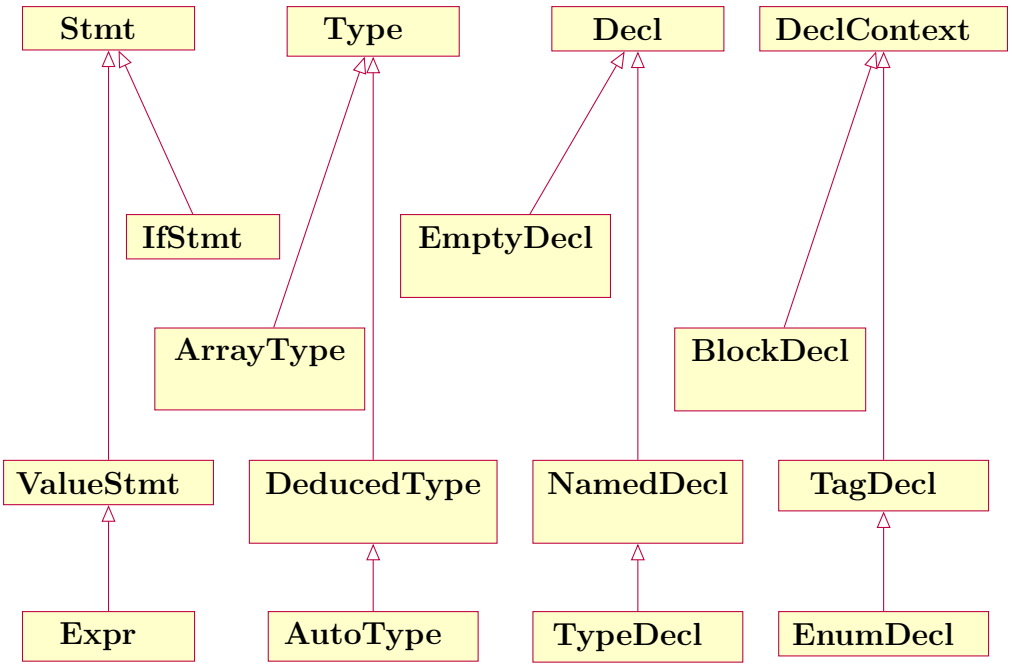

Figure 4.1: An example of the Clang AST class hierarchy. The figure contains only a handful of classes and their children. Note that the top most classes do not share a common ancestor.

makes information that would be lost otherwise, such as compile-time constants, available in the unreduced form.

For each parsed source code file, an instance of `ASTContext` is used to represent the AST. The `ASTContext` allows the programmer to use many valuable methods. Table 4.1 contains a part of `ASTContext`'s Doxygen documentation. It mainly presents methods that were used in this project.

The ASTContext bundles Clang's AST for a translation unit and allows its traversal from the `getTranslationUnitDecl` point, which is the file's highest

Return value	Method name	Description
`DynTypedNodeList`	`getParents(const NodeT &Node)`	Forwards to get node parents from the `ParentMapContext`.
`SourceManager&`	`getSourceManager()`	—
`const TargetInfo&`	`getTargetInfo() const`	—
`const LangOptions&`	`getLangOpts() const`	—
`TranslationUnit-Decl*`	`getTranslationUnitDecl()`	—

Table 4.1: Digest of `ASTContext`'s documentation. The documentation can be found at `https://clang.llvm.org/doxygen/classclang_1_1ASTContext.html`.

```
$ cat -n simple.cpp
    1 #include<iostream>
    2
    3 int main()
    4 {
    5     int x;
    6     std::cin >> x;
    7
    8     return (x / 42);
    9 }
$ clang-check -ast-dump -ast-dump-filter=main simple.cpp --
Dumping main:
FunctionDecl '...' <./simple...> line:3:5 main 'int ()'
`-CompoundStmt 0x556041ab84a0 <line:4:1, line:9:1>
  |-DeclStmt 0x556041ab6900 <line:5:2, col:7>
  | `-VarDecl 0x556041ab6898 <col:2, col:6> col:6 used x 'int'
  |-CXXOperatorCallExpr '...' <line:6:2, col:14> 'std::bas...'
  | |-ImplicitCastExpr 0x556041ab83a0 <col:11> 'std::basic...'
  | | `-DeclRefExpr 0x556041ab8318 <col:11> 'std::basic...'
  | |-DeclRefExpr 0x556041ab6980 <col:2, col:7> 'std::istr...'
  | `-DeclRefExpr '...' <col:14> 'int' lvalue Var '...' 'x'
  `-ReturnStmt 0x556041ab8490 <line:8:2, col:16>
    `-ParenExpr 0x556041ab8470 <col:9, col:16> 'int'
      `-BinaryOperator '...' <col:10, col:14> 'int' '/'
        |-ImplicitCastExpr '...' <col:10> 'int' <LValueToRValue>
        | `-DeclRefExpr 0x556041ab83f8 <col:10> 'int...'
        `-IntegerLiteral 0x556041ab8418 <col:14> 'int' 42
```

Figure 4.2: Clang AST Dump. The example source code visible in the figure has been filtered by function name and fed to a Clang tool.

node. Additionally, the context has access to the identifier table and the source manager. The `SourceManager` class offloads some of the data from AST's nodes. Nodes store their `SourceLocation`. The location is not in its complete form since it is required to be small in size. Instead, the node's full location is referenced in `SourceManager`.

Extracting Clang AST comes at the cost of compiling the program's source code. Usually, this is done using an instance of `FrontEndAction`, which specifies what and how should be compiled. The front-end compilation is essential to note because it can affect LibTooling's performance on large projects. In comparison, clang-format does not execute any compilations. Therefore, clang-format runs efficiently on large projects and correctly on incomplete ones. The compilation action also implies that LibTooling tools often do not support incomplete source codes. The same can be said for programs that contain compile-time errors.

An additional characteristic of Clang's AST is its immutability. The AST has strong invariants that might be broken upon changing its structure. Generally, changes to the Clang AST are strongly discouraged, although some changes happen internally. Those changes include template instantiation.

```
1  /**
2   * Creates a consumer, performs actions after
3   * the AST traversal.
4   */
5  class CountAction final : public ASTFrontendAction
6  {
7    int statementCount_;
8
9  public:
10
11   // Perform the desired action after the traversal.
12   void EndSourceFileAction() override
13   {
14     outs() << "Statement count: "
15       << statementCount_ << "\n";
16   }
17
18   std::unique_ptr<ASTConsumer> CreateASTConsumer(
19     CompilerInstance& ci, StringRef file) override
20   {
21     // Pass any data to the consumer.
22     return std::unique_ptr<ASTConsumer>(
23       std::make_unique<CountASTConsumer>(
24         &ci, statementCount_));
25   }
26 };
```

Figure 4.3: Custom ASTFrontendAction. An instance can be created before parsing a source file. The example shows the ability to perform a body of actions after the file is parsed.

4.2.3 Traversal

Traversing the Clang AST is possible through two different APIs. First, it is possible to invoke an `ASTFrontendAction` instance, which creates and manages an instance of `ASTConsumer`. The latter then constructs the `ASTRecursiveVisitor` object and calls the visitor's methods. The front-end action is invoked upon parsing a source file. The action can be overridden to create a consumer and pass any necessary data to it. For example, this data might include references to variables used for counting objects in the AST or more complicated constructs.

Listing 4.3 showcases an example of such frontend action. The custom class contains a variable used for counting statements in the source code. The reference to that variable is passed further when creating a consumer. After the source code is parsed, the overridden `EndSourceFileAction` method is launched. Inside the method's body, the data gathered during the traversal is displayed.

```
1  /**
2   * Dispatches the CountASTVisitor on the translation
3   * unit decl.
4   */
5  class CountASTConsumer final : public ASTConsumer
6  {
7    std::unique_ptr<CountASTVisitor> visitor_;
8
9  public:
10   // Pass any desired data to the visitor.
11   CountASTConsumer(CompilerInstance* ci, int& counter)
12     : visitor_(std::make_unique<CountASTVisitor>(ci,
         counter)) { }
13
14   void HandleTranslationUnit(ASTContext& context) override
15   {
16     // Use the ASTContext to reference
17     // the translation unit decl.
18     visitor_->TraverseDecl(
19       context.getTranslationUnitDecl());
20   }
21 };
```

Figure 4.4: An example of a custom ASTConsumer implementation. Showcased is the ability to transfer data to a visitor and to dispatch the visitor.

The `ASTConsumer`'s job is to read the Clang AST and handle actions on the tree's specific items. One such action is `HandleTopLevelDecl()`, which, as the name suggests, handles the highest priority declaration in a file. These handle functions are overridable. The consumer also keeps track of a visitor implemented by inhering from the ASTRecursiveVisitor class. The consumer dispatches the visitor from overridden handle methods. However, it is not always beneficial to override granular handle methods. Handling specific events in the consumer might lead to an intriguing case in which a part of the code is parsed while the rest is not. This unwanted behavior can be avoided by overriding just the `HandleTranslationUnit()` method. The translation unit is handled once the entire source file is parsed. Dispatching the visitor internally from a consumer is the preferred approach. Visit methods of the `ASTRecursiveVisitor` should not be called directly. Details concerning the visitor can be found in the following section.

In the example shown in listing 4.4, the consumer passes variable references to the visitor. These references have previously been attained from the frontend action. A reference to the constructed visitor is stored inside the consumer. The visitor is then dispatched in the overridden `HandleTranslationUnit` method. As was described earlier, `ASTContext` helps to retrieve references to top-level nodes.

Second, one can use AST Matchers. Matchers, unlike the visitor approach, do not require a complicated setup. Instead, they provide a query-like syntax for matching Clangs AST's nodes. Matchers will be talked about in detail later.

4.3 ASTVisitor

LibTooling offers a built-in curiously recurring template pattern (CRTP) visitor. The class `RecursiveASTVisitor` [14] offers `Visit` methods that can be overridden to the programmer's liking. Each override specifies the type of node on which the method triggers and the actions that should be performed. A portion of these methods is presented in table 4.2. The table's contents are based on the Doxygen documentation.

The implementation seen on listing 4.5 illustrates the idea. a custom class with a strict dedication, i.e., counting program's statements, has two visit functions. Firstly, a `VisitStmt` method, which is triggered upon encountering a node of type `Stmt`, as seen in its parameters. Furthermore, since no additional visit functions for children of `Stmt` have been overridden, `VisitStmt` will trigger on every node type inheriting from `Stmt` as well. Secondly, the method `VisitVarDecl` only accepts `VarDecl` and its inheriting types. Because `VarDecl` is a child of `Decl`, not the other way around, `Decl` will not trigger this visit function. Typically,

Method[a]	**Description**
`shouldVisitImplicitCode()`	Return whether this visitor should recurse into implicit code, e.g., implicit constructors and destructors.
`shouldTraversePostOrder()`	Return whether this visitor should traverse post-order.
`TraverseAST(ASTContext &AST)`	Recursively visits an entire AST, starting from the top-level Decls in the AST traversal scope (by default, the `TranslationUnitDecl`).
`TraverseStmt(Stmt *S, DataRecursionQueue *Queue=nullptr)`	Recursively visit a statement or expression, by dispatching to `Traverse*()` based on the argument's dynamic type.
`TraverseType(QualType T)`	Recursively visit a type, by dispatching to `Traverse*Type()` based on the argument's `getTypeClass()` property.
`TraverseDecl(Decl *D)`	Recursively visit a declaration, by dispatching to `Traverse*Decl()` based on the argument's dynamic type.
`WalkUpFromStmt(Stmt *S)`	—
`VisitStmt(Stmt *S)`	—
`WalkUpFromType(Type *T)`	—
`VisitType(Type *T)`	—
`WalkUpFromDecl(Decl *D)`	—
`VisitDecl(Decl *D)`	—

Note: [a] All presented methods return `bool`.

Table 4.2: Digest of `RecursiveASTVisitor`'s documentation. The documentation can be found at `https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html`.

when using less specific visit methods, a good way of differentiating node types is casting them dynamically.

```
1  /**
2   * Counts the number of statements.
3   */
4  class CountASTVisitor : public
       clang::RecursiveASTVisitor<CountASTVisitor>
5  {
6      clang::ASTContext& astContext_;
7      int& statementCount_;
8
9  public:
10     CountASTVisitor(clang::CompilerInstance* ci, int&
       counter)
11         : astContext_(&ci->getASTContext()),
12           statementCount_(counter) { }
13
14     // Perform a body of actions upon
15     // encountering a statement.
16     virtual bool VisitStmt(clang::Stmt* st)
17     {
18         outs() << "Found a statement.\n";
19         statementCount_++;
20
21         return true;
22     }
23
24     // Perform a body of actions upon encountering
25     // a variable declaration.
26     virtual bool VisitVarDecl(clang::VarDecl* decl)
27     {
28         outs() << "Found a variable declaration.\n";
29
30         return true;
31     }
32 };
```

Figure 4.5: CountASTVisitor. A custom implementation of the ASTRecursiveVisitor which tracks the number of encountered statements.

Visiting statements, expressions, declarations, and types is straightforward. The same applies to children of these classes. However, it is challenging to visit more complicated entities such as nested types, e.g., `int* const* x`. Such cases require navigation through source locations in order to reach a particular built-in type. In an example from the LLVM Euro Conference 2013[2], one can reach

[2]The particular speech in which the example is mentioned can be found at `https://youtu.be/VqCkCDFLSsc?t=916`. The recording starts at the relevant slide.

the built-in type of `int * p;` in two ways. The declaration is for a pointer type, which has a `PointerTypeLoc`. On the one hand, it is possible to reach the `BuiltinType` node by calling the `getPointeeLoc()` method. The result is a `BuiltinTypeLoc` instance, through which a `QualType` object can be extracted. The qualifier leads to the desired `BuiltinType` instance. On the other hand, one can extract a `QualType` object from the starting `PointerTypeLoc` node and use it to get a `PointerType` instance. By calling the `getPointeeType()` method, it is possible to get to the `QualType` node that leads to the desired built-in type.

Both of these approaches start at the same source location and end with the same built-in type object. The steps necessary to traverse this simple pointer type, however, were not trivial.

The `RecursiveASTVisitor` is launched by visiting the root node using a `TraverseDecl` method. It then dispatches to other nodes and their children. For each node, the visitor searches the class hierarchy from the node's dynamic type up. Once the type is determined, the visitor calls the appropriate overridden `Visit` method. Traversing the class hierarchy this way translates to calling the methods for abstract types first, followed by more specific visit functions.

The tree traversal can be done in a preorder or postorder fashion. Preorder traversal is the default. the developer can also stop the traversal at any point by returning `false` from the visit function as opposed to `true`.

4.4 Matchers

Clang's ASTMatchers [15] is a domain-specific language (DSL) used for querying specified AST nodes. Each matcher represents a predicate on nodes. Together, they form a query-like expression that matches particular nodes. Like the rest of LibTooling, the DSL is written in C++ and is used from C++ as well. Matchers are useful for query tools and code transformations. In a query tool, one might want to extract a niche subset of Clang's AST, inspect it, and perhaps perform some action on it. Similarly, refactoring tools can use matchers to navigate and extract similar nodes, rewrite their source code, or add descriptive comments. A matcher will match on some adequate node. It might match multiple times if the AST has enough of these nodes. When combined, multiple matchers form a matcher expression. Such expression can be seen as a query for the Clang AST. The expression reads like an English sentence, from left to right, alternating several type-specifying and node-narrowing matchers.

All available matchers fall into three basic categories. The first one being node matchers. Node matcher's job is to match a specific type of AST node. An example of such a matcher could be the `binaryOperator(...)` matcher, whose purpose is to look for nodes of that exact type: `BinaryOperator`. Node matchers are the core of matcher expressions. Expressions start with them, and they specify which node type is expected. Node matchers also serve as arguments for other matcher types. Furthermore, they allow binding nodes. Binding nodes allows the programmer to retrieve matched nodes later and use them for code transformation tasks.

The second category, called narrowing matchers, serves a different purpose. By matching specific attributes on the current AST node, they narrow down the search range. Narrowing matchers allow specifying more granular demands for

the searched node. A concrete example would be the `hasOperatorName("+")` matcher. As one might guess, this matcher narrows down the search to those nodes whose binary operator is the plus sign. Narrowing matchers also provide more general logical matchers. These include `allOf`, `anyOf`, `anything`, and `unless`.

The last category specifies the relationship between nodes. Traversal matchers are used for filtering reachable nodes based on the AST's structure. Most notably, they include matchers for specifying node's children, such as `has`, `hasDescendant`, and `forEachDescendant`. Traversal matchers take node matchers, the first category, as arguments. For example, the `hasLHS(integerLiteral(equals(0)))` matcher specifies the requirement for the current node to have the given child. In this case, it is an integer with the value 0 on the left hand side.

Together, these three examples form a matcher expression found in the AST Matcher tutorial[3]. Going by the mentioned rules of building an expression, it would have the following form:

```
binaryOperator(hasOperatorName("+"),
    hasLHS(integerLiteral(equals(0)))).
```

Figure 4.6: Matcher expression.

The expression in listing 4.6 searches for a binary operator. The search is further narrowed to a plus sign with a zero left-hand side of the operation.

In the tool, expressions are build by calling a creator function. The expression is then represented as a tree of matchers. While the developer has access to a plethora of predefined matchers, as seen in the Matchers Reference [16], they can define custom ones as well. Creating a custom matcher can be done in two ways. First, a matcher can be created by inheriting an existing Matcher class and overriding it to one's liking. Second, one can use a matcher creation macro. These macros specify the type, the name, and the parameters of the matcher.

The default behavior, defined by the `AsIs` mode, is to traverse the entire AST and visit all nodes, including implicit ones. Implicit nodes might include constructs omitted in the source code, such as parentheses. Working with these nodes increases the difficulty of writing matcher expressions severely since it requires a deep knowledge of the AST's hierarchy and its corner-cases. The traversal mode can, however, be changed to ignore implicit nodes. One such traversal mode is `IgnoreUnlessSpelledInSource`, which conveniently only looks at nodes represented by the source code.

4.5 Source-to-source transformation

To transform source code based on its AST, the programmer must extract the AST from the code, alter the AST, and then translate it back to valid source code. LibTooling allows the programmer to extract the AST and examine it. Additional

[3]The AST Matcher tutorial contains valuable practical information as well as a well-written introduction to matchers. It can be found at `https://clang.llvm.org/docs/LibASTMatchersTutorial.html`.

functionality also allows modifying the AST both directly and indirectly [17]. However, there are obstacles and limitations to both approaches.

Let us examine the pitfalls of direct AST transformation first. Before explaining the possibilities of direct modifications, it should be noted that these transformations are not recommended. Clang has powerful invariants about its AST, and changes might break them. Although it is not encouraged, the methods to change the AST are available.

Given an `ASTContext`, it is possible to create specific nodes using their `Create` method. Likewise, nodes with public constructors and destructors can combine keywords `placement new`, `delete` and the ASTContext to add or remove nodes. The job of `ASTContext` is then to manage the memory internally.

A more sophisticated approach is the one offered by the `TreeTransform` class. Although it is rarely used and no real examples can be found, the premise is simple. The `TreeTransform` class needs to be inherited from, and its `Rebuild` methods need to be overridden. The overrides then transform specified nodes of an input AST into a modified AST.

One additional dirty way of replacing nodes is by utilizing `std::replace`. The child container of the replaced node's immediate parent must be specified in parameters of `std::replace`, together with the node itself and the new node.

When attempting to modify the AST indirectly, which is how LibTooling intends it to, the developer can run into a couple of issues. First of all, the AST does not reference the source code entirely. The programmer has access to `SourceManager`, `Lexer`, `Rewriter`, and `Replacement` classes. When used individually or in combinations, they can map to and alter a given node's source code. It is then possible to add, remove, or replace the AST's underlying code with node-level precision.

Accessing this information through these classes can result in node-to-code mapping issues. Compound statements might mismatch parentheses and curly brackets. Similarly, declarations and statements might miss a reference to a semicolon. These and more obstacles could surface anytime a programmer attempts to debug their source-to-source transformation tool.

The programmer must be careful in managing object instances when transforming multiple files. Each source file creates a new FrontendAction, and with it, the developer needs a new instance of Rewriter.

Discovering these obstacles is not as straightforward and intuitive as the rest of the LibTooling framework. Templates, the language feature of C++, further complicate the matter. In Clang AST, multiple types derived from a template might share some nodes. Having multiple parent nodes is also not uncommon for template types. Thankfully, templates are rarely used. A more common threat, macros, has a similar effect. Modifying a source code containing macros and comments results in losing both.

TODO: Show an example of instrumentation code.

Doing source-to-source transformation is often accompanied by inserting instrumentation code. By performing so-called cross-checking, one can make sure that the transformation behaves as intended. Cross-checking works by inserting code with the same behavior into the original and the transformed source code. This insertion can be done in a sophisticated manner using the AST. If, for example, the transformation alters calls to functions in the code, the instrumentation

code should be inserted inside the function's body—that way, the developer can check whether the transformation had its intended result. Cross-checking is a safe way of ensuring source-to-source transformations work as intended. While they might be excessive for small refactorings, they are beneficial when debugging source-to-source transformations of larger scales, such as translating one language's source code to another's.

5. Program minimization

As described in the first chapter, debugging is a time-consuming task. Any amount of help with debugging is always appreciated by developers. In this project, we attempt to help by providing means to minimize the debugged program w.r.t. a given runtime error. The minimization's goal is to reduce the amount of source code programmers must go through when debugging, thus speeding up the process. The size reduction of the program should be fully automated and reasonably fast on simple inputs. Furthermore, it should correctly handle any source code from the program domain specified below. Great attention is given to the accuracy with which the minimizing algorithms work and their time efficiency.

The domain in which the algorithms that are shown below operate can be described as small and simple projects. The presented approaches take into consideration code written in C and C++. Support for more complicated concepts of those languages, such as templates, is omitted. Additionally, programs that involve multiple threads and other advanced features that might trigger nondeterministic behaviour are also not taken into consideration. Programs that rely on randomly generated numbers during their runtime do not fit into the domain as well. This is because executions of multithreaded and random programs cannot be easily reproduced. Instead, the program minimization described in this project focuses on simple single-threaded console applications with consistent executions.

The problem of program minimization while preserving runtime errors can be described as follows. Assume that a developer has encountered a runtime error in his application. Using logging or debugging tools, he can extract the stack trace at that given point. The stack trace provides valuable information for a minimizing algorithm. The presented algorithms notably require a description of the error and the source code location at which the error was produced. Based on the described scenario, we can draw the following definitions.

Definition 7 (Location)**.** *Let loc:* $\mathcal{S} \mapsto \{x | x = (file, line, col)\}$*, where* $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ *is the set of program's statements. We call the result of* $loc(S_i)$ *the* location *of statement* S_i*.*

The source code's location is specified by a file name, the line number, and on that line, the number of characters from the left. The location could be described in further detail by including starting and ending points. However, in this simplified description, only the starting point is taken into consideration.

Definition 8 (Failure-inducing statement)**.** *Let* $E = (location, desc)$ *be a runtime error specified by its location and description. Let the program* $\mathcal{P} = (S_1, S_2, \ldots,$- $S_n)$ *result in E upon execution. We call* S_i *the* failure-inducing statement *of E if* $loc(S_i) = E(location)$*.*

Failure-inducing statements are the sites from which an error was thrown. That means the statements were present at the error's location when the error occurred.

Having found the site's source code location, the developer can now investigate the source code for a potential bug. In the process, he might consider the values of

application arguments present at launch-time and change his debugging process accordingly. Nonetheless, the developer has to look through the source code to find the error's root cause. This exact point is where the source code size reduction starts being beneficial. Using static and dynamic analysis, it is possible to effectively and safely remove unnecessary source code. Such code includes statements, declarations, and expressions that do not affect the program's state at the point given by the error. With additional verification, it is also possible to remove code constructs that affect the state, but the error occurs regardless of whether they are present or not.

The reduced program's source code can then be used for debugging the given runtime error. The newly generated program has to fulfill the following invariant.

Invariant 1 (Location alignment)**.** *Every program $\mathcal{P}'$ created by reducing the original program $\mathcal{P}$ based on dynamic information given by the execution of $\mathcal{P}$ with arguments A must result in the same runtime error E. The error's absolute location can differ; it must, however, occur in the same context.*

The rule specifies that a program must end in the same runtime error as the original program to be considered a correctly reduced variant. Though, with the change to the program's size, the location of failure-inducing statements also changes. In $\mathcal{P}$, the error's location in the file should be lower compared to $\mathcal{P}'$ since $\mathcal{P}'$ has less code in general. Stress is placed on the location's context in which the error arises. As long as locations in $\mathcal{P}'$ are adjusted based on those in $\mathcal{P}$, the absolute location of the error does not matter.

Figure 5.1 contains an example of $\mathcal{P}$ and its minimal variant $\mathcal{P}'$. All statements that do not directly contribute to the specified runtime error are removed. A non-minimal variant might contain additional non-impactful lines, such as line 37.

So far, this chapter has talked about both minimization and reduction simultaneously. It is crucial to make a distinction between those two terms. In this context, the reduction is simply the process of making the program smaller in size. The reduced program must also result in the same runtime error. Minimization is built on the same rules as reduction; however, it must fulfill one additional property. No statement of the minimized program can be removed while preserving the error. The task of reduction is more straightforward than that of minimization. Finding the program's minimal variant is computationally infeasible for large inputs. Whereas reducing the program to a rough approximation of the optimal solution can be done in polynomial time. Although this project focuses on minimal program variants, we recognize that it is expensive to compute them. Instead, we use reduction and minimization interchangeably throughout the text.

Minimization of programs requires two steps—first, the removal of chunks of the given source code. The following sections describe several techniques of code removal. The naive approach is explained briefly. Possible improvements to that approach concerning runtime are then described. Subsequent approaches employ techniques discussed in chapter 2. The method based on Delta debugging offers a modified version of the debugging algorithm. Another approach combines different types of slicing to achieve the best results.

Second, the minimization needs to perform a validation to determine whether the result meets the required criteria, i.e., minimality (or an approximation) and correctness. The description of naive validation is shown in the sections below.

Listing 5.1: Program $\mathcal{P}$.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  long get_factorial(int n)
5  {
6    // Missing the stopping
7    // constraint
8    // => segmentation fault.
9    return (n *
10     get_factorial(n - 1));
11 }
12
13 int main()
14 {
15   const int n = 20;
16   long loop_result = 1;
17
18   for (int i = 1; i <= n;
19     i++)
20   {
21     loop_result *= i;
22   }
23
24   long recursive_result =
25     get_factorial(n);
26
27   if (loop_result !=
28     recursive_result)
29   {
30     printf("%ld, %ld\n",
31       loop_result,
32       recursive_result);
33
34     return (1);
35   }
36
37   printf("Success.\n");
38
39   return (0);
40 }
```

Listing 5.2: Minimal variant $\mathcal{P}'$.

```
#include <stdio.h>                1
#include <stdlib.h>               2
                                  3
long get_factorial(int n)         4
{                                 5
  // Missing the stopping         6
  // constraint                   7
  // => segmentation fault.       8
  return (n *                     9
    get_factorial(n - 1));        10
}                                 11
                                  12
int main()                        13
{                                 14
  const int n = 20;               15
                                  16
                                  17
                                  18
                                  19
                                  20
                                  21
                                  22
                                  23
  long recursive_result =         24
    get_factorial(n);             25
                                  26
                                  27
                                  28
                                  29
                                  30
                                  31
                                  32
                                  33
                                  34
                                  35
                                  36
                                  37
                                  38
                                  39
}                                 40
```

Figure 5.1: A program resulting in a segmentation fault error and its minimal erroneous variant. The variant is stripped off all statements unnecessary for the error to occur.

5.1 Naive reduction

The simplest approach examined in this project is the naive removal of each source code statement. This technique aims to try every possible variation of the code and find the smallest correct solution through trial and error. All possible variations, both valid and invalid at compile-time, can be generated by separating the source code into units of statements, declarations, and expressions and removing one code unit at a time.

Definition 9 (Code unit)**.** *Let $\mathcal{P}$ be a program consisting of a sequence of statements, expressions, and declarations $(S_1, S_2, \ldots, S_n)$. We call $U_i = (S_{i_1}, S_{i_2}, \ldots, S_{i_n}), U_i \subseteq \mathcal{P}$ a* code unit *if the sequence $(S_{i_1}, S_{i_2}, \ldots, S_{i_n})$ is syntactically correct.*

A code unit represents any syntactically correct subset of the original program. Working with all code units is not practical in our case. Instead, we will focus on atomic code units. Atomic code units are the minor code units in the given program. For example, the atomic code unit might be a `for` loop with its body or an assignment expression, such as `x = 3;`. The rest of the text will refer to atomic code units simply as code units.

Algorithm in figure 5.2 describes the naive process. Once the input source code is provided, it is split into n code units. Every unit has two possible states: it is either kept or removed. Let us represent each statement with a single bit. For a program $\mathcal{P} = \{S_1, \ldots, S_n\}$, we would suffice with a bitfield of size n. This bitfield would keep track of whether each bit is kept or removed. Statement S_i is kept if the i^{th} bit in the bitfield is set to 1. On the other hand, S_i is removed when the i^{th} bit is set to 0. The bitfield representation is used for generating every subset of n given elements. It works by considering the bitfield as an unsigned binary number and gradually incrementing that number. With each increment, a new variant of the bitfield is generated. The very same approach can be used in naive minimization. Analogically to generating every subset of a set of elements, this variant generating algoritm results in 2^n possible variants.

The naive time complexity is, therefore, the abysmal $\mathcal{O}(2^n)$. Moreover, the 2^n variants require some verification and classification to determine whether they are minimal or not. Nevertheless, we can be sure that a set of those 2^n variants contains the desired minimal variant. The correctness of many of the invalid variants can be ruled out immediately since they indeed are not syntactically correct. The rest, however, must be adequately tested for the desired runtime error.

5.1.1 Heuristics

The algorithm can be sped up by using various heuristics. The search space normally contains variants that are syntactically or semantically incorrect. Generating such variants and validating them needlessly wastes time. We can overcome this issue by introducing a mechanism that validates some aspects of the variant beforehand. An example of such a mechanism is keeping track of code unit dependencies. Let us create a directed graph of dependencies. Each node of the graph represents a code unit in the input source code. There is an edge from node u to node v if the code unit v is the subset of code unit u. It is natural to wonder what exactly do these edges achieve.

Input:
$L \ldots$ location of the error.
$P \ldots$ the input source code.
$A \ldots$ the input program's arguments.
Output: The reduced source code.

1: $allVariants \leftarrow \{\}$
2: $(C_1, C_2, \ldots, C_n) \leftarrow$ SplitIntoCodeUnits($currentVariant$)
3: $bitField \leftarrow [bit_{n-1}, bit_{n-2}, \ldots, bit_0]$, $\forall i \in 0 \ldots n-1 : bit_i = 0$
4: **while** $\exists i \in 0 \ldots n-1 : bit_i = 0$ **do**
5: $\quad bitField \leftarrow Increment(bitField)$
6: $\quad currentVariant \leftarrow (C_1, C_2, \ldots, C_n)$
7: $\quad$ **for** $i \in 0 \ldots n-1$ **do**
8: $\quad\quad$ **if** bitField[i] $= 0 \wedge loc(C_{i+1}) \neq L$ **then**
9: $\quad\quad\quad currentVariant \leftarrow currentVariant \setminus \{C_{i+1}\}$
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: $\quad allVariants.Add(currentVariant)$
13: **end while**
14: $allVariants \leftarrow$ SortBySize($allVariants$, $Ascending$)
15: **for all** $V \in allVariants$ **do**
16: $\quad$ **if** IsValid(V, L, A) **then return** V.
17: $\quad$ **end if**
18: **end for**
19: **return** none.

Figure 5.2: Naive Statement Removal.

Listing 5.3: An if - else statement.

```
1 if (x % 2 == 0)
2 {
3   CreateEvenSpacedTable();
4 }
5 else
6 {
7   CreateOddSpacedTable();
8   trim = true;
9 }
```

Listing 5.4: An invalid variant of 5.3.

```
1
2
3 CreateEvenSpacedTable();
4
5
6
7 CreateOddSpacedTable();
8 trim = true;
9
```

Figure 5.3: An example of a semantically incorrect variant. The original source code snippet in Listing 5.3 performs different actions in each branch. A variant with removed control statements shown in Listing 5.4 performs both of those inconsistent actions during the same run.

Figure 5.3 attempts to illustrate the usefulness of those edges. The example shows an `if - else` statement. Let us assume that the naive algorithm attempts to remove two code units. First, the code unit `if(...)` is removed while keeping the statements inside the body. Second, the code unit `else(...)` is removed while, again, keeping the statements inside the body. Originally, the input pro-

gram would never run statements in those two branches in the same execution path. However, the generated variant invalidates the original behavior. The statements from those two branches will be run together. Such variant results in an obvious semantical error. While the described process might lead to a smaller failure-inducing program, we believe that such a program is incorrect.

This semantical error can be avoided by validating the dependency graph mentioned earlier. For this example, we start by creating a node u for the `if(...){}` statement. Once we encounter the body of the `if(...)` statement, we add it to the dependency graph as node v. The `if(...)` statement's children also syntactically contain the `else(...){}` statement. Therefore, we add nodes x and y for the `else(...)` statement and its body, respectively. The nodes v, x, and y are subsets of u. This holds since the code unit u contains both the `if(...)` statement and other parts. Those parts include the statement's body, the else branch, and the body of the else branch. By completing the edges between the four nodes, we get the dependency graph that solves our issue.

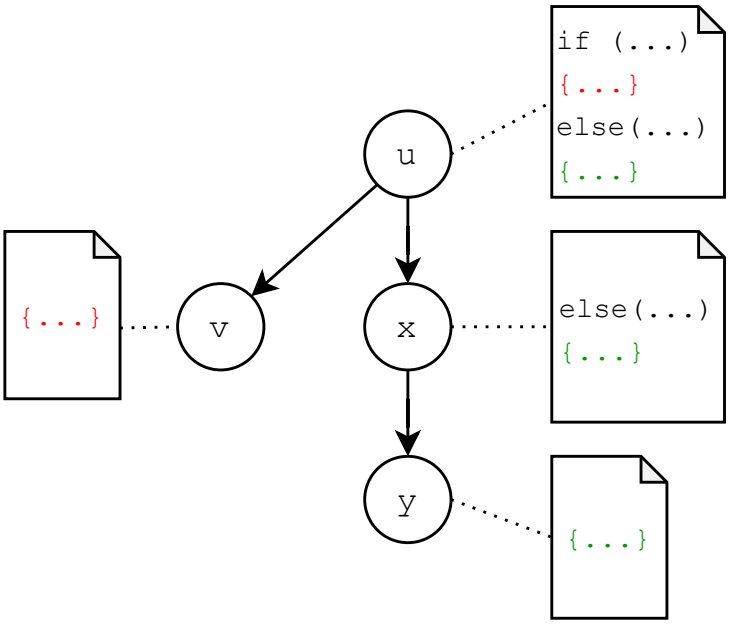

Figure 5.4: A dependency graph of the source code shown in Listing 5.3

The graph, visualized on Figure 5.4, states that the body of the `if(...)` statement depends on the `if(...)` statement. Analogically, the `else(...){}` statement and its body depend on the `if(...)` statement. We can use this information and only remove parent nodes if all their children are removed as well. This simple rule prevents invalid execution flow in the shown example. A variant generated by following this rule has the bodies of `if(...)` and `else(...)` only if `if(...)` and `else(...)` are present as well. Creating and following a graph of dependencies is a heuristic worth deploying in this project. It can also be extended further to include syntax errors. One such error that immediately comes to mind is referencing a removed declaration. Assume that the algorithm has generated a variant in which a variable declaration is removed, such as the one shown in Figure 5.5. The algorithm has not, however, removed usages of the declared variable. Such variant will fail at compile time. However, we can

avoid generating this variant by creating edges from the variable declaration u to variable references $\{v_1, v_2, \ldots, v_k\}$.

Listing 5.5: A variable declaration and its usage.

```
1 int factor = 3;
2 int x = 0xffff;
3
4 while (x > 0)
5 {
6   std::cout << factor * x
      << "\n";
7 }
```

Listing 5.6: Reference to an invalid declaration.

```
int factor = 3;                  1
                                 2
                                 3
while (x > 0)                    4
{                                5
  std::cout << factor * x        6
     << "\n";
}                                7
```

Figure 5.5: An example of a syntactically incorrect variant. The declaration of `x` in Listing 5.5 was removed in Listing 5.6, while the usage of `x` was kept.

Another helpful heuristic involves predicting a variant's size before the variant is generated. We can significantly improve the best case and the average time complexities by introducing a size metric. Let us keep track of every code unit's size. Upon creating a bitfield, we can determine the variant's size by analyzing the bitfield. By summing up the size of each present code unit, i.e., each bit set to 1, we get the variant's total size. Such metrics can be used to enumerate bitfields and sort them based on their size. Once we have a sorted list of bitfields, we can start generating variants and validating them. We can then terminate the algorithm upon finding the first valid variant. This heuristic saves time and memory that would otherwise be spent on generating all possible variants.

5.2 Delta debugging

Zeller's Delta debugging [1, 2, 18] has been described in detail in section 2.1. It is an automated debugging technique that focuses on input size reduction. For a given input, Delta debugging attempts to find the input's smallest failure-inducing subset and isolate the a failure-inducing element using two algorithms. This project is only concerned with the minimizing algorithm described in section 2.1 and in the figure 2.1.

For the rest of this section, the input will be referred to as a *test case*. Other than the test case, Delta debugging also requires the debugged program (in its executable form) and a method of validating its output. Let us draw parallels between the mentioned requirements and this project's minimization task. The input for program minimization is the given program's source code. The code can be labeled as the test case Delta debugging takes. It is essential to clarify that this Delta debugging usage does not utilize the source code as the debugged program. Instead, it considers it as a given test case. Then, we must specify the expected output and a method to validate it. It is required that the program terminates with a given runtime error. Let us label that runtime error and its location as the expected output. Whenever the executed test case results in that particular

error, we interpret the run as a failing one. Every other terminating run will be interpreted as a passing one. Lastly, we must define the debugged program. In our case, to get from the test case (source code) to the expected output (a specific runtime error), the code must first be compiled and then executed. The fitting debugged program is, therefore, a pipeline of a compiler and an execution environment.

The minimizing algorithm uses binary search in a greedy manner. We have modified the way binary search is performed to better fit the input's structure. The original minimizing algorithm operates with partitions of equal size. That is not necessarily the best approach for structured test cases such as source code. To give a concrete example, we can look at splitting a function definition into two partitions. Originally, we would get two syntactically invalid code snippets. One would contain the function's head and the first half of its body. The body would not contain the terminating curly bracket. Similarly, the second would be missing an opening curly bracket. Instead, it makes more sense to operate on code units. A more detailed description of code units can be found in section 5.1.

Initially, the test case is split into k code units. Those code units are then assigned into n partitions of roughly the same size. The number of partitions changes based on the current iteration, as described in figure 2.1. In our case, each iteration compiles current test case subsets and executes them. Executions are validated as described earlier and the algorithm reduces the size of the test case gradually. The result is roughly what we need - a minimal program variant approximation that fails with a particular error. As was already mentioned in this chapter's introduction, the location of the error might differ based on the variant's structure. Nonetheless, the location could be aligned based on the source code in an additional step, so that Invariant 1 holds.

The running time complexity of this modified algorithm, which is measured for the number of executed validations v, remains unchanged. Let us assume the test case consists of k code units. Zeller and Hildebrandt[2] presented both the worst and the best case complexity as follows.

Worst case. Two possible scenarios lead to the worst time complexity. First, every executed validation is inconclusive. That would lead to $v = 2+4+8+\ldots+2*k = 4*k$ validations. Second, the validation succeeds, i.e., finds a failing subset, for every last complement. This case given us $v = (k-1)+(k-2)+\ldots+2 = k^2-k$ validations. Combined, these two scenarios lead to k^2+3*k validations at worst.

Best case. The best case is the ideal scenario for utilizing binary search. We would be searching for a single failure-inducing code unit. This scenario leads to $2 * \log k$ validations.

Extracting an approximation of the minimal program variant after $\mathcal{O}(k^2)$ validations is undoubtedly practical. Especially when compared to the exponential time complexity of the naive approach described in section 5.1. Those who desire the minimal variant might want to get the approximation first and provide it to the naive reduction. Nevertheless, there is no way of avoiding the exponential complexity when searching for optimal results.

5.3 Slicing-based solution

The slicing-based approach attempts to help with the shortcomings of the naive reduction described in Section 5.1. The algorithm described below combines static and dynamic slicing, minimization using Delta debugging, and the naive algorithm. The primary point is its preprocessing steps described below.

It is crucial to keep the input size as low as possible due to the naive algorithm's exponential complexity. The input's size can be significantly reduced by slicing the input program as a preprocessing step. Let us compare the two slicing techniques described in Section 2.2 and Section 2.3 and apply them to our problem. The main focus of the comparison should be on two aspects - the size of the slice and the running time of the slicing algorithm.

It is known that dynamic slices are the smallest they can be. However, they require information available at execution time. The question is whether running the program is necessary. We know that the program that is being minimized has been run before. Hence the availability of the information about the encountered runtime error. If the program ran deterministically, it would have to terminate in future executions as well. That is considering it would run with the same arguments as previously. We can therefore conclude that the program terminates. The time of the termination might vary depending on the purpose of the program. For server-like applications, it might take months to encounter an error at runtime. Static slicing does not suffer from the mentioned issue. It is inexpensive in terms of execution time regardless of the purpose of the sliced program.

We see that static slicing is a significantly less expensive operation in terms of running time. Out of the two main issues concerning the previously discussed approaches - the input size and its execution time - static slicing helps to eliminate both. Both the input size and subsequent execution time could both be brought down further by using dynamic slicing. The usage of dynamic slices, as opposed to static, has the mentioned benefit of generating smaller slices. On the other hand, it has definitive limitations. One such constraint is the requirement to run the said program.

However, one can perform preprocessing steps to help dynamic slicing run more efficiently. Let us consider a program that performs multiple demanding tasks such as computations. These tasks are primarily independent, and their running time is longspun. Using dynamic slicing alone would be inconvenient. However, by first employing static slicing to remove these long-running unnecessary tasks, the program's execution time can be significantly reduced. The reduced program could then be sliced dynamically. The result would be a minimal slice at a fraction of the original time compared to dynamic slicing alone. This crafted ideal use case only concerns a very narrow range of existing programs. However, static slicing could be used before just any attempt at dynamic slicing due to its low running time.

The improvement in the form of a static slice is genuinely convenient. However, checking whether the improvement has any effect before running dynamic slicing is not an easy task. The issue stems from the halting problem [19] and Rice's theorem [20]. The halting problem states that it is undecidable whether a program terminates on its particular input. Rice expanded the thought further by stating that all interesting semantic properties of a program are undecidable.

Without proper and accurate means to determine many wanted properties, we are required to approximate them.

Amongst such properties is the factor of how effective static slicing is. The approximation will be required in the following sections as well. In particular, the Section concerning program validation will look at this issue in more detail. One way of guessing the effectiveness of static slicing in terms of size reduction is by analyzing the program's branching factor. We can approximate static slicing's relative performance by employing a metric for the number and density of control-flow altering statements. It is assumed that programs with a high branching factor, i.e., with more control-flow-altering statements, are less likely to reduce their size during static slicing. Nonetheless, slicing statically before doing so dynamically has been a rule of thumb for this project.

We can, however, reap the benefits of both static and dynamic slicing while avoiding running the program. We need to follow the ensuing thought process. Since static slicing does not handle branching and other control statements nearly as efficiently as dynamic slicing, we can employ a trick to help. Using the same additional input information as dynamic slicing, i.e., program's arguments, we can provide more specific information to the static slicing algorithm. All that is required is to define the arguments with their respective values inside the code.

This process will be referred to as *argument injection*. Slices generated from this modified source code will be more precise since they will not contain unnecessary branching. It is important to restate that this modification only affects control statements dependent on the program's arguments. If the arguments do not appear in the original, unmodified static slice, their values will not affect the slice's size. Input modified using argument injection is guaranteed to be smaller or equal in size.

The proposed slicing-based solution is described in figure 5.6. The input program is sliced statically w.r.t. every variable available at the failure-inducing line. The slices are then unified and given as the input to a dynamic slicer. Similarly, the dynamic slicer generates slices w.r.t. those potentially failure-inducing variables. Those dynamic slices are then unified.

The intermediate result extracted after performing the two slicing types should be significantly smaller than the original program. Since the result so far contains slices for multiple variables, it might not be minimal yet. However, it can be assumed that it is valid, i.e., ends with the desired runtime error. Using the observations made in Section 5.1 and Section 5.2, we can create an efficient and precise minimizing algorithm that takes care of the penultimate step in Figure 5.6. We utilize a pipeline of the minimizing Delta debugging algorithm and the naive reduction:

1. The sliced intermediate result is fed to the Delta debugging algorithm. Due to its smaller size, the intermediate result contributes to a lower amount of Delta iterations. Therefore, Delta produces a local minimum more efficiently.

2. The local minimum is optimized to the minimal variant by running the naive reduction.

Each step of the preprocessing and the pipeline leads to a smaller result. Additionally, each step benefits from the size reduction caused by the previous steps.

Input:
L ... location of the error.
P ... the input source code.
A ... the input program's arguments.
Output: The reduced source code.

```
1: S ← GetStatementAtLocation(L)
2: variableList ← {}
3: for all Expr ∈ S do
4:     if Expr is Variable then
5:         variableList.Add(Expr)
6:     end if
7: end for
8: sliceList ← {}
9: for all V ∈ variableList do
10:    sliceList.Add(StaticSlice(P, L, V))
11: end for
12: unifiedSlice ← Unify(sliceList)
13: P′ ← Compile(unifiedSlice)
14: L′ ← AdjustLocation(P, P′, L)
15: sliceList ← {}
16: for all V ∈ variableList do
17:    sliceList.Add(DynamicSlice(P′, L′, V, A))
18: end for
19: unifiedSlice ← Unify(sliceList)
20: P′ ← Compile(unifiedSlice)
21: L′ ← AdjustLocation(P, P′, L)
22: P′ ← PreciseReduction(P′, L′, A)
23: return P′.
```

Figure 5.6: Minimization Based on Slicing.

Another thought-about approach is hybrid slicing [21]. This slicing technique is a compromise between static and dynamic slicing. The main benefit is that it produces smaller slices than static slicing. It also uses fewer resources than dynamic slicing. Hybrid slicing works by stopping at a set of breakpoints and using the information available at those points. The comparison of hybrid slicing and the combination of static and dynamic could yield exciting results. It can be assumed that hybrid slicing would be more effective on smaller programs with a short execution time. The static-dynamic combination could work better on larger-scale applications, where static slicing can remove unnecessarily long-running chunks of code.

5.4 Program validation

Once we generate a variant, we must ensure that the variant is correct. A correct variant results in the same runtime error as the original program. Moreover, the error must arise in the exact location. Section 5.3 introduced the reader to Rice's theorem. This theorem is relevant in this chapter as well. By validating a variant,

we are searching for a non-trivial property that cannot be obtained statically. This fact leaves two possible options. We can either determine the validness naively or approximate it. The latter is undoubtedly more exciting than the former. There is room for improvement when it comes to approximations. We could have come up with new methods altogether. While there might be several sophisticated validation techniques involving flow analysis, instrumentation, and even pattern recognition, we settled for a naive approach. This decision significantly lowers the amount of time required for this project. Furthermore, it allows us to focus on the minimization itself.

A simple way of finding the minimal failure-inducing subset of the program is by performing the following steps:

- We attempt to compile each variant. If a variant fails, we can rule it out definitely.
- We execute a static analyzer and check its warnings. If a fatal warning or an error are generated, we rule the variant out as well.
- We launch an execution environment that provides us with symbol information of the running program. We run the variant inside that environment and observe its output. If the program crashes and generates the same error, it is considered a valid variant. Otherwise, it is ruled out.

These three steps define three barriers a program must overcome. Each barrier has a great chance of invaliding a typical variant. This chance increases the further into the verification the variant gets. Let us discuss these three steps in more detail.

Compilation. The original program was compiled and executed by the user. We can expect a smaller variant to be compiled using the same compiler invocation as the original program. A successful compilation is the first step towards a valid reduced program. The compilation will rule out most syntactical errors. It does not, however, catch semantically invalid variants. Programs containing infinite loops and wrong control flow might pass the compilation just fine.

Static analysis. The compilation above has its pitfalls. Invalid programs might be compiled without any issues. These programs would then cause issues when executing them: memory leaks, infinite loops, and pointless execution flow. By utilizing static analyzers, we increase our chances of catching these errors before executing the variant. This step feeds the variant's source code into a static analyzer. The analyzer might then return warnings or errors concerning the source code. Fatal errors are ruled out immediately. Other severe issues should be handled using hand-written rules.

Execution. Once a variant is compiled and free of obvious bugs, it is launched in a debugging environment. During its execution, we are checking the program for any runtime errors. Once an error occurs, we observe its location. This check can be done using the messages shown by the debugger or by analyzing the top stack frame.

Depending on the input program's execution time, the verification might take more time than the already long variant generating step. Future work might tackle this issue and improve the validation's performance.

6. Implementation

TODO: Mention that the location in implementation does not use columns, since the presumed location in LibTooling is not precise.

The problem analysis in Section 5 established several approaches to program reduction and minimization. The next step is to compare the described algorithms and techniques. Nevertheless, first, we must settle on their concrete implementations. More straightforward approaches such as the naive reduction and the minimizing Delta debugging algorithm deserve their implementation. However, implementing more complicated techniques - static and dynamic slicing - is simply out of scope for this project. This chapter will explain the structure of the project and the implementation process of the naive reduction and the minimizing Delta debugging algorithm. Moreover, it will introduce the reader to all used technologies and existing implementations. Each implementation will be described in a high-level overview while highlighting the necessary steps for its inclusion in this project. The chapter concludes with a discussion on some of the design choices and the project's limitations.

6.1 Technologies

This project heavily depends on existing compiler and debugger frameworks. Section 3 has compared these frameworks and chose the ideal candidate for the project. That ideal candidate is Clang's LibTooling. Section 4 described in detail the choice of AST as the de facto representation of this project's input. LibTooling offers multiple ways of traversing and modifying the AST. Additionally, it does so for both C and C++.

LibTooling can be built from the LLVM repository together with Clang. The required versions of Clang (11.0.0) and LibTooling are built from LLVM version 11.0.0. Building LLVM from source is a time-consuming process that does not always end in the desired result. The user must specify all required projects in advance using CMake's options. LLVM and its projects are then built using a different build tool such as ninja or make. Even though the building process can run multiple jobs at once, it can still take up to several hours, consuming a significant amount of the system's memory. The debug build utilizes tens of gigabytes of disk space. Thankfully, debugging symbols are not required for this project.

Clang is not the only LLVM project required as a prerequisite. Section 5.4 described the steps of naive validation. The final step requires a sort of execution environment. One proposed environment is a debugger. In order to keep the setup process as simple as possible, we settled on LLDB. LLDB is an open-source debugger that ships as a part of the LLVM project. The user needs to build LLDB with its scripting bridge API. This can be achieved by adding LLDB to the LLVM project list when invoking CMake. The Python API and its C++ scripting bridge can also be included by specifying a few other arguments. By default, LLVM builds for all available platforms, including ARM and PowerPC. However, only a single platform is required/supported for this project. The target platform with which LLVM should be built is x86 64 bits.

LibTooling changes with every release. Projects dependent on an older version of LibTooling might not work with a newer one. Moreover, older releases of LLVM cannot always be built on new platforms. From experience, the issue might arise when an old LLVM version attempts to link new system headers and libraries. An easy and reliable way of preserving older LibTooling environments is by storing them in a Docker container.

Docker is another dependency of this project. It is required to run slicing implementations as well as support the entire minimization process on Windows.

6.2 Design

Since this project compares several techniques, some of which are composed in a pipeline, we decided on using a modular design. Each reduction approach described in Section 5 consists of one or more components.

These components can be shared and reused across multiple minimizing algorithms. This choice reduces the amount of copy-pasted code and increases its maintainability in the future. Depending on the technologies used in each approach, the components form either a C++ application or a Python script. The latter is required for executing slicer implementations. This fact leads to inconsistent interface across multiple reduction approaches. For example, the user can launch the minimizing Delta debugging algorithm as a C++ console application. However, if he decides to execute the slicing-based approach, he will need to launch it as a Python script.

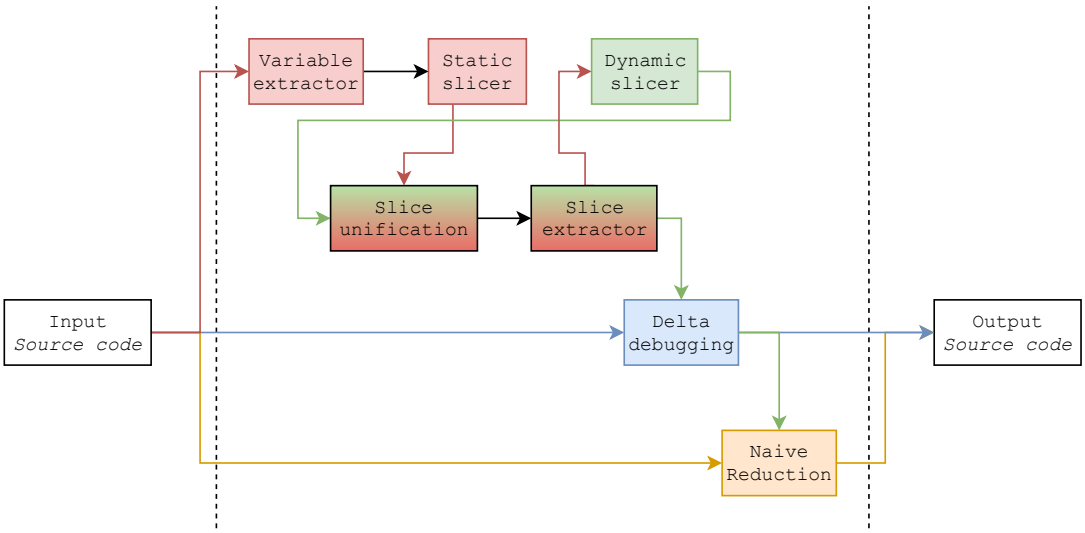

Figure 6.1: Communication between modules in three different pipelines: Naive (orange), Delta (blue), Slicing (red and green).

Figure 6.1 shows a diagram of all available pipelines and their shared modules. Since some pipelines only add a preprocessing layer, they end up reusing the fundamental components of simpler pipelines. Each module shown in the diagram will be described in the following sections.

6.3 Shared components

As was already mentioned, this project reuses several ideas and components in multiple approaches. One component does not necessarily translate to one idea. It often takes a couple of modules to capture that idea. The following subsections will describe each idea proposed in previous chapters. Each subsection will then outline components used by its particular idea.

Recursive visitor

Some of the ideas in the following sections require us to traverse and alter the AST. As explained in Section 4.2, LibTooling allows us to traverse the AST using two methods. Splitting the code into code units could be done with either the `RecursiveASTVisitor` or `ASTMatchers`. We chose to implement the visitor interface to preserve more consistency across the project. Using a LibTooling visitor requires implementing three classes. Types of their implementation are described in detail in the following paragraphs.

Actions. Each derivation of `ASTFrontendAction` can have its own preprocessing and postprocessing steps. Other than performing an action before and after a file is handled, it also creates a specific visitor. In this project, we only utilize `ASTFrontendAction` to create a consumer. Moreover, the usage of actions tends to be module-specific. Usually, each component has its *Actions.cpp* and *Actions.h* files. Further, some of the derivations use their custom factories. By default, any `ASTFrontendAction` can be created by calling the `FrontendActionFactory::create()` method. The method uses the default constructor and thus cannot provide any arguments for concrete `ASTFrontendAction` implementations. A workaround can be seen in *Actions.cpp* files, most of which contain a custom factory. The factory takes the necessary parameters, calls the desired constructor, and passes them to a `Consumer` instance in its `create()` method.

Consumers. This project distinguishes two types of `Consumer` implementations. The first is a module-specific type. These `Consumer` implementations contain high-level actions of a specific algorithm and are used for data transfer. For example, the `VariantGeneratingConsumer` does not invoke any visitor instances. Instead, it keeps references to two more granular `Consumer` objects. The `VariantGeneratingConsumer` contains the main loop of the naive algorithm shown in Figure 5.1. The pair of more granular consumers then carries out specific actions. The module-specific consumer also handles data exchanges between its more specific workers. One way of doing these exchanges is by keeping data structure instances inside the parent consumer. Another is by accessing the worker's internal states, either through their fields or getter methods. In summary, the module-specific consumer's job is to organize the granular parts of an algorithm, execute their actions, and collect their results.

The second `Consumer` type is the shared consumer. These consumers are used for dispatching shared visitors, reporting their state, and collecting their data.

The shared consumer responsible for code unit separation is the `DependencyMappingASTConsumer`. It dispatches a concrete visitor to the top-most node and offers getter methods to retrieve the visitor's data. This consumer also contains some logic in its `HandleTranslationUnit` method. Other than dispatching the visitor, it also saves a visualization of the current AST. Other shared consumers might contain different logic in that method.

Visitors. Visitors are once again separated into two types. A specific algorithm might use its unique visitor implementation to do an action such as slice extraction. This type of visitor is tied to a corresponding module-specific consumer. Similarly, shared visitors are also bound to their corresponding shared consumers. However, shared visitors are used for tasks that are used in both the naive and Delta algorithms. Those tasks are the node mapping and the variant generation. In our case, visitors contain the logic of a single iteration. Be it an iteration of the naive minimization or the Delta debugging; visitors are dispatched polynomially or even exponentially many times. The logic generally contains rules for handling each type of language construct. Two examples of such rules are:

- Skipping the traversal of an implicit cast node of type `ImplicitCastExpr`, since removing it will not alter the source code.
- Creating a code unit for each statement that is not an expression.

The logic of a shared visitor is reused in multiple projects. This logic might include splitting the source code into code units. A module-specific visitor's logic is oriented to a particular problem, such as scanning the locations of statements in the code.

6.3.1 Code units

Initially, a code unit was defined as any syntactically correct subset of the source code. It was then narrowed down to an atomic code unit - the smallest sensible syntactically correct source code element. Both the naive and the Delta algorithms use code units to perform their reductions. The goal is to split a given source code into partitions that can be later kept or removed. Those partitions would ideally not overlap and could be therefore independently removed.

We can split the input source code into code units by traversing its AST with a specified granularity. The idea is to analyze nodes based on hand-written rules. These rules specify which nodes are worth visiting and how to process them. While we are partitioning the source code, we might as well map the code's dependencies. This mapping can be done by extending our rules to perform specific lookup operations during the traversal. For example, the algorithm looks up all variable usages for a given variable declaration node. The results of those lookups are then kept in the dependency graph described in Section 5.1.1.

Consumers. The single consumer dedicated to code unit separation is implemented in the `DependencyMappingASTConsumer` class. The consumer serves as a middle man for data transfers. Notably, it can retrieve data for the following structures:

- Node mapping - this mapping is a lookup table that transitions between a node's internal ID given by Clang and the current traversal order number. The traversal order number specifies the order in which the current node is visited once the visitor is dispatched using `HandleTranslationUnit`. The mapping is later transferred to other visitors to achieve the same traversal order.
- Code units count - this property returns the total amount of code units once their partitioning is complete.
- Dependency graph - is represented by a directional graph of nodes, where each node is represented using its traversal order number. Nodes in the graph might also contain additional information used for debugging, such as their type and the code snippet they represent.
- Skipped nodes - this container serves as a list of nodes not worth visiting. These nodes might be too granular or solely implicit.

The consumer dispatches its concrete visitor, which carries most of the logic. The only standalone logic in the consumer is to dump a visualization of the source code based on the dependency graph.

Visitors. What becomes a code unit is decided inside the `MappingASTVisitor` class. The visitor traverses the AST in a postorder manner and stops at specified declarations, statements, and expressions. Each of these nodes must fulfill a predicate - their specific rule - to become a code unit. The node must not be a duplicate of an already visited node. It must also be a part of the main source file, i.e., not a part of a system header file. If the node satisfies all three requirements, it is considered a code unit. The node's ID mapping is then created, and the node is added to the dependency graph. On the other hand, if the node is not a valid code unit, it is added to the skipped nodes list. In that case, the next visitor will not consider visiting and manipulating the node when traversing the AST.

6.3.2 Bitfield to variant transformation

Once the number of code units is known, we can create a bitfield of that size and generate representations of possible variants. The mechanism of generating bitfields was explained in Section 5.1. This section describes the necessary parts of a technique that converts bitfields into source code variants. The idea is to flag AST nodes as code units and then traverse the AST again, removing the code unit nodes specified by a given bitfield. A single iteration of the variant-generating techniques produces one source code variant. The variant is created by starting with the original source code and gradually removing nodes. Each node has its index in the given bitfield. The index corresponds to the node's traversal order number. A node is removed if its corresponding bit is set to zero. To be more precise: the node is not removed from the AST. Instead, only its underlying source code is removed. The following paragraphs describe the code behind the removal.

Consumers. All the necessary data for bitfield to source code conversion is acquired using the `VariantPrintingASTConsumer`. This consumer dispatches its specific visitor and provides it with a bitfield. The visitor then removes snippets of the code based on that bitfield. The consumer's job is to reset the visitor's state, launch it with the provided bitfield, and save its output into a specified file. The consumer can also access the visitor's internal state, namely its adjusted error line location. More details about this location can be found in the *Visitors* section.

Visitors. Similar to the code unit mapping issue, the `VariantPrintingASTVisitor` carries most of the conversion's logic. It is provided with a bitfield, which it saves, and a `Rewriter` instance, which it uses to generate the variant. Section 4.5 describes the `Rewriter` class in more detail. Traversal of the source code is done in a postorder fashion to stay consistent with the previous visitor. The previous visitor shares its dependency graph and its list of skipped nodes. This way, the `VariantPrintingASTVisitor` can traverse the AST the way it was intended to.

Each code unit, i.e., visited node, is removed if it fulfills specific requirements. First, the node's bit must be set to zero. Second, all its parent's bits must be set to one. The latter is a `Rewriter` requirement. A parent node might represent a compound statement, and its children might be the individual statements inside the parent. If we were to remove the parent and one of its children, we would remove the same snippet twice. This catch would eventually lead to an error. The error can be avoided by only removing children while keeping their parents.

Removing code units tends to shift lines down. This phenomenon might lead to the statements on the original error line changing their location. The number of the error-inducing line must be adjusted. During the removal of each node, we extract its underlying code and analyze its presumed location. If the location comes before the original error-inducing line and the underlying code contains newline characters, we decrease the error-inducing line's number by the amount of found newline characters. At the end of the traversal, we are left with a correct line number for further validation, and all unwanted code units are removed. We are left with a `Rewriter` instance whose buffer reflects the current variant. The visitor's iteration is done, and it is the consumer's job to save the contents of the buffer.

6.3.3 Variant validation

Testing results on whether they are valid variants requires a standard interface, too. Section 5.4 describes the steps in the process of validation. The implementation contains three parts that are used in all approaches presented in this project. Below is the description of compilation, analysis, and execution.

Compilation. By calling the `Compile` function in *Helper.cpp*, one can invoke the Clang compiler driver. The compiler has two goals. Firstly, it filters out non-compilable and thus invalid variants. Secondly, it prepares compilable variants for the execution stage. The compilation can be invoked with a wide range of

arguments. In this case, it is provided with the `-g` and `-O0` options. The former generates debug symbols for the executable, while the latter ensures reliable debugging by eliminating any compiler optimizations. Compilation's output is printed to the standard output, and its exit status determines the function's return value. If the compiler terminates with a valid exit code but does not create the binary, the function returns as if the compilation failed. The binary is stored to a specified path, which by default is the same file path as the input source file. The file extension is substituted with `.exe`.

Static analysis.

TODO: Research and implement calls to the Clang static analyzer.

Execution. Compiled binaries need to be validated at runtime. This way, we check whether the program results in the desired runtime error. Programs are executed in the LLDB environment. LLDB provides Python API, which allows invoking more or less all of the debugger's commands. The API is also available from C++ using a scripting bridge. SWIG processes function calls made from C++. They then produce bindings to the Python API. Thanks to the scripting bridge, every validation step is written in C++. The `ValidateResults` function creates a debugging environment for every executable.

The programs are then run in separate processes. During the execution, events are broadcasted from the forked processes. The stack trace is investigated whenever the program broadcasts a stopped state, indicating a thrown exception. If the symbol's location on top of the stack trace is the same as the one of the desired error, the program is tagged as valid. Otherwise, the execution continues.

6.4 Naive reduction

The algorithm for naive minimization is the foundation of this project. The approach and its heuristics were described in detail in Section 5.1. The naive algorithm can guarantee minimality. Unfortunately, the same cannot be said for greedy approaches. Since this project focuses heavily on minimization, we dedicated a significant amount of time to improving the naive approach's implementation.

The reader has already seen most of this algorithm's implementation in Section 6.3. Nearly all shared components were created during the development of the naive approach. The implementation closely follows the algorithm shown in Figure 5.2. As such, it required us to implement several vital mechanisms. First of all, the algorithm works with code units. As such, we had to create a way of splitting the code into several partitions. This lead to the implementation of a shared module that handles code units. That module has already been described in Section 6.3.1. Second, we needed to convert bitfield representations into source code variants. This task required the development of a shared component that removes AST nodes based on a given bitfield. Details of this component can be found in Section 6.3.2. The algorithm's last major requirement was the means to validate a variant. The implemented variant verification follows the steps described in Section 5.4. Its implementation details as a shared module can be found

in Section 6.3.3. The parts described above are not specific to the naive approach. Instead, they are used in the implementation of every suggested approach.

However, the naive algorithm also uses its specific classes and functions. Two distinguishing factors are the bitfield manipulation and the validation of the dependency graph.

The naive algorithm works by iterating over all possible bitfield variants. Typically, the bitfield would be represented using an unsigned binary number. To make working with the bitfield easier, we opted to represent it using a `std::vector<bool>`. The vector is a good choice since its boolean variant uses a memory-efficient implementation in which each element only takes up a single bit. Working with the vector is also significantly more straightforward than with other representations. Possible bitfield variants are created by continuously incrementing the bitfield.

However, the vector representation lacks the increment operation. We implemented it by flipping the first bit of the vector and keeping note of the carry-over bit. The implemented function's name is simply `Increment`. As long as there is a carry-over bit, we keep flipping the consecutive indexes of the vector. Incrementing a bitfield where all bits are set to one leads to an overflow. While generating variants, we can prevent the overflow from happening by checking the `IsFull` function. The function decides whether a given bitfield has all its bits set to one. The last used bitfield function is the `IsValid` check. It determines whether a given bitfield might correspond to a valid source code variant.

This function is also the first to utilize the dependency graph for its heuristic purposes. The function iterates over all bits in the bitfield. Each bit has to fulfill the following requirements. First, if the bit is set to zero, so must all of its children. This rule translates to ruling out both syntactically and semantically invalid variants. Second, if the bit represents a node in the criterion, i.e., the error-inducing location, it cannot be removed. This requirement preserves the failure-inducing line, ruling out pointless variants.

The dependency graph validation is straightforward. Each node in the graph is represented using its traversal order number. The exact number corresponds to the node's index in the bitfield. When validating all children of a particular node, we simply traverse a container of indices, checking whether the bitfield contains the desired bit value on each index. The `IsValid` function also calculates the size metric suggested in Section 5.1.1. Each node has its presumed size in characters. The total size of the variant is calculated during the dependency graph validation. This way, we get an approximation of how large the source code variant is before generating it. The function returns a ratio of the size of the variant compared to the original size.

This ratio serves two purposes. Firstly, the user can set their desired maximum ratio. If the user assumes the minimal variant is at most a quarter of the original size, they can set their target ratio to 0.25. This assumption reduces the search space to only those variants whose presumed size is at most a quarter of the original size. Secondly, the implementation uses a sort of iterative deepening search. For a given number k, all valid bitfield variants are separated into k bins. These bins represent parts of the interval from zero to the target ratio set by the user. The interval gets divided into k evenly-sized parts, which then dictate the order in which the algorithm searches. First, the algorithm generates and

validates all variants in the bin that contains the presumably smallest variants. If no valid result is found, the search continues with the bin containing the next smallest variants. Ideally, the search finishes faster than usual. This case would save time otherwise spent on generating all variants. In the worst case, the algorithm will generate and validate all possible variants, just like it did originally. Figure 6.2 shows the pseudocode of the bin heuristic.

Input:
B ... a set of bins - containers of bitfield.
L ... location of the error.
A ... the input program's arguments.
Output: The reduced source code.

```
1: for all Bin ∈ B do
2:     for all Variant ∈ B do
3:         if IsValid(Variant, L, A) then return V.
4:         end if
5:     end for
6: end for
```

Figure 6.2: Binning search.

In the ideal world, we would know the exact size of a variant ahead of time. Knowing this, we could sort the variants before generating them, resulting in the best possible outcome. However, the presumed variant size is just an approximation. Clang's information on the underlying source code is not perfect. Using bins decreases the chances of classifying a non-minimal variant as the optimal result.

All the mentioned functions can be in the *Common/src/Helper.cpp* file. The dependency graph resides in its file, *Common/include/DependencyGraph.h*.

The mentioned functions are called from the algorithm's main loop. In our implementation, the loop is in the `HandleTranslationUnit` function of the `VariantGeneratingConsumer` class. This consumer is specific to the naive reduction problem. It does not dispatch any visitors; it only governs two other consumer objects. Those objects are the instances of the `DependencyMappingASTConsumer` and the `VariantPrintingASTConsumer` classes. The former partitions the input into code units, while the latter converts bitfields into source code variants. The body of `HandleTranslationUnit` manages all data exchanges between the two consumers. It also contains the bitfield iteration logic. The consumer handles generating the variants, while their validation is invoked from the program's `main` function.

The consumer can be constructed using a custom factory. The `VariantGeneratingFrontendActionFactory` function takes a reference to a `GlobalContext` object and later passes it to the visitor. The context is used for looking up the original input and the current state of the algorithm.

The problem-specific consumer can be found in the *NaiveReduction/src/Consumers.cpp* file. The action that invokes the consumer is in *NaiveReduction/src/Actions.cpp* and *NaiveReduction/include/Actions.h*. The `GlobalContext` resides in its own file under *Common/include/Context.h*.

6.5 Delta debugging

Implementing the minimizing Delta debugging algorithm took significantly less time than expected. The implementation of the naive approach laid down a good enough foundation to make implementing Delta easy. The algorithm reuses existing components for code unit partitioning, bitfield to source code conversion, and validation. This section describes how it does so.

The minimizing Delta debugging algorithm is typically implemented using a text-modifying approach. The input test case could be split into n partitions based on many factors. One implementation might split the input on an exact number of characters, while another might do so on a specific number of lines. Text-oriented Delta debugging is not the only way of approaching Delta's implementation. Hierarchical Delta debugging (HDD) presented by Misherghi and Su [22] handles the minimization using an AST. Implementing HDD would be ideal for this project since it we are working with the AST already. However, due to work already done on the naive approach in Section 6.4, we chose to implement a modified version of the minimizing Delta algorithm.

Our implementation performs the following steps during each iteration:

1. Input source code is partitioned into code units in the same manner as in the naive approach, using the shared component described in Section 6.3.1.

2. Bitfields are created in order to represent possible Delta partitions and their complements. The Delta algorithm splits the current test case into n partitions and their complements in each iteration. This split translates to code units being assigned into specific variants. Bitfields corresponds to one of the $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$ variants. The bitfield's elements are set to one if they represent snippets in their variant.

3. The bitfields are iterated over according to the algorithm in Figure 2.1. Each bitfield is converted into source code using the component shown in Section 6.3.2. The code is then validated. If it causes the desired runtime error, it is regarded as the test case for the next iteration. The validation uses the module described in Section 6.3.3.

The described approach is nearly equivalent to removing lines of the source code. However, it does leave some room for structure-based heuristics for future work.

The algorithm's main loop is inside the program's `main` function. Each iteration invokes a custom visitor that handles it. Clang parses the current test case, and a `DeltaDebuggingConsumer` instance performs its logic on the generated AST. The logic consists of the three steps shown in the previous paragraph: partitioning, generating variants, and validating. The iteration's result is then passed by reference back to the `main` function.

Similar to implementing the naive approach, the problem-specific consumer is invoked from a custom `ASTFrontendAction`. The `DeltaDebuggingFrontendActionFactory` function supplies data to the consumer. The data contains the following:

- Iteration number - this number is used for correct file naming.

- Partition count - this number represents the number of even-sized partitions into which the input should be split.
- Iteration result - the result is passed as a reference to an `enum`, which the consumer then specifies.
- Global context - analogically to the naive reduction's implementation, the context serves to lookup information concerning the algorithm's input and state.

The mentioned classes can be found in the following files:

- *DeltaReduction/include/Consumers.h*
- *DeltaReduction/include/Actions.h*
- *DeltaReduction/src/Actions.cpp*
- *Common/include/Context.h*

6.6 External code

As was already mentioned at the beginning of this chapter, implementing a slicer is out of this project's scope. We have turned to existing implementations of static and dynamic slicers. We know that these implementations will be more reliable and accurate. Below are the details concerning the two chosen slicer implementations. We were searching for LLVM-based slicers in the hope of running them natively during the reduction. However, due to compatibility reasons, the slicers cannot be easily shipped and executed with the current implementation of the rest of the project. Therefore, we run both slicers in their available Docker containers.

DG. We chose DG as it is the best LLVM-based static slicer found. It slices LLVM bitcode and therefore requires the input in the compiled form. This requirement imposes the first obstacle - compiling the given source file to the LLVM intermediate representation. This task is not difficult for programs with a simple compilation command; however, handling large projects might be difficult. DG supports multiple criteria. One can slice w.r.t. either a function call or a variable on a given line. The variable criterion only works correctly for bitcode compiled with debugging symbols. DG also handles secondary slicing criteria, which are not required for this project.

The output of DG is the sliced bitcode. The slice can be analyzed either by disassembling the bitcode or converting it to a list of line numbers using provided scripts. Nonetheless, we end up with a usable slice representation.

It should be noted that DG does not support C++ by design. Slicing its bitcode might not be an issue. However, there is a chance of encountering unsupported instruction.

DG is available on GitHub. The repository can be found at `https://github.com/mchalupa/dg`.

Giri. Giri is an LLVM-based backward dynamic slicer. The project started its development during the Google Summer of Code in 2013, and it has not received many updates since. It runs on a deprecated version of LLVM that cannot be installed on modern systems. In this project, we attempted to port Giri to a newer version of LLVM but ultimately failed. While most changes to Giri would have been trivial, others required much effort in reimplementing legacy constructs. Our update was therefore dropped.

One can use Giri via a `Makefile`. This unconventional approach hides the standard interface of the slicer. Instead, the user specifies their desired criterion in a text file. Criteria must specify a path to the sliced source file and a line number. The text file is then passed to the `make` command, and a prepared `Makefile` is executed. The user can also specify a concrete instruction's number instead of its line number. However, this feature is not used in our project.

Similar to DG, Giri also produces a list of line numbers. This slice representation then has to be extracted from the original source code.

Giri's repository can be found on GitHub on the following address: `https://github.com/liuml07/giri`.

6.7 Systematic approach

The systematic approach is a pipeline comprised of multiple steps. The steps are represented by components, some of which are shared with the previous implementation. The entire pipeline consists of a variable extraction tool, static and dynamic slicers, a slice unification tool, a slice extraction tool, the minimizing Delta debugging algorithm, and the naive approach. The motivation behind these steps and an overview of the algorithm can be found in Section 5.3.

This approach uses external code that cannot be trivially added to the project. Therefore, it is launched and operated differently. The external code forms two modules - containers. The static slicer is accessed via the DG container and the dynamic slicer via the Giri container. The unifying base is a Python script that invokes all necessary components. The script uses Docker API to launch the DG container. It also maps input and output directories to that container in order to send and retrieve data. The container's launch command invokes the slicer to process the given input and store it in the given output directory. The Giri container is launched analogically.

Both slicers produce a list of lines that represent the slices. The slicers are also run repeatedly, once for each variable on a given line. These variables are extracted using a `VariableExtractor` tool. This module uses LibTooling's AST-Matchers to search for variable usages on a given line. The names of the symbols are then dumped into an output file. Slicers are executed with different criteria, and the criteria are taken from that file. These runs generate multiple slices, which must be unified into a single one. The unification is simple. The slice is represented by a file containing a list of line numbers. The multiple generated files are merged, removing any duplicates. The result is a single file containing a single enhanced slice.

This output needs to be processed further. The base script invokes the `SliceExtractor` component. This module uses LibTooling's recursive AST visitor to access statements on given lines and preserve them. The program trans-

forms a source file into the desired slice based on the given list of lines. First, the tool creates a container of preserved lines. Initially, the container consists of the lines given through the input. Later, the container expands by adding other previously poorly-mapped lines. For example, if a `for` loop stretches over several lines and the slicer result only contains its first line, we must include those other lines as well. We do this by traversing the AST and inspecting nodes on given lines. If we encounter a node whose underlying code stretches over several lines, we include those lines in the container. The input source file is then opened and read, and the lines present in the container are preserved.

Once the `SliceExtractor` produces the desired source file, the file is considered the respective slicer's output. This way, each step of the algorithm results in a valid source file.

Once the slicer preprocessing is done, the pipeline executes the implemented Delta debugging algorithm and finds a local minimum. The output is a source file which then serves as the input for the naive approach. The Python script executes the naive algorithm, which then produces the desired results. The implementation of both the naive reduction and the minimizing Delta debugging algorithm can be found in Section 6.4 and Section 6.5, respectively.

The classes and functions used for implementing the `VariableExtractor` and the `SliceExtractor` modules can be found in their respective directories:

- *VariableExtractor/include*
- *VariableExtractor/src*
- *SliceExtractor/include*
- *SliceExtractor/src*

The listed directories contain the problem-specific code. However, the tools also utilize code from shared components to carry out some repetitive tasks.

7. Evaluation

Having described the implementation of proposed reduction approaches, we can finally compare them. The goal of the comparison is to show that the slicing-based approach is the most practical minimization algorithm.

7.1 Metrics

There are several points of interest in this comparison. Each presented algorithm contained a description of this time complexity and its minimization properties. Due to this fact, we believe both the level of reduction and the algorithm's efficiency should be measured. In order to capture the performance of each approach, we employed the following metrics.

- This project focuses on minimality. We want to test whether an algorithm produces the optimal result. The *minimality* metric is measured in two values: true and false. True corresponds to the result being the desired minimal variant, while false means that the result is suboptimal. We expect the naive and slicing-based algorithms to acquire significantly more minimality than the Delta debugging approach.

- We have noted that some approaches, such as the minimizing Delta debugging algorithm, do not achieve optimal reduction. However, we can measure the ratio of the generated result compared to the minimal variant. The *minimization ratio* will be measured in percentage. The goal is to achieve 0%, which translates to the result being minimal.

- Approaches can be compared against each other using a *proportion ratio*. This metric is also measured in percentage. The number between 0% and 100% can be interpreted as the result's proportion of the original program's size. The lower the proportion ratio is, the better the algorithm is at source code reduction.

- A straightforward way of measuring a program's performance is by watching its *execution time*. The time will be measured in seconds and will serve as the primary indicator of each algorithm's performance.

- The metric for testing heuristics is the *processed variants* count. By counting how many actual results had to be generated and validated before settling on the final output, we can evaluate the effectiveness of a heuristic. The number of processed variants is compared with the expected number of variants, i.e., the worst-case scenario.

By measuring these properties, we also observe whether the presented approaches struggle with a given input. Poor handling of specific inputs could also boil down to lacking implementation. However, we assume that errors in the implementation can be spotted in three ways. The program either throws an exception, produces an unreduced source code as its result, or outputs a program that does not result in the desired runtime error.

7.2 Data set

Each implemented approach works with three major input parts:

- It receives the source code it should minimize.
- It is given a location of the desired runtime error.
- It requires a set of arguments used when running the input source code, which leads to the mentioned runtime error.

Some approaches might benefit from other user inputs. For example, the implementation of the naive algorithm can cut the search short if it exceeds a given amount of variants. These inputs will be referred to as minor arguments.

Our dataset consists of 30 simple programs. Data for each program includes its source code written in C or C++, the target location, its execution arguments, and each approach's minor arguments. The dataset is made up of three equally-sized parts. Those parts differ based on the techniques used in their entries' source code. The first ten data entries represent non-structured programs. These entries contain source code exclusively in their `main` function. Generally, they do not follow any specific programming paradigms and represent the simplest input type. The second part contains ten structured programs. The source code of these programs uses control flow statements, functions, and procedures. This type of input represents the regular program that this project is expected to handle. The last ten programs use aspects of object-oriented programming. Their code contains classes, inheritance, and polymorphism. Furthermore, the code is almost exclusively written in C++. However, advanced features of the language, such as templates, are omitted.

All three parts of the dataset are similar in terms of size. The source code for each program ranges from 30 to 100 lines. Moreover, the source code is contained in a single file. Therefore, the programs do not use any other include headers outside of system headers and standard libraries. Runtime errors in the dataset are caused mainly by invaliding an assertion. However, segmentation faults make up a large number of errors as well. We have also limited each program to cause only a single runtime error initially.

7.3 Results

We tested a total of three approaches. These include the naive approach with its heuristics, the minimizing Delta debugging approach, and the slicer-based approach with argument injection. The dataset was executed on a 6C/12T AMD Ryzen 5 5600X processor with 8GB of memory. Specifically, the chip was clocked at 4.85GHz for single-threaded tasks and 4.5GHz for multi-threaded workloads. The naive approach ran parallelized on 12 threads, while other approaches ran single-threaded. The project was compiled using Clang 11.0.0 with high optimizations (`-O2`) and ran on Ubuntu 20.04.2.0 LTS.

Table 7.1 contains the results of all executed benchmarks. Each row represents a single entry from the data set. The entry has been used three times: in the naive approach, the Delta debugging algorithm, and the slicing-based approach.

Those 30 rows thus represent a matrix of all executed runs and their results. Metrics introduced in Section 7.1 are denoted as follows:

- The *minimization ratio* is represented by the $\mathcal{R}_m$ column. Values are in percentages.
- The binary *minimality* metric does not have its column. However, it can be derived from the $\mathcal{R}_m$ column by looking for rows in which the value of $\mathcal{R}_m$ is zero.
- The *proportion ratio* is represented by the $\mathcal{R}_p$ column. Values are in percentages and correspond to the ratio of input and output sizes in bytes.
- Run's *execution time* can be found in the Δ_t column. Time is measured in the most relevant unit based on the run's duration.
- The *processed variants count* is in the $\mathcal{I}_a$ column. It can be directly compared to the $\mathcal{I}_e$ column. $\mathcal{I}_e$ represents the expected number of processed variants - the worst-case scenario.

		Naive				Delta				Slicing		
$Test$	$\mathcal{I}_e$	$\mathcal{I}_a$	$\mathcal{R}_m$	$\mathcal{R}_p$	Δ_t	$\mathcal{I}_a$	$\mathcal{R}_m$	$\mathcal{R}_p$	Δ_t	$\mathcal{I}_a$	$\mathcal{R}_m$	$\mathcal{R}_p$
1	2^{14}	15	0.0	0.0	0.581s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	2^{19}	567	7.3	0.0	2.448s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	2^{10}	1186	0.0	0.0	0.715s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	2^{26}	761	0.0	0.0	1 m34s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	2^{15}	615	0.0	0.0	1.364s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
6	2^{11}	4	0.0	0.0	0.519s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	2^{31}	915	16.7	0.0	29 m7s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
8	2^{40}	465	4.2	0.0	47 m59s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
9	2^{13}	17	0.0	0.0	1.848s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10	2^{16}	61	0.0	0.0	8.155s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
11	2^{18}	37	0.0	0.0	55.168s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
12	2^{22}	18	7.1	0.0	2 m14s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
13	2^{14}	138	0.0	0.0	0.917s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14	2^{16}	84	0.0	0.0	16.189s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
15	2^{10}	9	0.0	0.0	0.534s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
16	2^{13}	14	0.0	0.0	0.783s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
17	2^{11}	31	0.0	0.0	0.634s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
18	2^{34}	2156	19.9	0.0	38 m13s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
19	2^{27}	1654	14.0	0.0	14 m6s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
20	2^{14}	276	0.0	0.0	1.003s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
21	2^{22}	315	0.0	0.0	2 m35s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
22	2^{24}	546	0.0	0.0	11 m52s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
23	2^{19}	34	33.3	0.0	1 m2s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
24	2^{18}	12	0.0	0.0	1 m33s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
25	2^{24}	463	5.8	0.0	5 m9s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
26	2^{20}	154	0.0	0.0	3 m25s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
27	2^{15}	93	0.0	0.0	2.018s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
28	2^{21}	75	0.0	0.0	2 m15s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
29	2^{14}	140	21.8	0.0	0.882s	0.0	0.0	0.0	0.0	0.0	0.0	0.0
30	2^{16}	168	0.0	0.0	4.333s	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 7.1: A benchmark of the three minimization approaches.

Conclusion

Bibliography

[1] A. Zeller. Yesterday, my program worked. Today, it does not. Why? *LNCS*, 1687:253–267, 1999.

[2] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[3] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[4] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[5] B. Korel and J. Laski. Dynamic program slicing. *Inform. Process.*, Letters 29(3):155–163, 1988.

[6] GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`. [Online; accessed 14-March-2021].

[7] The LLVM Compiler Infrastructure Project. `https://llvm.org/`. [Online; accessed 14-March-2021].

[8] Clang: a C language family frontend for LLVM. `https://clang.llvm.org/`. [Online; accessed 14-March-2021].

[9] About The ANTLR Parser Generator. `https://www.antlr.org/about.html`. [Online; accessed 14-March-2021].

[10] Semantic Designs. Dms® software reengineering toolkit. `http://www.semdesigns.com/Products/DMS/DMSToolkit.html`. [Online; accessed 14-March-2021].

[11] LibTooling - Clang 12 documentation. `https://clang.llvm.org/docs/LibTooling.html`. [Online; accessed 15-March-2021].

[12] Eli Bendersky. Compilation databases for Clang-based tools. `https://eli.thegreenplace.net/2014/05/21/compilation-databases-for-clang-based-tools`. [Online; accessed 15-March-2021].

[13] Introduction to the Clang AST - Clang 12 documentation. `https://clang.llvm.org/docs/IntroductionToTheClangAST.html`. [Online; accessed 15-March-2021].

[14] RecursiveASTVisitor Class Template Reference. `https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html`. [Online; accessed 15-March-2021].

[15] Matching the Clang AST - Clang 12 documentation. `https://clang.llvm.org/docs/LibASTMatchers.html`. [Online; accessed 16-March-2021].

[16] AST Matcher Reference. https://clang.llvm.org/docs/LibASTMatchersReference.html. [Online; accessed 16-March-2021].

[17] Eli Bendersky. Modern source-to-source transformation with Clang and libTooling. https://eli.thegreenplace.net/2014/05/01/modern-source-to-source-transformation-with-clang-and-libtooling. [Online; accessed 16-March-2021].

[18] A. Zeller. Automated Debugging: Are We Close? *IEEE Computer*, 2001.

[19] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 01 1937.

[20] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[21] Rajiv Gupta, Mary Lou Soffa, and John Howard. Hybrid Slicing: Integrating Dynamic Information with Static Analysis. *ACM Trans. Softw. Eng. Methodol.*, 6(4):370–397, October 1997.

[22] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical Delta Debugging. page 142–151, 2006.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment