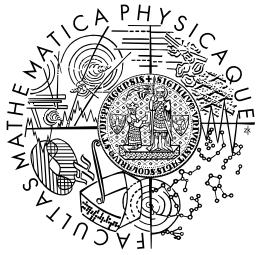

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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

TODO: Rewrite the introduction to include goals.

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 mostly manual chore. This is 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.

Let program $\mathcal{P}$ contain a runtime error E that consistently occurs when $\mathcal{P}$ is run with arguments A. Since the error E is present at runtime and not compile-time, it can be assumed that syntax wise the code is mostly correct. Therefore, any syntax-based error can be ruled out. This, in turn, leaves us with a set of semantic errors $\mathcal{E}_{\mathcal{S}}$. Those include wrongly indexed arrays and calculations that lead to either the incorrect result or an altered control flow of the program. Let $E \in \mathcal{E}_{\mathcal{S}}$. As the generality of errors in $\mathcal{E}_{\mathcal{S}}$ appears too complicated to be solved for all programming languages at once, it is necessary to break the problem down for each programming language. This project is concerned with the semantic errors of C and C++. Although C is not a subset of C++, the semantic errors made in C can be approached similarly to those in C++. Both languages share mostly comparable constructs. Finding the cause of a semantic error in a concrete language requires knowing these constructs, their behavior, and their general handling.

To make finding the cause of an error a systematic approach, one might try removing unnecessary statements in the code, thus minimizing the program. 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 A. If done carefully and correctly, $\mathcal{P}'$ represents the smallest subset of $\mathcal{P}$ regarding code size, while preserving the cause of the error in that subset. Upon manual inspection, the developer is required to make less of an effort to find the root cause in $\mathcal{P}'$ as opposed to $\mathcal{P}$.

The minimization of a program can be achieved in numerous ways. In further sections, the project describes and compares three different approaches. The first is based on naive statement removal and its consequences during runtime. The second removes major chunks of the code while periodically testing the generated program's correctness. The third deploys a sequence of code altering techniques, namely slicing and delta debugging.

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. $C = (write(x)_{42}, \{x\})$.

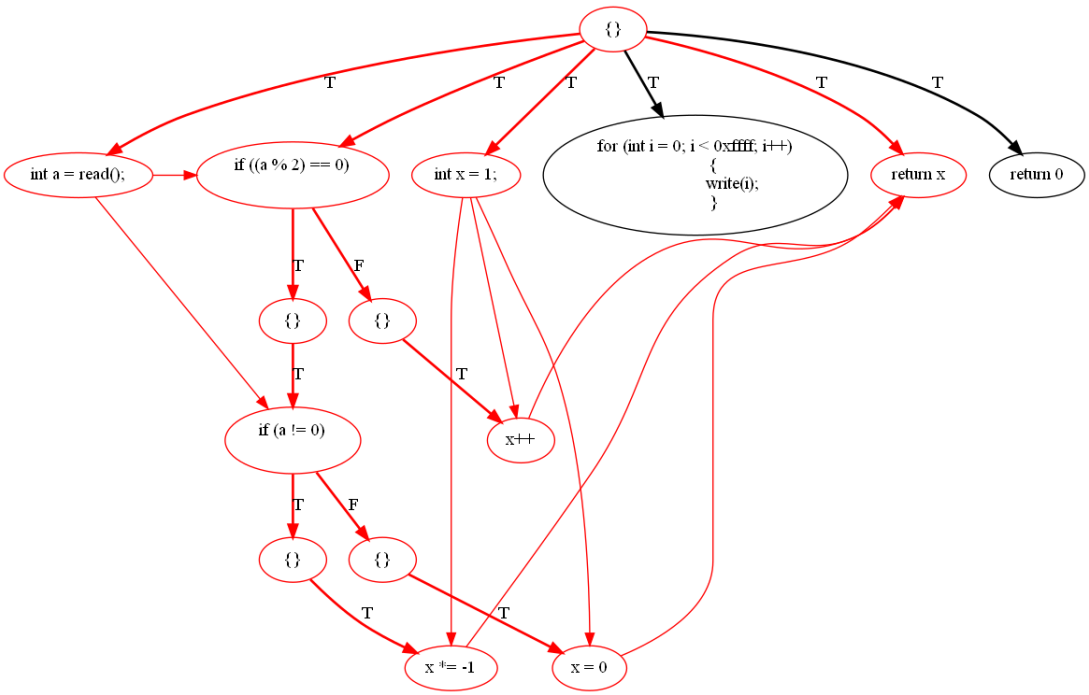

Figure 2.3: Sliced PDG. The graph was created from the source code shown in listing 2.1. Red edges indicate the sliced part of the program w.r.t. $C = (write(x)_{42}, \{x\})$.

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.

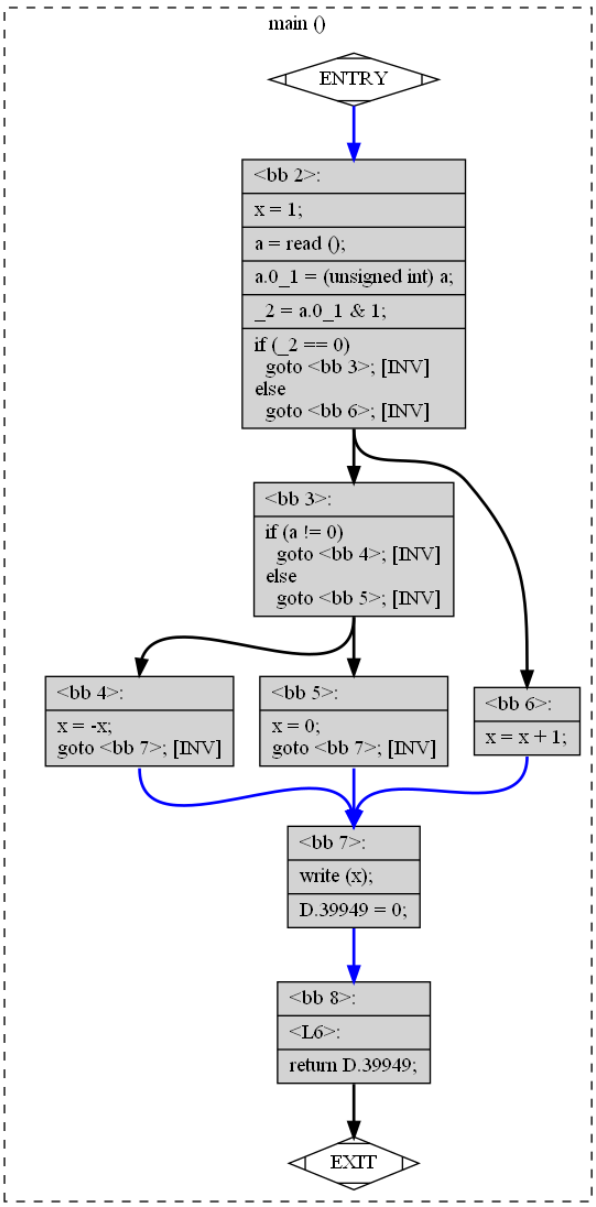

Figure 3.1: GCC AST Dump. This figure showcases the AST representation of listing 2.1 as dumped by GCC. Note that it is not easily comprehendable.

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`.

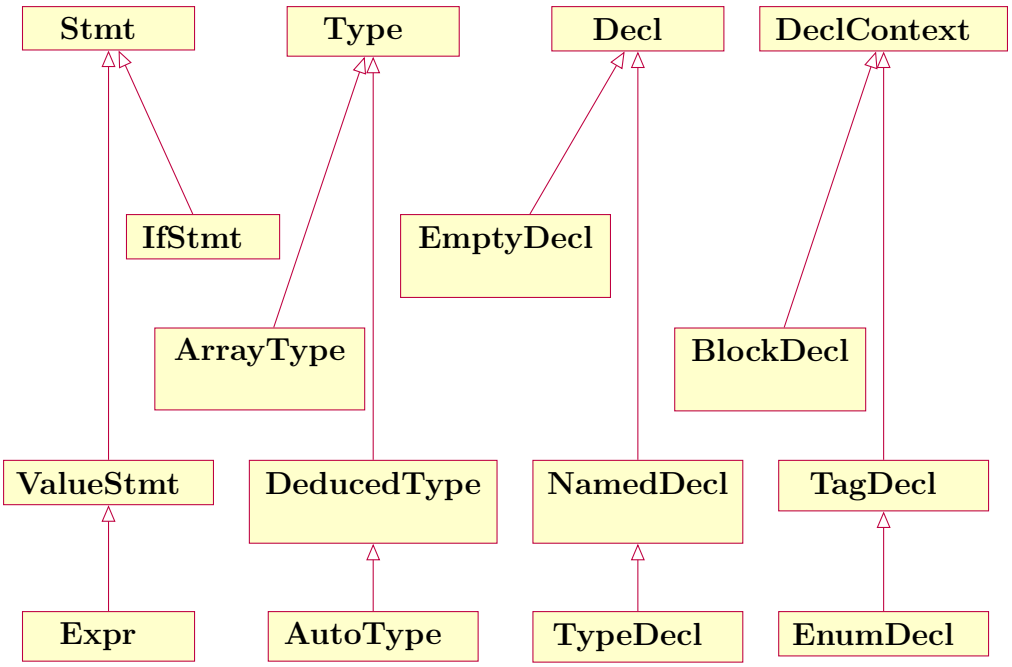

Figure 4.1: An example of the Clang AST class hierarchy. The figure contains only a handful of classes and their children. Note that the top most classes do not share a common ancestor.

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.

TODO: Come up with an actual way to make sure a program is minimal.

TODO: Come up with an approximation to guess whether the program will terminate.

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.

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.

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.

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.

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

Input:
L ... location of the error.
P ... the input source code.
A ... 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: $bitField \leftarrow Increment(bitField)$
6: $currentVariant \leftarrow (C_1, C_2, \ldots, C_n)$
7: **for** $i \in 0 \ldots n-1$ **do**
8: **if** bitField[i] $= 0$ **then**
9: $currentVariant \leftarrow currentVariant \setminus \{C_{i+1}\}$
10: **end if**
11: **end for**
12: $allVariants.Add(currentVariant)$
13: **end while**
14: $allVariants \leftarrow$ SortBySize($allVariants$, $Ascending$)
15: **for all** $V \in allVariants$ **do**
16: **if** IsValid(V, L, A) **then return** V.
17: **end if**
18: **end for**
19: **return** none.

Figure 5.2: Naive Statement Removal.

node u to node v if the code unit v is the subset of code unit u. I t is natural to wonder what exactly do these edges achieve.

TODO: Create an example of `if (...){} else(...){}` and their bodies.

Figure [ref] 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 program 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 statements 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.

The graph, visualized on Figure [ref], 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. 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 $\{v_1, v_2, \ldots, v_k\}$.

TODO: Add an example of the removed decl reference.

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.

TODO: Add references to the halting problem and Rice's theorem.

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 and Rice's theorem. 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.

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

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

Figure 5.3: Minimization Based on Slicing.

The proposed slicing-based solution is described in figure 5.3. 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 5.3. 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 can be 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.

Another thought-about approach is hybrid slicing. 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.

TODO: Get more information on hybrid slicing.

5.4 Program verification

TODO: Add https://youtu.be/UcxF6CVueDM?t=177 as a reference.

TODO: Add pseudocode explaining the steps of verification.

A simple way of finding the minimal failure-inducing subset of the program is by performing the following steps: We sort the generated variants by size, from the smallest to the largest. 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 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.

Depending on the input program's execution time, the verification might take more time than the already long variant generating step.

6. Implementation

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

In order to compare results, the different approaches described in chapter 5 need their concrete implementations. There are reliable implementations of some mentioned techniques, such as Delta debugging. However, only some of these implementations were used. Most notably, the static and dynamic slicers are reused from other works. The majority of the project was built using LibTooling and LLDB API.

TODO: link DD implementations.

The following sections describe the process of this project's development. Used technologies and implementations are discussed first. The rest is a description of the development of different approaches and their components.

6.1 Technologies

This project requires an effective way of recognizing and removing a language construct of C or C++ source code. In previous chapters, it has been concluded that an AST would be a good candidate for representing source code. In particular, the Clang AST offers the ability to remove source code mapped to nodes in the AST. By deleting either single nodes or entire subtrees, we can carefully reduce a program's source code. All AST-oriented operations and transformations are available in the LibTooling library.

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. The user also 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 is changing 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 External code

Some of the external code cannot be added to the C++ project due to compatibility reasons.

Giri

DG

TODO: Read the notes and the overview, summarize the contents.

6.3 Shared components

This project compares several reduction techniques. Each of these techniques is represented by its project or its script. Some projects work with code that is shared with other projects. This section will discuss the parts of this work that were reused across multiple techniques. Thus, these parts serve as a joint base for these techniques.

TODO: Read the notes and the overview, summarize the contents.

The following paragraphs explain the code behind validation and AST transformations. Generating variants is done by altering the AST or its underlying code. Sections 4.2 and 4.5 talk about the AST and how it can be traversed and modified. First, a frontend action is created, which then constructs a `Consumer` instance. The consumer can dispatch specific visitors and perform various operations.

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. The `Actions.h` and `Actions.cpp` files show concrete derivations of `ASTFrontendAction`. Moreover, some of the derivations use their custom factories. By default, any `ASTFrontendAction` can be created by calling the `FrontendActionFactory::create()` method. However, this function cannot provide any arguments for concrete `ASTFrontendAction` implementations. A workaround can be seen in the `Actions.cpp` file, which contains a custom factory. The factory takes the necessary parameters and passes them to a `Consumer` instance in its `create()` method.

Consumers. High-level actions have been coded into `Consumer` implementations. Current `Consumer` classes serve specific purposes. For example, the `VariantGenerationConsumer` does not invoke any visitor instances. Instead, it keeps two different `Consumer` objects. The pair of consumers has its required input and output. The `VariantGenerationConsumer` unifies the interface between the two consumers and allows them to communicate. Thus, these concrete `Consumer` classes also serve as a middleman for data transfers between visitors and the caller. An example of a consumer action would be generating every possible program variant. The `Consumer` contains the generating loop. It then dispatches a visitor inside the loop. The visitor might return results, which the consumer stores and uses for future operations. An example of data transfer might be specifying how a variant should look. This might be done by passing an object to the visitor. The visitor could then return a string representing that variant to the consumer.

TODO: Change to VariantGeneratingConsumer

Visitors. In this project, visitors perform relatively short actions. They might collect information during their traversal lifetime and return that information once the AST has been traversed. They might perform more complicated `Rewriter` actions based on the current node type. An example of an essential visitor for this project is the `MappingASTVisitor`. The purpose of this class is to split the code into units. Furthermore, it also specifies the dependencies between these units.

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 most approaches presented by 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 naive algorithm was described in section refchap:naive. Unlike greedy algorithms, the naive approach can guarantee minimality. As such, much time went into improving the implementation.

The approach works by deploying a `DependencyMappingConsumer`, whose primary job is to split the AST into code units. The mentioned consumer dispatches a visitor that creates the traversal order. The visitor considers declarations and statements. It determines whether these nodes should be visited by the other visitors and maps their dependencies. In short, one node is dependent on another if it is in the other node's subtree. Another rule for node dependencies states that usages of variables depend on their declarations. Function definitions and calls follow a similar rule.

Once the `MappingASTVisitor` has traversed the AST, the `DependencyMappingConsumer` collects its output, and the variant generating function is initialized. Before any actual variants are created, the algorithm first separates all valid variants into bins of different sizes—the binning works as follows. The `DependencyMappingConsumer` has determined n: the number of code units in the file. Moreover, it has set the order in which nodes are traversed.

We can create a bitfield of size n, where the i^{th} bit represents the i^{th} node in the traversal order. If the bit is set to `true`, the node will be preserved. Otherwise, it will be removed from the variant. This gives us 2^n bitfield variants, the same number required for all program variants. With the bitfield representation, we could use bitwise operations to move between different variants. The only operation necessary for generating all variants is the increment function. While cycling through all possible bitfield configurations, we check that each obeys the dependency rules. Valid bitfields also carry the source code size of the program variant they represent. Variants are assigned into categories based on their represented size.

The idea is to search iteratively, generate minor source code variants, and validate them first before moving on to the remaining possible variants. The number of bins represents the granularity with which the deepening search is conducted. It makes sense to set the granularity high for more extensive programs. After the binning process is complete, the variant generating loop is launched.

The loop considers all bitfields in a given bin. In each iteration, the bitfield is passed to a `VariantPrintingASTVisitor`. The visitor traverses the AST in the order given by the `MappingASTVisitor`. The nodes represented by the bitfield are either kept or removed based on the value of each bit. It should be stated that the nodes are not removed. Instead, it is the underlying source code that is being deleted using a `Rewriter` operation. The removal also follows an explicit rule. Nodes are removed only if their parents will not be removed. This rule eliminates the chance of removing an underlying code snippet twice. Each `VariantPrintingASTVisitor` handles a single variant.

After all variants from the given bit are processed, all results are tested for validation. In case a valid result is found, the search ends. Otherwise, the search continues with the bin representing the following smallest variant sizes.

6.5 Delta debugging

6.6 Systematic approach

TODO: Might require mentioning the component that unifies slices or extracts all variables from a line.

The systematic approach comprises multiple steps. 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 base is a Python script that invokes all necessary components. The script uses Docker API to launch a 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. Giri is launched analogically.

Both slicers return a list of lines that represent the slices. This output needs to be processed further. The script invokes the `SliceExtractor` program. The program transforms a source file into the desired slice based on the given list of lines. It does so based on ASTMatchers, removing the complement of the given list of lines by using `Rewriter` operations. 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.

After passing through the two slicers, the intermediate result is further reduced using the naive approach. The Python script executes the naive algorithm, which then produces the desired results. Another approach can also substitute the ultimate step. One can easily swap between the naive reduction and Delta debugging by simply changing the path to the executable in the Python script.

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.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment