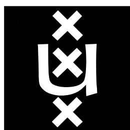
Making compiler tests based on compiler source code coverage

Jelle Witsen Elias
jellewitsenelias@gmail.com

January 19, 2021, 33 pages

Academic supervisor:	Clemens Grelck, c.grelck@uva.nl
Daily supervisors:	José Luis March Cabrelles, joseluis@solidsands.nl
	Remi van Veen, remi@solidsands.nl
Host organisation:	Solid Sands, www.solidsands.nl



Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Compilers are important elements in software infrastructure. It is therefore important that these complex pieces of software are tested. We found a way of making C compiler tests that is different from the usual approach of doing this. With our new process we base new tests on their coverage of compiler source code, instead of constructing them by solely looking at the rules defined in language standards. This way, we manage to make tests with constructions that were overlooked when building our existing compiler test suites. We built a tool that takes source code of a C application as input, along with a C compiler test suite. The tool outputs pieces of code which, when compiled, cover parts of compiler source code that the test suite does not yet cover. With this output, we can construct new tests for inclusion in our existing test suite.

Our process is as follows: we take an open source C application and compile its sources with an open source compiler. This compiler has been instrumented to enable source code coverage analysis. After compilation, we compare the coverage of compiler source code of the application with coverage of our existing test suites. If additional coverage is found, we reduce the source code of the application by removing parts of the code that do not contribute to the additional coverage. To do this, we use a reduction tool and a source code coverage analysis tool. The reduction process could introduce undefined or unspecified behaviour. We mostly prevent this from happening by using a static analysis tool that can detect most forms of undefined/unspecified behaviour. The final step in our approach, which is to make a test out of the reduced pieces of code, requires manual work at the moment. We manage to produce tests that are suitable for inclusion in our existing test suites.

Contents

1 Introduction **4**

2 Background **7**
2.1 C compilers . . . 7
2.2 Code coverage analysis tools . . . 7
2.3 Program reduction tools . . . 8
2.4 Tools that can identify undefined behaviour . . . 10

3 Methodology **11**
3.1 Compiler used . . . 11
3.2 Analysing compiler source code coverage . . . 12
3.3 Comparing compiler source code coverage . . . 12
3.4 Preparing the application for reduction . . . 13
3.5 Reducing application code based on compiler source code coverage . . . 13
3.6 Preventing undefined behaviour . . . 14
3.7 Unrelated additional coverage of a file . . . 15
3.8 Reducing larger projects . . . 16
3.9 Filtering results . . . 17
3.10 Making tests . . . 17

4 Results **18**
4.1 Experiment process . . . 18
4.2 Tool output . . . 18
4.2.1 Specific keyword combination . . . 19
4.2.2 Preprocessor directive . . . 19
4.2.3 File ending with a single line comment . . . 20
4.2.4 Complex but valid syntax . . . 20
4.2.5 Variable number of arguments with double type argument . . . 21
4.2.6 Binary constant . . . 21
4.2.7 Single instruction, multiple data processing . . . 22
4.2.8 Bitfield with long type . . . 22

5 Discussion **24**
5.1 Categorising the results . . . 24
5.2 Reducing several projects . . . 24
5.3 Speed of the reduction . . . 24
5.4 Limitations . . . 24
5.5 Automating the construction of tests . . . 25

6 Related work **26**
6.1 Automatic compiler test generation based on attribute grammars . . . 26
6.2 The test oracle problem . . . 26
6.3 Random C program generation: CSmith . . . 27
6.4 Test generation in functional programming . . . 27

7 Conclusion **28**
7.1 Summary . . . 28

7.2 Answering the research questions 29
7.2.1 RQ1: How can we reduce C programs that exhibit compiler coverage which is of specific interest, to a program that is smaller but still has this coverage? 29
7.2.2 RQ2: How can we make sure that such a reduced program does not exhibit undefined or unspecified behaviour? 29
7.2.3 RQ3: How can the problem that different additional coverage is found by unrelated lines of code be addressed? 29
7.2.4 RQ4: How can we make a test (appropriate for inclusion in a test suite) out of the reduced program? 29
7.3 Future work 29
7.3.1 Reducing generated C files 29
7.3.2 Test generation based on an attribute grammar 30
7.3.3 Making our tool work on different compilers 30

Bibliography **31**

Acronyms **33**

Chapter 1

Introduction

Compilers are used everywhere. They are often very complex pieces of software; for example, in 2015 the GNU Compiler Collection (GCC) consisted of more than 14.5 million lines of code [1]). Their high complexity means that compilers are very unlikely to be completely error-free.

Because compilers are so widespread and such crucial pieces of software [2], testing them is very important. Just as code that is compiled using a compiler should be tested, the compilers themselves should be tested. If a compiler contains a bug, potentially a lot of programs that are compiled with it will contain errors. Bugs in a compiler are also very hard to debug for an end-user when compiled programs behave incorrectly. The incorrect behaviour could be attributed to the compiled program instead of the compiler itself, because it might not be immediately clear that the compiler itself is responsible for the erroneous behaviour. Therefore, preventing bugs in the compiler has significant value.

Test suites can be used to test compilers. This project's host company, Solid Sands[1], maintains a collection of such test suites called SuperTest. Solid Sands was founded in 2014 and it provides compiler testing and qualification technology. Their business is built around the SuperTest test suites.

The goal of this project is to see if we can develop a new approach for constructing tests that can be included in our test suites. The idea is to build tests by looking at coverage of compiler source code.

Normally, tests are made by looking at language specifications. For every rule in a language specification, a test is made to check whether this rule is correctly followed by the compiler. This means that tests are not made or checked for their coverage of compiler source code. It might be the case that important portions of compiler source code are not tested by using this method of making tests. For example, making tests based on language specifications does not ensure that optimization code of the compiler will be tested. This means that there might still be errors in the compilers in these uncovered sections of code. We would like to examine whether it is possible to also make tests based on compiler source code coverage, to improve the total compiler source code coverage of the test suites.

To make tests based on compiler source code coverage, our approach is to compare coverage of real-world applications (taken from GitHub [3], for example) to the coverage of our existing test suites. This is illustrated in Figure 1.1. If an application covers a part of compiler source code that the test suites do not, the source code of the real-world application would need to be analysed to see which sections of code cause the additional compiler coverage. Then we would be able to make a test that has this additional coverage as well. This analysis can be a very tedious process if done entirely by hand, especially for large applications.

Reducing the size of the program by eliminating sections of code which do not contribute to the compiler coverage of interest can make this process a lot easier. Research has been done to reduce programs based on the program triggering a certain compiler warning or having a different property [4–6]. How programs can be reduced based on their coverage of compiler code has not been researched yet. This is what we would like to examine with this research project.

The tools at the core of this project are a code coverage analysis tool called Gcov [7] and a source code reduction tool called C-Reduce [4]. The former can analyse code coverage when running a program. We use this tool to check an application's compiler source code coverage when it is compiled. The latter tool can reduce source code based on a given property. Such a property could be, for example, that the compiler throws a certain warning when compiling this piece of code. C-Reduce will then remove parts of code that do not contribute to the compiler producing this warning. The result will be a smaller version of the code that still makes the compiler emit this warning. How C-Reduce works will be explained

[1] `https://solidsands.com/`

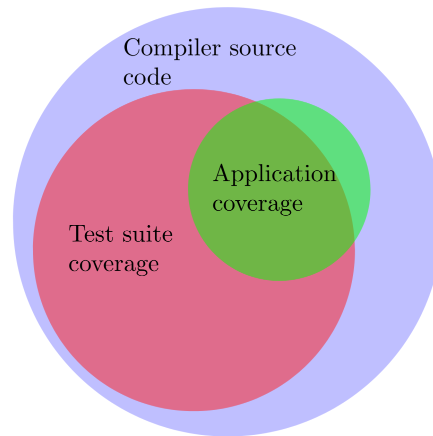


Figure 1.1: Diagram showing coverage of compiler source code. We are interested in the coverage of the application (green circle) that is not covered by the test suite (red circle).

in more detail in Chapter 2. C-Reduce was built for isolating compiler bugs, but we use this tool in a different way: to reduce a file based on its coverage of compiler source code.

We use these tools in an automated way, and our use case is not necessarily the use case that these tools were intended for. This makes our project unique. It requires work to adapt the tools and combine them to build our own, overarching tool that is built for our purposes. Our tool uses existing tools as sub-components. By combining them and adapting them for our purpose we add value.

The input of our tool is the source code of a C application, which could be taken from GitHub. Our tool then produces reduced versions of all the C source files in the input project. These reduced files need to be analysed further (by hand) to build tests out of them. These tests can then be included in our SuperTest test suites.

There are a few things to consider when building a tool like ours. Firstly, how can we prevent undefined behaviour in the output? If a file is reduced while only defining that the reduced file should compile, it could very well be that undefined behaviour will be introduced. This would be a problem, since undefined behaviour means that the language standard specifies that anything is allowed to happen when this code is run. This makes building a test for such code impossible, since the standard does not define what should happen. Different compilers can do different things with the same piece of code if there is undefined behaviour in it, and we want to make generic tests to test for a compiler's compliance with the language specification. We cannot do this with tests that include undefined behaviour. To prevent undefined behaviour in the output of the reduction tool, we use code analysis tools. These are discussed in Chapter 2.

Another problem to think about is that it could well be that an application covers two sections of compiler source code that are completely unrelated. This would mean that there are two 'parts' in the application source code that independently cause additional coverage. If this is the case, we would end up with a reduced file with two separate elements of interest, and actually two tests should be made out of it. Would it be possible to split the reduction up in two: once for the first element that causes additional coverage, and once for the second element? We experimented with this by grouping lines of covered compiler source code together (for instance, because they are less than 30 lines apart), and then reducing a file several times; once for each group.

One final issue is the question of, given that the reduction process has worked, how we can make a test out of the reduced file. How to define what the behaviour of the reduced code should be? This question concerns the actual final goal of this project: to construct tests out of these reduced programs, so the quality of existing test suites can be improved.

This brings us to the following research questions:

- RQ1: How can we reduce C programs that exhibit compiler coverage which is of specific interest, to a program that is smaller but still has this coverage?
- RQ2: How can we make sure that such a reduced program does not exhibit undefined or unspecified behaviour?
- RQ3: How can the problem that different additional coverage is found by unrelated lines of code

be addressed?

- RQ4: How can we make a test (appropriate for inclusion in a test suite) out of the reduced program?

An overview of the process that we used to reach our goal of making tests is shown in Figure 1.2.

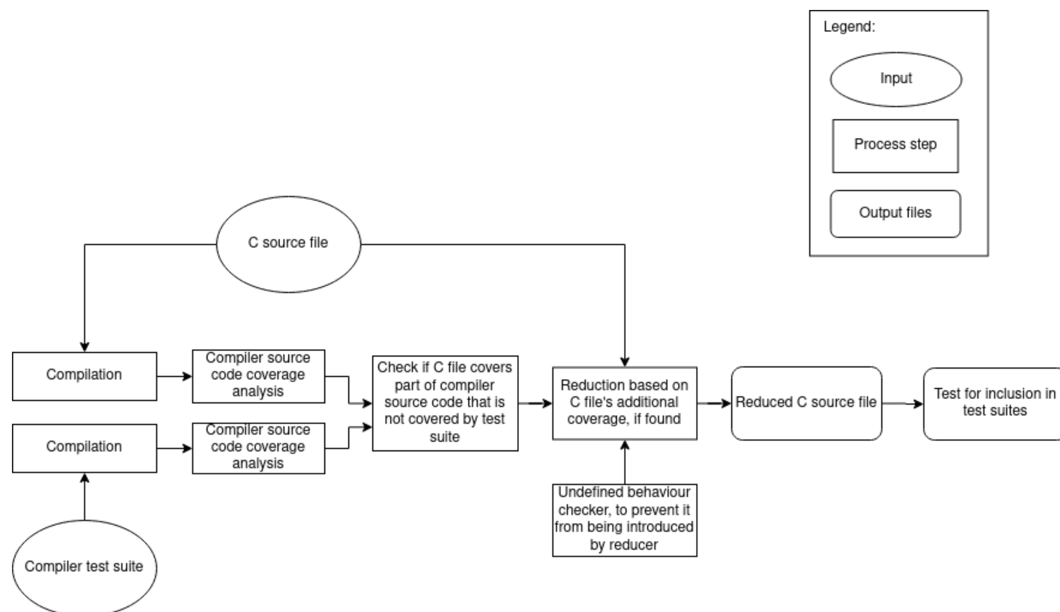


Figure 1.2: Diagram showing the process of how to reach our goal of making tests.

We hope to find new tests that we can include in our test suites using our approach. Because our usual process of making tests is based on looking at language specifications, it makes sense to think that we miss certain coverage of compiler source code. Compiler source code coverage was not taken into account when building the tests. This is why we expect to find uncovered parts of compiler source code with our project, and we hope to be able to make new tests that do cover these parts.

The remainder of this thesis is organised as follows. In Chapter 2 we describe the background of this thesis. In Chapter 3 we describe our methodology. Results are shown in Chapter 4 and discussed in Chapter 5. Chapter 6 contains the work related to this thesis. Finally, we present our concluding remarks in Chapter 7 together with future work.

Chapter 2

Background

This chapter presents the necessary background information for this thesis. We explain the existing software tools that are relevant for this project. These include code coverage information tools, code reduction tools and tools that can identify undefined behaviour. We also discuss C compilers that we could use for this project.

2.1 C compilers

Many different compilers exist that we could experiment with in this project. We will discuss a few options here. Firstly, there is the TinyC Compiler (TCC) [8], which is a relatively small compiler that has around eighty thousand lines of source code. Its distinguishing features are that it's small and fast ('several times faster than GCC', according to the website of TCC [8]).

Another compiler we could use for our project is GCC. It This is a 'free' piece of software, meaning that it respects the freedom of the user [9]. Users have the freedom to use, copy, spread, study, change and improve the software [10]. This compiler is a lot more complex than TCC: it contained over 14.5 million lines of code in January 2015 [1].

Other well-known C compilers are the Intel C++ Compiler (ICC) and Microsoft's Visual Studio C compiler. These compilers are closed source and are not free like GCC. The ICC compiler is part of Intel's OneAPI Base Toolkit, which a set of tools to develop applications across a variety of architectures. ICC is built to compile source code as efficiently as possible for Intel processors and supports the latest C and C++ language standards. It integrates into Microsoft's Visual Studio Integrated Development Environment (IDE) [11].

Microsoft maintains a compiler that is included in their Visual Studio IDE called Microsoft Visual C++ (MSVC) [12]. This is proprietary software and its source code is not openly available.

2.2 Code coverage analysis tools

A tool that can analyse code coverage when a certain application has been run is needed for this project. These tools work as follows: an application is compiled with a specific compiler flag to enable code coverage analysis. This generates analysable coverage files. The coverage tool can then be run to analyse these coverage files. The tool outputs annotated source code files. These are copies of the original source code files where every line is annotated with a number, which indicates how often the line was executed.

Some lines of output of such a tool called Gcov [7] is shown in Listing 2.1.

```
...
        1: 1427:        switch(op) {
    #####: 1428:        case '+': l1 += l2; break;
    #####: 1429:        case '-': l1 -= l2; break;
    #####: 1430:        case '&': l1 &= l2; break;
    #####: 1431:        case '^': l1 ^= l2; break;
    #####: 1432:        case '|': l1 |= l2; break;
        1: 1433:        case '*': l1 *= l2; break;
...
```

Listing 2.1: Partial coverage information of a compiler source file

The 1s before the colons indicate that lines 1427 and 1433 have been executed once. The `#` symbols at the start of the other lines mean that these lines were not executed. Apparently, the application source code contained a multiplication operator.

Gcov comes with GCC. Gcov can analyse source code coverage by compiling a piece of code with GCC, while applying the `-fprofile-arcs` flag to instrument the code, and then executing it. An alternative to Gcov is llvm-cov [13]. It is compatible with Gcov and its functionality is similar.

We can use these tools for analysing compiler source code coverage. This is done by instrumenting a compiler by compiling it with a specific flag, and then running the compiler on a piece of C code. We can then analyse which lines of the compiler source code have been covered by compiling this piece of C code.

2.3 Program reduction tools

Several reduction tools exist that can reduce a file (for instance, a piece of code written in C) based on a certain property [4–6]. The reduction tools take a file with source code and a so-called property script as input. This property script should return zero when it holds for a given piece of C code, and a non-zero value if it does not hold. The property should hold for the file with source code, meaning the script should return zero when applied on the source code file. The source code is then reduced in such a way that the property still holds for the reduced version of that file. This way, a certain aspect of the code, defined in the property, can be isolated. The following example demonstrates how these reduction tools can generally be used. The example shows how the reduction tool C-Reduce [4] works, but the principles of how it works apply to other reduction tools as well.

Let's say we have a C file called `reduce.c`, shown in Listing 2.2.

```
1 #include <stdio.h>
2 int main(void) {
3     int a = 0;
4     printf("test");
5     return a;
6 }
```

Listing 2.2: `reduce.c` before applying reduction

We take this as the basis for the reduction tool to reduce. Now, we need to define what we want the file to be reduced to. Say that, for instance, we want to maintain the print statement but do not care about the other parts of the code. We can make a property script, written in Bash, as shown in Listing 2.3.

```
1 #!/bin/bash
2 grep "printf(\"test\");" reduce.c >/dev/null 2>&1
```

Listing 2.3: The property for `reduce.c`

The script returns zero when it finds a line with `printf("test")` in it, and a non-zero value otherwise. If we then run a reduction tool with these two files as input, we can expect output to be as shown in Listing 2.4.

```
1 printf("test");
```

Listing 2.4: `reduce.c` after the first reduction attempt

A problem with a file like this is, however, that it does not compile. To prevent this, we add a line to the property script that the file should be compilable without warnings (by GCC, in this example). The resulting property script is shown in Listing 2.5.

```
1 #!/bin/bash
2 gcc -Werror reduce.c >/dev/null 2>&1 &&\
3 grep "printf(\"test\");" reduce.c >/dev/null 2>&1
```

Listing 2.5: An improved property for reducing `reduce.c`

Running a reduction tool on this input could give us the code shown in Listing 2.6.

```
1 #include <stdio.h>
2 main() { printf("test"); }
```

Listing 2.6: `reduce.c` after the second reduction attempt

This file compiles fine and contains the code that we were interested in (the `printf` statement).

Reduction tools are generally made for finding compiler bugs, such as when an error is incorrectly produced while compiling a large file. In this case, the property script could be made to return zero when this error is given by the compiler, and a non-zero value otherwise. The reduction tool will then try to make the file as small as possible, to isolate what piece of code causes the compiler to produce this error. The reduction tools are not necessarily made for reducing based on compiler source code coverage, but because they work in such a general manner with property scripts, they might be adaptable to suit this purpose. We will discuss different reduction tools that we could use here.

C-Reduce [4] is a reduction tool that was originally made for C, but has been adapted to work with C++ as well. It is used in exactly the way as described in the previous example. It works by transforming the original source code using transformation modules, or 'passes'. Such a pass could be, for example, removing a comment or replacing a binary operator by one of the operands. Every time a transformation is done, C-Reduce considers the transformed source code as a new variant. The property script is then run which checks this variant.

If the property script returns zero, the variant is accepted as the new starting point, and the same process is repeated. If the property script returned a non-zero value, however, the variant is discarded and C-Reduce goes on with trying a different pass. When no transformation succeeds in producing a correct variant anymore, C-Reduce is finished. The output is the variant when this point has been

reached. This output is likely to be reduced in size compared to the input.

C-Reduce copies the file it needs to reduce into a fresh temporary directory, where it works on reducing the file. This means that the files need to stand on their own and cannot rely on other files such as files included from the projects directory.

An alternative to C-Reduce, that was developed after C-Reduce, is Perses [5]. One reason for developing Perses as an alternative to C-Reduce, is that C-Reduce is not generally applicable to any language: in the basis it only works for C [5]. It is time-consuming to adapt it for other languages. Perses tried to improve on this by taking any language grammar specified in Backus-Naur Form (BNF) as input. It is able to reduce any language that is formalised in this way. Any context-free grammar can be represented in a BNF, so Perses can handle any such grammar. Perses is based on language syntax, and can thus be called a Syntax Guided Program Reduction (SGPR) tool. It works by taking an Abstract Syntax Tree (AST) of the source code, and transforming it node for node. C-Reduce is not a SGPR tool, so it is fundamentally different. Perses produces generally larger file size compared to C-Reduce, but it is faster and applicable to more languages.

Another reduction program that expanded on the work of Perses is Pardis. Pardis is, just like Perses, an SGPR tool. An observation that the developers of Pardis did about Perses was that it suffers from *priority inversion*. This means that significant time can be spent on transforming AST nodes with a low weight: nodes with a little amount of tokens in its subtree. It would be better to focus on the nodes of the AST that do have a lot of tokens in their subtree. This is what the developers of Pardis improved. Their paper is thus called 'Priority Aware Test Case Reduction' [6]. Pardis maintains a priority queue, which is based on the number of tokens a given AST node has. Pardis also uses other pruning tactics. These improvements result in Pardis on average performing better than Perses on three aspects: reduction time, amount of tokens in the final output and amount of oracle queries done, the last being the amount of times the property to check was queried.

2.4 Tools that can identify undefined behaviour

Regehr et al. [4] discuss in their C-Reduce paper that they can prevent undefined behaviour in the output of C-Reduce by using Frama-C [14] and KCC [15]. These are C code analysis tools that can catch certain types of undefined behavior, such as usage of uninitialized variables. Frama-C and KCC both do run-time analysis by instrumenting the code.

KCC is a C semantics interpreter which is also capable of debugging and catching undefined behaviour, among other things. It needs source code with an entry point to do its work, because it compiles and then runs the code to do its analysis.

Frama-C is a suite of tools for static analysis of C source code. It has a value analysis plugin that can detect run-time errors statically in a programs source code. It produces alarms for sections of code that have a possibility of causing run-time errors. With this plugin, undefined behaviour can be checked for. The developers of C-Reduce also fixed bugs in Frama-C and extended this value analysis plugin [4].

GCC [16] and a tool called Clang-Tidy [17] can detect certain types of undefined behavior as well. They are less thorough analysers than Frama-C and KCC, but they do their analysis statically. This means that they do not need an entry point in the code, such as a main function, to do their work. GCC has an option to check for usage of uninitialized variables/memory (the `-Wuninitialized` flag). However, compared to Clang-Tidy, it does not have an as extensive array of undefined behavior checks that can be enabled. Clang-Tidy has very specific flags that can be enabled individually to check for undefined behavior. Some common types of undefined behaviour, such as usage of uninitialized memory, can be checked with this tool. Both GCC and Clang-Tidy can only catch undefined behaviour that is statically detectable.

Chapter 3

Methodology

In this Chapter we discuss how we implemented our tool to reduce programs based on compiler source code coverage. We explain how the tool is built up and which existing software we used to build it. We also discuss why we chose to use certain existing pieces of software over others. An updated version of Figure 1.2 (showing the process that we use to reach our goal of building tests) is shown in Figure 3.1. This new figure shows the names of existing software tools that we use as sub-components for our own overarching tool between parentheses, at each step of the process.

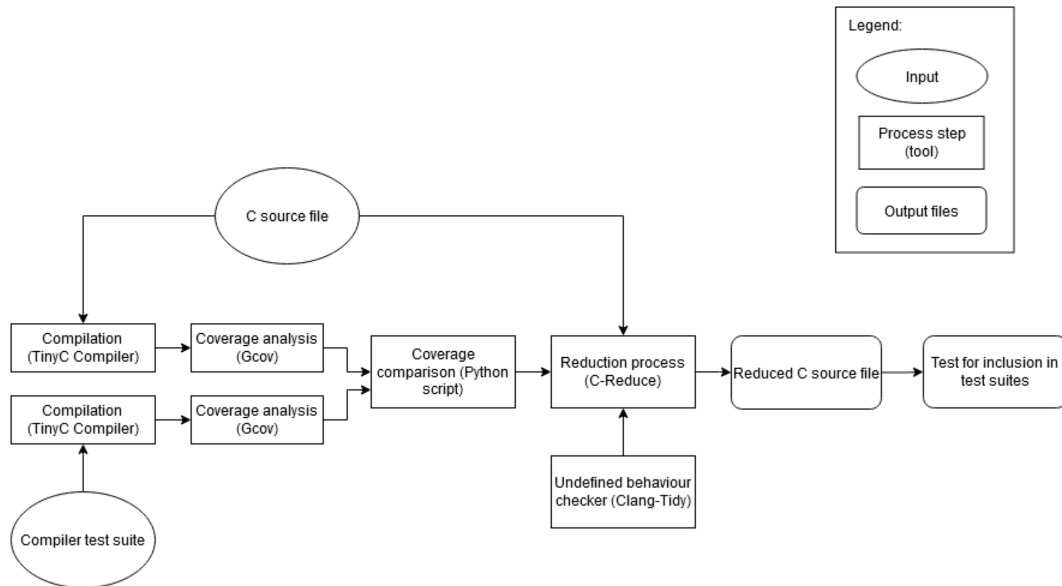


Figure 3.1: Updated version of Figure 1.2, showing how our tool is built up. Names of existing software that we used as sub-components of our tool are added in parentheses.

3.1 Compiler used

First, we need to choose which compiler we use to work with. We like the compiler to be lightweight and not too complex, so we can experiment well. If the compiler has fewer lines of code, this will mean that running our tool will take less time. The reason for this is that there are less lines of code to compare between our test suite and the C project we are analysing.

We decide to use the TCC compiler to experiment with in our project. It is a compiler with a relatively low complexity and not too many lines of source code. The compiler is easy to compile, and this takes a relatively small amount of time. This is an advantage because we need to experiment with instrumenting the code for coverage analysis, which might mean that we need to compile its sources often. Another advantage is that all the important source code of the compiler is in one directory - there is no complex directory structure with source files in different subdirectories. This makes it easier to do coverage analysis. Also, if our approach works for TCC, it will work for other compilers of which the

source code is available as well, such as GCC and Clang. It is not possible to work with closed-source compilers such as ICC and the Microsoft Visual C compiler, because it is not possible to do source code coverage analysis without source code access.

3.2 Analysing compiler source code coverage

In order to be able to do a comparison of compiler source code coverage between a test suite and a C project, it is necessary to get this compiler source code coverage information for both the C application and the test suite.

We do such coverage analysis with the the Gcov [7] tool. The compatible llvm-cov [13] tool could be used as well. Because we compile TCC with GCC, we use the GCC tool Gcov.

The following approach is used to analyse compiler source code coverage:

1. Compile TCC with GCC, with flags that instrument the code to enable the ability to do coverage analysis with Gcov.
2. Compile the program of which compiler source code coverage needs to be analysed with the instrumented TCC binary. This compilation produces coverage data files, which is used by Gcov at the next step to analyse coverage.
3. Invoke Gcov to analyse the data files produced in the previous step. This will analyse the coverage of compiler source code by the compiled application.
4. Look at the Gcov output for covered code.

3.3 Comparing compiler source code coverage

The next step in our process is to compare the compiler coverage information from the test suite to the coverage information from the C project. We need to do this to define what the lines of compiler source code that were covered by the C project, but not by the test suite. This is the basic analysis we do to reach our goal of making reduced pieces of code that exhibit this additional coverage.

We use a script to compare the compiler source code coverage of the test suite with the coverage of an application. Some example output of this script shown in Listing 3.1.

```
...
    Line 1 1:        1:        1: 1427:        switch(op) {
    Line 1 0:        1:    #####: 1428:        case '+': l1 += l2; break;
    Line 0 0:    #####:    #####: 1429:        case '-': l1 -= l2; break;
    Line 0 0:    #####:    #####: 1430:        case '&': l1 &= l2; break;
    Line 0 0:    #####:    #####: 1431:        case '^': l1 ^= l2; break;
    Line 0 0:    #####:    #####: 1432:        case '|': l1 |= l2; break;
    Line 0 1:    #####:        1: 1433:        case '*': l1 *= l2; break;
...
```

Listing 3.1: Partial output of the coverage comparison script

This Listing is a comparison file of the compiler coverage of two very simple C files. For this example, we show the same switch statement as shown in Listing 2.1. Note that this Listing 3.1 differs from Listing 2.1, because this one shows the output of our coverage comparison script. In Listing 2.1, we showed Gcov output instead. Our comparison script compares two Gcov output files and produces a result as shown in the current Listing.

The first of the C files we use for the example to demonstrate our comparison script has an addition expression, and the other a multiplication expression in it. With the comparison script, Gcov output for both files is combined and compared. As can be seen in the example output from Listing 3.1, the + case of the switch is covered by one of the files, but not by the other. This is indicated by the column containing `Line 1 0:`. The `1` means that the first file does cover this line of compiler source code, and the `0` means that the second file does not cover it. The next two columns contain the Gcov execution marking of the addition file and multiplication file. Here it can again be seen that line 1428 is covered

by the first file, but not by the second file. The last line of the example shows the opposite: line 1433 is covered by the second file but not the first file. Using this difference, we mark line 1433 in our reduction property and reduce based on the coverage of this line.

3.4 Preparing the application for reduction

The reduction tools work on a file by file basis, so when reducing an application with several source code files this might cause problems. The source files could include other source files, such as header files, for example. When source files rely on each other in this manner, this could be problematic for reducing the files separately. This is why we need to prepare the source files before reducing them.

This preparation entails that we apply the preprocessor of the compiler on the files before reducing them. This solves the problem, because it makes sure that the reduction tools can reduce the files individually. For instance, when macros are defined in a certain header file that are used in a file that we want to reduce, the files cannot be reduced separately but only when taken together. Since the contents of the header file are included in the file to reduce after preprocessing it, this issue is resolved.

Besides, we do not engage the linker of the compiler when checking for the files compilability in the reduction property script. The files that we want to reduce will not always contain a `main` function. We don't need our output files to have this function, because we will not run the reduced files. We only care about compiling them to analyse their coverage of compiler source code.

3.5 Reducing application code based on compiler source code coverage

Next we need to bring the coverage information and reduction tool together: we want to reduce a project based on the additional coverage it has over the test suite we are comparing to. The reduction tools are not necessarily made for this purpose. We need to use them in such a way that we get a reduced file with the additional compiler coverage.

Although the reduction tools are not built for our specific use case, the fact that they take a single script that should either return a zero or a non-zero value make them quite flexible. While they are built for finding compiler bugs, they can be used for a far wider range of applications because they work in such a generic manner. This is a major reason for these tools being suitable for adaptation to our purpose.

We started out by determining which reduction tool would fit best for our use case. We ended up using C-Reduce, instead of Perses or Pardis, for a few reasons:

- It is modular. It has transformation modules which can each be disabled independently.
- It produces small output. The output of C-Reduce is on average about three times as small as Perses' output [5]. Pardis' main improvement over Perses is speed, not necessarily output size [6]. A downside to using C-Reduce is that it is somewhat slower than the other tools. However, for the purpose of this project, we want to build a test out of the reduced programs. This is easiest if the size of the reduced file is as small as possible. Speed does not have the highest priority for this project, because we do not expect to find a lot of additional coverage in most projects. This means that we generally will not need to reduce files in bulk. Also, every reduced file is quite valuable and will need to be analysed by hand to make a test, which is time consuming. We don't need a very large amount of reduced files in a short amount of time because of this.
- The main motivation for the Perses developers to make an alternative to C-Reduce (and the Pardis developers to improve on Perses), is that C-Reduce lacks generality. It is highly customized for certain languages [5]. This downside does not apply for us, since we are only focusing on C. This is the language that C-Reduce was initially built for.

With our tool, we can reduce a small file with only a few elements in it, based on its coverage of compiler source code. An example program to reduce is the file shown in Listing 3.2.

```
1 int main(void) {
2     int a = 1 + 2;
3     int b = 1 * 3;
4 }
```

Listing 3.2: Code before reduction

Our tool analyses the coverage of the compiler sources of this file by using Gcov. In the Gcov output, line 1428 and line 1433 as seen in Listing 3.1 are marked as covered. These lines correspond to the addition and multiplication operators in the C file. In the hypothetical situation that line 1428 is covered by SuperTest, but line 1433 is not, our tool will start reducing the file based on coverage of line 1433. The property script returns a zero when line 1433 of this compiler source file is covered, and a non-zero value otherwise. Also defined in this property is that the code should compile without errors. Using this property, our tool transforms the code from Listing 3.2 into the code seen in Listing 3.3.

```
1 int a() { 1 * 3; }
```

Listing 3.3: Code after reduction

So the reduction works as expected. Only a multiplication expression in a function is left in the result. It needs to be encapsulated in a function, because in the property we defined that the file should compile without errors.

3.6 Preventing undefined behaviour

One problem we encounter in the reduction of projects is that undefined behaviour could be introduced. An example of a reduced program with undefined behaviour in it is shown in Listing 3.4. This piece of code has three problems. Firstly, the `c` variable is not initialized, but its field `b` is read on line 5. This causes undefined behaviour. Secondly, the function `a` does not specify a return type. While the C90 standard does allow this, defaulting the return type to `int` [18], it is not allowed from C99 onwards [19]. Lastly, an empty second operand in the ternary expression (line 5, after the `?` symbol) is a GNU extension [20]. This is not something that we want to test in our test suites. We test for compliance with the C language specifications, which do not include these GNU extensions.

```
1 a() {
2   struct {
3     long b : 5;
4   } c;
5   c.b ?: "";
6 }
```

Listing 3.4: Reduced code without using Clang-Tidy

To prevent this undefined behaviour to be introduced by our tool, we need a pass in the property script for the reduction tool that tells the reducer to disregard files with undefined behaviour in it. To be able to do this, we need a static analysis program that can detect certain types of undefined behaviour.

We use Clang-Tidy for this detection. Frama-C and KCC are not viable options for us, because they do runtime analysis. This means that they need an entry point in the code, such as a `main` function. As discussed in Section 3.4, we do not always have such an entry point.

Another problem with Frama-C is that it does not work properly on TCC-preprocessed code. This

means the options that were left for us were GCC and Clang-Tidy. GCC is not so reliable on uninitialized memory checks, since it is susceptible to false positives. With GCC 11, the checks should be more sophisticated [21], which means that it might become an interesting option in the future. For the time being we use Clang-Tidy, which suits our needs well.

```
1 int a();
2 int main() {
3   struct {
4     long b : 5;
5   } c = {1};
6   a(c.b);
7 }
```

Listing 3.5: Reduced code with using Clang-Tidy

Output with Clang-Tidy enabled is shown in Listing 3.5. The three issues from Listing 3.4 are resolved. The c object is properly initialized, and the return types of the functions are now specified. The problematic ternary expression is also not present in this file. This example demonstrates that Clang-Tidy can be configured to do more than just preventing undefined behaviour.

3.7 Unrelated additional coverage of a file

One further issue that could arise with our tool is that a file in a given project covers multiple distinct 'groups' of compiler code. A situation could arise where there are two separate parts of the compiler code that our test suite does not cover - for instance some compiler code about specific operator usage and another part of compiler code about bitfield usage. If we now try to reduce a file with both of these elements in it (this specific operator usage and usage of a bitfield), our tool will try to reduce it based on two separate properties. It would then not be very clear what the element in the output is that causes additional coverage, since there are two elements. We should also make two tests out of the reduced file, since the separate elements should both be tested individually. The reason for this is that it should be clear when a test fails, what the type of construction is that causes the failure. If there are two constructions that could both be attributed to the failure, this is not clear.

This problem of unrelated coverage is partially addressed by doing multiple reductions per input file: one reduction for every compiler source file (except if no additional coverage is found for a given compiler file). If unrelated additional coverage is found in different compiler source files, we prevent doing complex reduction by going one by one through the compiler files. The idea for doing this is to make a reduced test file as small as possible, and to make it only exhibit a single interesting bit of coverage, but not multiple unrelated bits. This would make the reduced files unnecessarily more complex.

We also use another solution to prevent the problem of RQ3: we group lines of covered code together. Lines with less than 30 (uncovered) lines in between them are grouped together. For instance, if lines 100 and 101 are covered, and lines 120 and 121, they fall in the same group. However is there are more than 30 lines of code between the two subsequent blocks, seperate groups are made for both blocks. We chose a limit of 30 lines because it is reasonable to assume that lines with less than 30 lines of code in between them are related. Some style guides for C recommend not making functions longer than a page long [22]. With a 30 line limit, it is likely that the lines are still in the same function when grouping together. The number is configurable however, if a different number is desired.

As an example, if both lines 100-102 and lines 300-302 are additionally covered in a given compiler source file, we make two groups of covered code blocks. We then first reduce based on the first group, so lines 100-102. After that, we check whether lines 300-302 are also covered by the reduced file. If they are, then the covered lines were probably related and we do not reduce again for lines 300-302. However, if lines 300-302 are not covered after reducing the first time, then we make another version of the reduced file which specifically covers lines 300-302. Together with compiling per compiler source file, this addresses the problem of RQ3.

An example of related covered lines which are fairly far apart is in the Gcov output shown in Listing 3.6.

```
...
 1: 2760:           next();
 1: 2761:           switch(t) {
...
 1: 2884:           default:
 1: 2885:               if (tcc_state->warn_unsupported)
...
```

Listing 3.6: Gcov output with related lines that are far apart

After line 2761 the next covered line is 2884. These lines are somewhat far apart. Despite this, the lines are related because line 2884 is the `default` case for the switch statement from line 2761. Our tool does not reduce twice based on the two groups of covered lines, but only once. After reducing for lines 2760-2761, lines from 2884 onwards are also covered by the produced file. This is how our tool determines that reduction is not necessary for the second group of covered lines.

3.8 Reducing larger projects

The example of Listings 3.2 and 3.3 are about reducing a single file. To achieve our goal of using an open source project as input, our tool should also work for larger projects with many files.

Our tool automatically reduces all the files in a project that has additional coverage and puts them in an output directory with a structure as shown in Listing 3.7. Our tool copies the directory structure of the project to reduce, and puts the reduced files in their original position in the directory structure.

```
1  smaz
2  +-- smaz.c
3  |   |-- smaz_tccpp_reduced_0.c
4  |   +-- smaz_tccpp_reduced_0.c_cov
5  |       |-- i386-asm.c.gcov
6  |       |-- libtcc.c.gcov
7  |       |-- ...
8  +-- smaz_test.c
9      |-- smaz_test_tccpp_reduced_0.c
10     +-- smaz_test_tccpp_reduced_0.c_cov
11         |-- i386-asm.c.gcov
12         |-- libtcc.c.gcov
13         |-- ...
```

Listing 3.7: Directory structure of tool output

The example of Listing 3.7 is output for a reduced project which has 2 C files in it: `smaz.c` and `smaz_test.c`. The position of the C files is copied over to our output directory structure, but they are now directories. The name of these directories still end with `.c`, but they are not C files. Inside each of these directories, the reduced files reside, along with directories containing the coverage data for the reduced files. The filename of the reduced file `smaz_tccpp_reduced_0.c` is built up as follows: `smaz` means that the original file was called `smaz.c`. `tccpp` means that the compiler source file the coverage analysis was based on was `tccpp.c`. `reduced` speaks for itself, and the `0` at the end means that this was the first reduction for this particular file. If a different group of covered lines were found that was not covered by this file, the tool would reduce another time, based on the other group of covered lines. The next output file would have a `1` at the end. This number increments for every independent group.

With this approach, our tool can reduce large projects with complex directory structures. For example, our tool can reduce the source code of TCC itself.

3.9 Filtering results

While experimenting with an unfinished version of our tool, we observed that some results relied on GCC extensions. We do not want to make tests out of these files, because our test suites only test for compliance with language specifications. We filter these files by using GCC at the end of our tool pipeline. If GCC cannot compile the file with extensions disabled, we discard the file as output.

3.10 Making tests

The final step in our process is to make proper tests out of the output files from our reducer. These files are not proper tests yet. We cannot include them in our test suites as-is, because we should define what the behaviour of the file should be. This can be done by adding assert statements ensuring correct behaviour of the program. At the moment, constructing tests out of the tools output files requires manual work.

Chapter 4

Results

In this chapter, we present a selection of the results of our experiments. In Section 4.2, pieces of code that we got as output from our reduction tool are shown and discussed. These are the 'Reduced C source files' as shown in the overview of Figure 1.2. While we got more results out of our tool than the results that we show here, the other results do not differ significantly enough to discuss them separately. Overall, we only found a handful of truly different results from the project that we gave as input for our tool. For some of these, we want to build a test for inclusion in our test suite, but for others we do not want to do this; for instance, this is the case for a file that relies on a GCC extension. This is not standard C, and we do not want to test it with our test suites.

A final step is still needed to reach our final goal: we need to make tests out of suitable pieces of code. This is a step that still requires manual work at the moment. We will discuss whether it might be possible to automate this process in Chapter 5.

4.1 Experiment process

Our experiments are conducted in the following manner:

1. Coverage data of the SuperTest suite is generated using Gcov
2. Source code of an open source C project is downloaded from the internet
3. The tool is run on the C project and the SuperTest coverage data
4. The tool reduces the files in the C project and puts the output in a separate directory

The process is repeated for several open source projects. These open source projects are our input, and our tool produces reduced C source files following the process shown in 1.2. After the last step, we analyse the reduced C files one-by-one and determine whether we want to make tests out of them.

4.2 Tool output

The pieces of code that we show in this section were chosen because they give an overview of what types of results we get out of our tool. We explain for every piece of code why it is interesting, for instance because we would want to make a test out of it (i.e. Listing 4.1), or because it is not useful as a result but expected output for a specific reason (i.e. Listing 4.12).

Examples of results that were not interesting to us were reduced files that relied on `asm` sections containing assembly code. Assembly is target-specific, while the SuperTest suite should work for any target. So, this result is explainable, but it is not useful. SuperTest does not cover it because it is not desired to have tests using these assembly statements. We do not show these types of results. In this Chapter we focus on results that are interesting for one reason or another.

For the results that we do want to make tests out of, manual adaptation is required in order to build such a test. We need to at least alter the files so that they have a main function. Without a main function, the test will not be executable after it is compiled. We also need assert statements to ensure that the behaviour of the program is correct. When there is no real behaviour in the reduced file, we need to add behaviour to make a proper test.

The reason why some files have no behaviour is because all the code with behaviour of the original file was removed by the reduction process. So it is likely that the original file did have behaviour.

The following examples of output are generally quite exotic pieces of code. This is expected, however, since the idea is that our tool outputs certain constructions and pieces of code that we 'missed', or didn't think about, when building the tests we made ourselves (by hand).

4.2.1 Specific keyword combination

```
1 extern inline void a() {}
```

Listing 4.1: `extern inline` keyword combination

In the result of Listing 4.1, a combination of keywords is used that we do not yet test for in our test suites, namely `extern inline`. This is a useful result to make a test out of.

To make a test out of this file, it needs to become executable; it thus needs a main function. We could call function `a` in this main function, and assert that the function is executed by altering the value of a global variable in function `a`. The updated value of this global variable is then checked for in the main function, after the function call. This way we ensure function `a` is called and executed when running the test program. The test code is shown in Listing 4.2.

```
1  #include<assert.h>
2  int glob = 0;
3
4  extern inline void a() {
5    glob = 1;
6  }
7
8  int main(void) {
9    a();
10   assert(glob == 1);
11   return 0;
12 }
```

Listing 4.2: Testing the `extern inline` keyword combination

4.2.2 Preprocessor directive

```
1 # 1
```

Listing 4.3: File with a single preprocessor directive

We see that a lot of files are reduced to the same thing: a single preprocessor directive `# 1` as seen in Listing 4.3. All the tests in our test suites start with a multi line comment with copyright information, meaning the files never start with a preprocessor directive such as in this file. This would make an interesting test to incorporate in our suites. We could add a simple `main` function with an `assert(1)` statement to make the file executable.

4.2.3 File ending with a single line comment

```
1 //
```

Listing 4.4: File with only a single line comment

A single line comment like ending in an End Of File (EOF) character, as seen in Listing 4.4, covers one specific line of compiler code. The tests in our test suites never end in this way. This makes it an interesting find. We should make a test ending with a `//` comment. We could, again, turn this file into an executable test by simply adding a `main` function with an `assert(1)` statement.

4.2.4 Complex but valid syntax

```
1 void a() {
2   ({
3     b:;
4   });
5 }
```

Listing 4.5: File with complex syntax but no output

The piece of code from Listing 4.5 does not have any behaviour. It is just a function with a very specific syntax. The `b:` on line 3 is a label, but it has no code to execute after it. The file does have additional coverage, so we do want to include a test based on it in our test suites. The original file (before reduction) did have behaviour, but the tool reduced it to a file without any. We should add behaviour to test in our test file.

A potential test is shown in Listing 4.6. In this test file, function `a` is called once, and inside the function the program jumps to the `b` label by means of a `goto` statement. After that, the value of the global variable `glob` is changed. Finally, it is asserted in the main function that the `glob` variable was updated succesfully in the `a` function.

```
1  int glob = 0;
2
3  void a() {
4    goto b;
5    ({
6      b:
7      glob = 1;
8    });
9  }
10
11 int main(void) {
12   a();
13   assert(glob == 1);
14 }
```

Listing 4.6: Test file with complex syntax but no output

4.2.5 Variable number of arguments with double type argument

```
void a(double b, ...) {}
```

Listing 4.7: File with double type parameter in combination with ellipsis

The `...` syntax from Listing 4.7 means that the function may take a variable number of arguments. This ellipsis in combination with the preceding parameter with the `double` type, covers some compiler source code that our test suites do not. Apparently, when these types of parameters are arranged in this manner, a specific bit of code from the compiler is used.

A test based on this file is shown in Listing 4.8. The `a` function is called from the main function, with a few arguments after the first argument of the `double` type. In the `a` function, there are `assert` statements that check the values that are passed to it.

```
#include<assert.h>
#include<stdarg.h>

void a(double b, ...) {
  assert(b == 1.5);

  va_list valist;
  va_start(valist, b);

  assert(va_arg(valist, int) == 2);
  assert(va_arg(valist, int) == 3);
  assert(va_arg(valist, double) == 2.5);

  va_end(valist);

  return;
}

int main(void) {
  double c = 1.5;
  a(c, 2, 3, 2.5);
  return 0;
}
```

Listing 4.8: Test file for double type parameter in combination with ellipsis

4.2.6 Binary constant

```
int main() { int a = 0b010101010101; }
```

Listing 4.9: File with binary constant

The integer literal starting with the `0b` characters, from Listing 4.9, is syntax for specifying binary constants. This is a GCC extension and not standard C [23] [24]. It is therefore not something we want to test with our test suites.

4.2.7 Single instruction, multiple data processing

```
1 int a();
2 int main() {
3   a("", 12.34 < 56.78, 12.34 <= 56.78, 12.34 == 56.78, 12.34 >= 56.78,
4     12.34 > 56.78, 12.34 != 56.78);
5 }
```

Listing 4.10: Reduced file covering SSE compiler source code

The piece of code from Listing 4.10 uses a specific x86 instruction set expansion called Streaming SIMD Extensions (SSE). This is an extension to perform the same operation on several pieces of data simultaneously. In this case, a floating point comparison instruction needs to be performed six times. The compiler optimizes this code to use the SSE capabilities of the processor.

Because the additional coverage is about an optimization part of the compiler, this is a construction that we do not test in our test suites yet. As discussed in Chapter 1, tests in our test suites are made by looking at the language standard, not if certain optimizations are triggered in the compiler. It would be useful to make a test out of this file, because ideally we do want to test these parts of the compiler code as well.

A test is shown in Listing 4.11. The correct outcomes of the floating point comparisons are asserted in the `a` function. The `char*` parameter of function `a` is needed to get the additional coverage.

```
1 int a(char* s, int a, int b, int c, int d, int e, int f) {
2   assert(a && b && !c && !d && !e && f);
3   return 0;
4 }
5
6
7 int main() {
8   a("", 12.34 < 56.78, 12.34 <= 56.78, 12.34 == 56.78, 12.34 >= 56.78,
9     12.34 > 56.78, 12.34 != 56.78);
10 }
```

Listing 4.11: Test for SSE code

4.2.8 Bitfield with long type

```
1 int a();
2 int main() {
3   struct {
4     long b : 5;
5   } c = {1};
6   a(c.b);
7 }
```

Listing 4.12: Result with bitfield used with a long type

Using a bitfield with a `long` type, as seen on line 4 of Listing 4.12 is allowed in C, but the behaviour is implementation dependent [25]. Because the behaviour depends on the compiler, we cannot test it

with our test suites. So, this result is not necessarily useful for the purpose of this project.

Chapter 5

Discussion

In this chapter we discuss the results as presented in the previous chapter. We categorise different results, and see which types of results are most interesting to us. We explain how the results can contribute to the goal of improving compiler test suites. Limitations of our research are also discussed.

5.1 Categorising the results

The results can be divided into three categories. The first category is files of which we cannot test the behaviour. This is for instance the case when the behaviour of the code is implementation dependent, such as the example in Listing 4.12. The second category is files which we do not want to include tests for in our test suites. This is for instance because a file relies on a GCC extension, such as the file from Listing 4.9. The third category is files of which the behaviour is testable, and we would want to include a test in our test suite. This is the case for the code from Listing 4.1. Also the other files, from Listings 4.3, 4.4, 4.5, 4.7 and 4.10, fall into this category.

5.2 Reducing several projects

While we applied our tool on several open source C projects, the most additional coverage, by far, was found by reducing the TCC source code itself. This additional coverage was mainly found in a large test suite for the TCC compiler that comes with the source code. This test suite presumably already covers edge cases and other specific parts of the source code. It might be interesting to reduce test suites that come with other open source compilers as well, because these might contain different edge cases and complex constructions.

5.3 Speed of the reduction

Because we find so few files with additional coverage (about ten files for the complete TCC source code, but generally only one or two for other large, complex projects), the speed of the reduction process is not vital to us. C-Reduce works well because it produces smaller output compared to other tools, and when paired with a tool like Clang-Tidy this output is still generally readable. Because output needs to be analysed by hand to potentially make a test out of it, it does not make sense to produce a large amount of reduced files in a short amount of time. The balance of C-Reduce, which takes longer but produces smaller output, is useful for the purpose of this project. Running the tool on the full TCC source code, it takes 22 minutes and 40 seconds to finish on an Intel Core i7-9750H machine. This is not a very long time considering TCC has around eighty thousand lines of source code.

5.4 Limitations

A limitation of our tool is that if a file to reduce is not accepted by Clang-Tidy, the reduction cannot be done. The reason for that is that in the property script for C-Reduce, we defined that Clang-Tidy should not emit any warning for the reduced file. However, if Clang-Tidy throws a warning for the file before reduction, the property does not hold at the start and C-Reduce cannot reduce it. Sometimes

we would still like to reduce the file, if we do not care about the specific warning Clang-Tidy produces. This can be done by disabling the warning in the property script, but this requires manual work, and for other files we might still want Clang-Tidy to emit this warning. It is not possible at the moment to reduce files in a completely automatic way if Clang-Tidy throws an error before reduction has started.

Another limitation is that we need code as input for our tool. While the amount of code available to use is virtually endless thanks to online repositories such as GitHub [3], not many projects proved to yield interesting results. Most of the results shown in Chapter 4 come from a TCC test suite which comes with the source code for the compiler.

5.5 Automating the construction of tests

At the moment, constructing tests out of the reduced files is manual work. The process is to first make the file executable by adding a main function. Then, we need to ensure that all code in the file is reached when executing it. Finally, we need to add `assert` statements to check that the behaviour is correct. Determining what the behaviour should be can be done by looking at it manually. For instance, in Listing 4.10, we need to determine what the outcome of every comparison expression is.

Because we need to be sure that our tests are correct, if we would partially do this automatically we would still want to check the results before we include the tests in our test suite.

To make tests out of the reduced C files, the C standard needs to be checked to determine the correct behaviour. However, the different C standards are often not clear in such a manner that the supposed behaviour of tests can be determined automatically [26].

Also, if we would want to fully automate this process, the standard documents needs to be analysed automatically somehow and transformed into formal 'rules'. Letting a program analyse a document automatically in this manner would be a very complex task. Transforming a language standard document into formal rules will therefore very likely stay manual work. We therefore think that fully automating the construction of tests is not feasible at the moment.

Chapter 6

Related work

6.1 Automatic compiler test generation based on attribute grammars

A survey of compiler test generation methods was done in 1997 by A.S. Boujarwah et al. [27]. If attribute grammars are used for generating test cases, complete semantic coverage is theoretically possible. Attribute grammars were invented in the 1960s by Donald Knuth [28]. With attribute grammars, information about behaviour of a test case can be embedded in the grammar definition.

An attribute grammar for C99 was developed by V. David et al. [29]. This grammar is large in size and relies on extensions to the attribute grammar language. It consisting of about 340 rules along with 90 attributes. It is, however, not publicly available.

A.G. Ducan et al. developed a way to generate semantically correct test cases using attribute grammars, concluding that it works well [30]. Both input and correct output are generated with their method: not only could a compiler input file be generated, but also its expected output.

6.2 The test oracle problem

Chen et al. did a study in which they compared different compiler testing techniques [31]. Before comparing different methods, they recognize a problem in automating compiler test generation: how can we know what exactly the behaviour of a generated test case should be? This is referred to as the 'test oracle problem'.

Several techniques for working around this problem are discussed. The first is Randomized Differential Testing (RDT), which compares the output of two different compilers on a given input test case. If the output of the compiled programs differ, it is assumed there is a bug in one of the compilers. However, it could also be the case that the behaviour of the compiler output is specified in the language standard to be implementation-defined. Another problem is that two compilers could give the same output which is incorrect. In this case, no bug would be detected.

For our research, RDT does not suffice because it is not guaranteed that the results are correct. We would include a generated test in our test suites for a long time, and therefore we need to be sure that it is correct. If the behaviour defined for the test case is incorrect, we would get false positives in testing compilers for bugs. This is unacceptable in our case.

A variant of RDT is Different Optimization Levels (DOL), where the output of different optimization levels is compared (instead of output from two different compilers). However, it suffers from the same issues as RDT.

Another method of generating compiler tests is called Equivalence Modulo Inputs (EMI). Here, the test oracle problem is addressed by comparing different test programs of which it is known that the behaviour should be the same. However, again this approach suffers from the problem that a compiler could give incorrect output for both variants, in which case a bug could slip through.

6.3 Random C program generation: CSmith

A random C program generator called CSmith was developed by Regehr et al. [32]. While it does generate valid programs, compiler bugs are only 'detected' by compiling the test case with two different compilers and checking whether the output is the same. If it is not, it is assumed that there is a bug in one of the compilers. As mentioned, this does not suffice for our purposes. It becomes especially problematic when CSmith produces output of which the language standard states that its behaviour is implementation-defined. Two different compilers are then allowed to differ in the output they produce. It requires manual work to determine whether CSmith produced a program with implementation-defined behaviour. CSmith uses static analysis means to avoid unspecified and undefined behaviour, which is a similarity to our approach.

6.4 Test generation in functional programming

For the functional programming language Haskell, a tool called QuickCheck was developed to automatically generate tests [33]. QuickCheck is open source. Because in functional languages properties can be defined that are precise and rely only on their explicit input, generating random tests works well for it. While C is not a functional language, studying QuickCheck's approach to test generation could be useful when writing a test generator for the C language.

Chapter 7

Conclusion

7.1 Summary

The goal of our research is to see if we can improve the quality of C compiler test suites by constructing tests using a different process than the process that we use currently. Secondly, using our new process we also investigate whether our current test suites are lacking in source code coverage of a compiler or that it is already quite thorough in this aspect.

If we find a lot of new coverage with our input of open source projects, we can conclude that our test suites apparently do not cover all parts of the compiler very well. On the other hand, if we do not find a lot of new coverage, this means that our test suites already covers a lot of compiler source that real-world applications use.

Our approach can be summarised as follows. We take the source code of a C compiler (in our case, TCC [8]), and compile it with added instrumentation for code coverage analysis. For this instrumentation and analysis, we use the Gcov [7] tool.

After this compilation with instrumentation, we compile one of our test suites and determine the lines of compiler source code that are covered by this compilation. We now have the data to compare against when compiling an open source real-world application. This is the next step: compiling of a real-world application and generating the compiler source code coverage the data for this application.

The next step is to compare the coverage data from the test suite against the coverage data from the real-world application. If we find that the application covers parts of the compiler code that test suite does not, we determine exactly which lines of code are additionally covered. We developed a Python script that generates this data.

Using the output of our script, we now throw away parts of the source code of the real-world application that do not contribute to the additional coverage that was found. We also ensure that the code remains compileable. We end up with a small C source code file that exhibits the additional compiler coverage.

For this reduction of the source code, we use a tool called C-Reduce [4]. We developed very specific scripts that make our goals achievable with C-Reduce. C-Reduce was not developed to be used in the way we need it to for our research, but because works in a very general manner and is configurable quite specifically, we were able to use it to achieve our goals. Our contributions here are the development of specific script files that make C-Reduce reduce in a manner that we need it to for our research.

One specific issue that could arise during the reduction that we need to address, is that undefined behaviour could be introduced while reducing. We prevent this by using the Clang-tidy tool [17], which can detect forms of undefined behaviour.

When a reduced C source file is produced, one final step is needed to achieve our goal of improving our test suites: we need to make a proper test out of the reduced file. This requires manual work at the moment. It mainly involves determining the expected behaviour of the file by looking at the C language standards.

Overall, we consider our research to be successful. We succeeded in achieving our final goal of making tests that we do not yet have in our test suites. Our developed approach works.

7.2 Answering the research questions

In this section, we discuss our findings for each research question individually.

7.2.1 RQ1: How can we reduce C programs that exhibit compiler coverage which is of specific interest, to a program that is smaller but still has this coverage?

The first part of our approach, addressing the problem of RQ1, is that we can reduce C programs based on compiler source code coverage with a combination of the C-Reduce [4] and Gcov [7] tools. We use a Python script to automate this process, and write a specific property script for C-Reduce that is based on data of covered compiler source code. This coverage data is produced by the Gcov tool. Alternative reduction tools or coverage tools could in principle also be used. These are discussed in Chapter 2.

7.2.2 RQ2: How can we make sure that such a reduced program does not exhibit undefined or unspecified behaviour?

The next part of our approach addresses the problem of undefined or unspecified behaviour that could be introduced by the reduction process. We prevent this by using an analysis tool to catch this kind of behaviour. We used the Clang-Tidy [17] tool, as discussed in Chapter 2. We add a rule to the reduction tool property script, where it is checked by Clang-Tidy whether a produced variant exhibits undefined/unspecified behaviour. If such behaviour is found, the variant is rejected and the reduction tool goes on to reduce a variant on which the Clang-Tidy check does not fail. So the solution to our problem is to use a rule in the reduction tool property script, which should entail that undefined and unspecified behaviour checks of a static analysis tool pass. Not all forms of undefined behaviour and unspecified behaviour can be detected by these tools, but Clang-Tidy does help preventing ones that are commonly introduced by the reduction process.

7.2.3 RQ3: How can the problem that different additional coverage is found by unrelated lines of code be addressed?

The third part of the approach addresses the question of how we can resolve the issue of an application covering unrelated parts of compiler source code. This problem is resolved by grouping lines of compiler source code that were covered by the application. We then apply the reduction process several times, on a per-group basis. Covered lines that are in different compiler source files are always considered separate groups.

7.2.4 RQ4: How can we make a test (appropriate for inclusion in a test suite) out of the reduced program?

Making a test out of a reduced file still requires manual work. As discussed in Chapter 5, we do not see a practical way in which this process can be automated. To make a test, the behaviour of the code should be checked with a C standard.

7.3 Future work

7.3.1 Reducing generated C files

We are interested in seeing whether reducing C files generated by the CSmith [32] would yield useful results. CSmith is a tool that generates random C programs that conform to the C99 standard. The tool was built to test C compilers with random input. We might be able to use this tool in a useful manner in the context of our work. We could generate a certain number of C files in a specific folder with CSmith, and then reduce this folder as a C 'project' with our tool. Because CSmith is random, it might generate files that are quite exotic and with constructs that we missed while building our existing tests manually. It would be interesting to see whether it generates many files with additional coverage compared to our test suites. With this approach, we might be able to overcome the limitation that the amount of interesting source code we can use as input for our tool is limited, as discussed in Chapter 5.

7.3.2 Test generation based on an attribute grammar

With an attribute grammar, it is possible to generate test cases along with information about what the behaviour of the test cases should be [30]. However, we did not find a publicly available attribute grammar for the C language. One has been developed by V. David et al [29], but it relies on extensions to the attribute grammar language to achieve full coverage of the language. This extended attribute grammar took around 1-2 man-month to develop. If we could develop an attribute grammar for the C language as well, the benefits could be quite large: we can automatically generate tests along with what their correct behaviour should be. We could use these generated tests together with our tool to find tests with coverage that our test suites do not have. However, we do not know if the process developed by Duncan et al. [30] is compatible with the extensions to the attribute grammar language as used by David et al. [29]. This will need investigation.

One thing to note is that the attribute grammar developed by V. David et al. is large and quite slow, due to ambiguities in the C language. Resolving ambiguities in the `stdio.h` header file takes 75 seconds on a 3GHz processor. This was in 2005 however, so developments in processor technology might reduce this duration significantly. Also, the purpose of their research was to disambiguate C files. Generating tests might be a simpler task which could be less computationally expensive. Generating tests using this approach could be worth investigating.

7.3.3 Making our tool work on different compilers

Our tool still only works for TCC. Potential future work could focus on expanding our tool to work with any, or at least some other compiler. Making it work for GCC would already be quite a bit more complex, because there are many more compiler source files than with TCC. These source files are in a more complex directory structure with subdirectories as well. TCC does have all its source files in a single directory, without having files in a tree of sub directories.

Bibliography

[1] *GCC Soars Past 14.5 Million Lines Of Code*, https://www.phoronix.com/scan.php?page=news_item&px=MTg3OTQ, Accessed: 2020-06-22.

[2] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing", *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[3] *Github*, https://github.com/, Accessed: 2020-09-15.

[4] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs", in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 335–346.

[5] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction", in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 361–371.

[6] G. Gharachorlu and N. Sumner, ": Priority aware test case reduction", in *International Conference on Fundamental Approaches to Software Engineering*, Springer, 2019, pp. 409–426.

[7] *Gcov*, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, Accessed: 2020-06-16.

[8] *TinyC compiler web page*, https://bellard.org/tcc/, Accessed: 2020-06-22.

[9] A. Almomany, A. Alquraan, and L. Balachandran, "Gcc vs. icc comparison using parsec benchmarks", *IJITEE*, vol. 4, no. 7, 2014.

[10] *Gnu: What is free software?*, http://www.gnu.org/philosophy/free-sw.html, Accessed: 2021-01-17.

[11] *Intel oneapi base toolkit documentation*, https://software.intel.com/content/www/us/en/develop/documentation/get-started-with-dpcpp-compiler/top.html, Accessed: 2021-01-17.

[12] *Msvc documentation*, https://docs.microsoft.com/en-us/cpp/build/reference/compiling-a-c-cpp-program?view=msvc-160, Accessed: 2021-01-17.

[13] *llvm-cov - emit coverage information*, https://llvm.org/docs/CommandGuide/llvm-cov.html, Accessed: 2020-06-25.

[14] P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti, "Experience report: Ocaml for an industrial-strength static analysis framework", *ACM Sigplan Notices*, vol. 44, no. 9, pp. 281–286, 2009.

[15] C. Ellison and G. Rosu, *An executable formal semantics of c with applications. university of illinois*, 2011.

[16] *GNU Compiler Collection (GCC)*, https://gcc.gnu.org/, Accessed: 2020-06-16.

[17] *Clang-Tidy*, https://clang.llvm.org/extra/clang-tidy/, Accessed: 2020-06-16.

[18] "Programming languages — C", International Organization for Standardization, Geneva, Switzerland, Standard, 1990.

[19] "Programming languages — C", International Organization for Standardization, Geneva, Switzerland, Standard, 1999.

[20] *GCC documentation: Conditionals with Omitted Operands*, https://gcc.gnu.org/onlinedocs/gcc-4.5.0/gcc/Conditionals.html, Accessed: 2020-07-03.

[21] *Analyzer: Remove -wanalyzer-use-of-uninitialized-value for gcc 10*, https://gcc.gnu.org/pipermail/gcc-patches/2020-April/544726.html, Accessed: 2020-06-26.

[22] *C code style guidelines*, https://www.cs.swarthmore.edu/~newhall/unixhelp/c_codestyle.html, Accessed: 2020-10-20.

[23] A. T. Committee, I. J. 1. W. Group, *et al.*, "Rationale for international standard, programming languages", C. Technical Report 897, rev. 2, ANSI, ISO/IEC, Tech. Rep., 1999.

[24] *Binary constants (Using GNU Compiler Collection (GCC))*, `https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html`, Accessed: 2020-07-15.

[25] "Programming languages — C", International Organization for Standardization, Geneva, Switzerland, Standard, 2011.

[26] *Defect report summary for C11*, `http://www.open-std.org/jtc1/sc22/wg14/www/docs/summary.htm`, Accessed: 2020-09-14.

[27] A. S. Boujarwah and K. Saleh, "Compiler test case generation methods: A survey and assessment", *Information and Software Technology*, vol. 39, no. 9, pp. 617–625, 1997.

[28] D. E. Knuth, "The genesis of attribute grammars", in *Attribute Grammars and Their Applications*, Springer, 1990, pp. 1–12.

[29] V. David, A. Demaille, R. Durlin, and O. Gournet, "C/c++ disambiguation using attribute grammars", *Project Transformers, Utrecht University, Netherland*, 2005.

[30] A. G. Duncan and J. S. Hutchison, "Using attributed grammars to test designs and implementations", in *Proceedings of the 5th international conference on Software engineering*, 1981, pp. 170–178.

[31] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques", in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 180–190.

[32] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers", in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[33] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs", *ACM Sigplan Notices*, vol. 46, no. 4, pp. 53–64, 2011.

Acronyms

AST Abstract Syntax Tree. 10
BNF Backus-Naur Form. 10
DOL Different Optimization Levels. 26
EMI Equivalence Modulo Inputs. 26
EOF End Of File. 20
GCC GNU Compiler Collection. 4, 7–10, 12, 15, 17, 21, 24, 30
ICC Intel C++ Compiler. 7, 12
IDE Integrated Development Environment. 7
MSVC Microsoft Visual C++. 7
RDT Randomized Differential Testing. 26
SGPR Syntax Guided Program Reduction. 10
SSE Streaming SIMD Extensions. 22
TCC TinyC Compiler. 7, 11, 12, 16, 24, 25, 28, 30