INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

Witness Modifications for Program Transformations

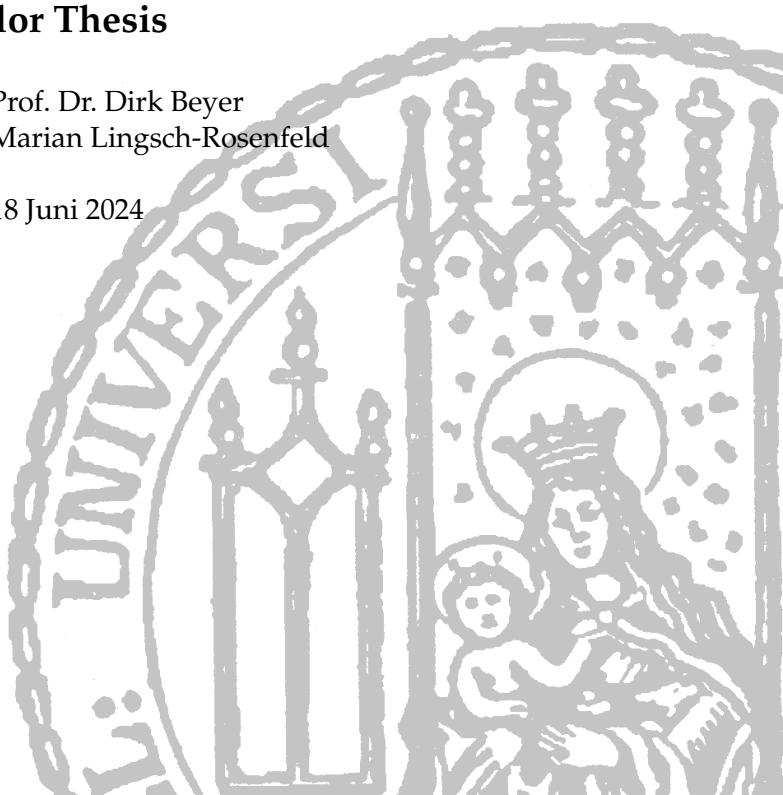
A Case Study on Side-Effect Removal

Anna Ovezova

Bachelor Thesis

Supervisor Prof. Dr. Dirk Beyer
Mentor Marian Lingsch-Rosenfeld

Submission Date 18 Juni 2024



Statement of Originality

English:

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments. I used ChatGPT and Deepl to generate and improve wordings of single sentences and small paragraphs, and to suggest small snippets of code for testing and evaluation purposes.

Deutsch:

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Ich habe ChatGPT und Deepl verwendet, um Formulierungen für einzelne Sätze und kleine Absätze zu erstellen und zu verbessern, und um kleine Codefragmente für Tests und Auswertungen vorzuschlagen.

München, 18 Juni 2024

Anna Ovezova

Acknowledgments

First of all, I would like to thank Prof. Dr. Dirk Beyer for the opportunity to write this thesis at the Software and Computational Systems Lab. I would especially like to thank my mentor, Marian Lingsch-Rosenfeld, for his outstanding support, for taking the time for meetings, and for always being available to answer my questions via Zulip. Writing this thesis has given me a lot of new knowledge and skills. I really appreciate the experience I have gained and will use it in my future life journey.

Abstract

In the context of a rapidly evolving information technology world, software verification is of great importance, since it secures that software meets its intended purpose and performs reliably. The software verification process is constantly improving. Program transformation is one of the techniques that make verification better and faster and is very widely used. The most common are removing side-effects, function inlining and loop abstractions. While program transformations usually have a positive impact on verification, another problem arises when replaying the obtained results. To provide independent validation of the verification result, verification results are accompanied by witnesses, which aid in the replay of the verification. However, program transformation may affect witnesses, making independent validation of verification results unreliable. In this thesis we describe an approach to the process of tracking modifications after transformation and adapting them to the witness.

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Motivation

The complexity of software being created is increasing day by day. The correctness and efficience of software are very important parts of software development. Software verification can be used to determine whether the software product under construction is being built to match its specification [15]. After software verification, to increase confidence the obtained results can be validated with reference to the verified program. A witness-based validation is a process of re-establashing of verification results to the needed visual format using verification witnesses. Witness is a set of verification results stored in a standardized exchange format [4]. It contains information about the outcome of program analysis, such as state of variables, invariants and error traces.
The process of software verification keeps improving every day by adding new techniqes and verification criteria. One of them are program transformations. They may involve changes such as adding new variables, loops or functions, which can alter the program structure, which in turn can be reflected in the verification results. Therefore, to ensure verification results stored in witness refer to the original program, transformations should be considered before the validation process. Verification witness should be adjusted according to the changes coused by transformation process. We call this process as witness backtransformation, which is the goal of this thesis.
To apply this thesis goal, it is neccessary to see what witness backtransformations are required. Consequently, some program transformation should be performed to create program modifications. As example of simple transformations we will analyse side-effects removal program transformations, which showcase the required witness backtransformations.

1.2 Overall Approach

Overview. Figure 1.1 shows a schematic overview of an approach. The first step is to transform the C program and document the applied modifications. In this step we will examine the side-effects in C programs and select those that are apropriate for implementation (steps 1 and 2 of Figure 1.1). The program transformation will be implemented using Coccinelle - a C program matching and transformation tool [12]. After it the witness will be generated for (step 3). A witness backtransformation

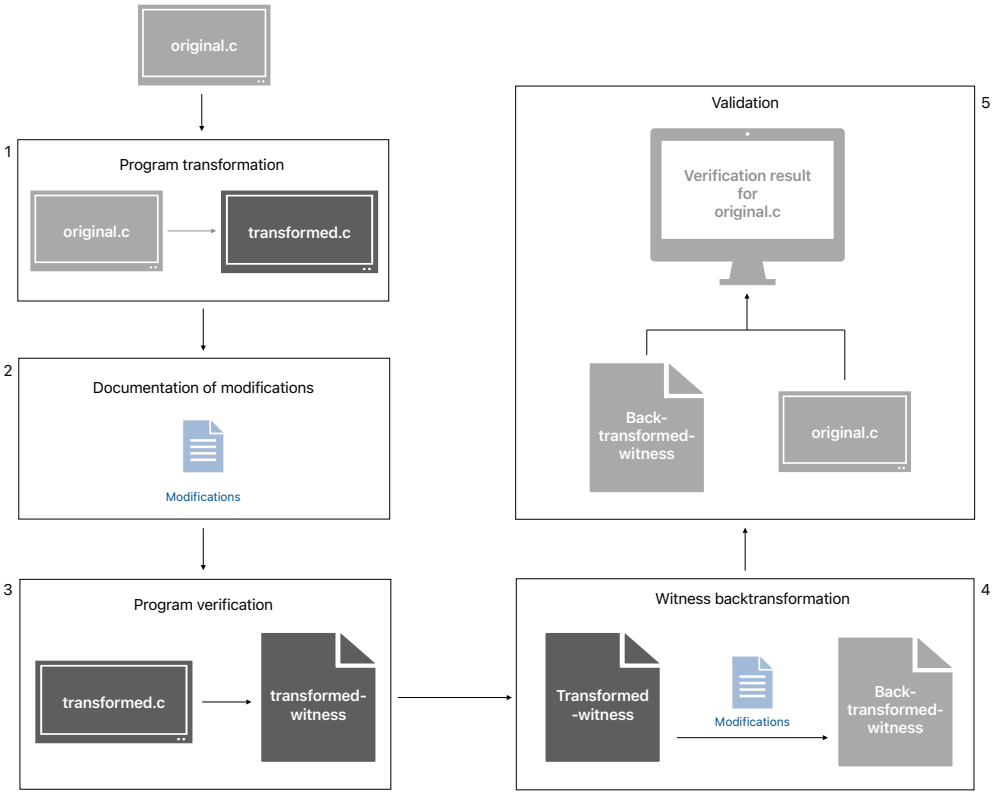


Figure 1.1: Schematic overview of the approach

is a goal of this thesis and is shown in step 4. The last part is the validation of the verification results using the backtransformed witness and the original C program (step 5), which is a part of thesis evaluation.

Example. To understand an approach better, let's look at a simple example. Table 1.1 and Table 1.2 show a C program before and after the transformation. The relevant parts of the witness generated by CPAchecker [6] shown next to the program code. To verify each while loop of the program, the loop invariants are created and stored in verification witness. As we can see, the start line of the second loop invariant has changed due to the addition of lines during the transformation. Consequently, the validation of this property will fail due to incorrect information about the loop invariant of the original program. The witness backtransformation algorithm mentioned above must verify and ensure that the data is correct for the original program.

Scope. The thesis focuses primarily on witness backtransformation. Therefore, it is more important to consider the different types of backtransformations that will be needed, rather than the efficiency of the side-effect removal algorithm or the amount of program transformations. In order to achieve the goal of this thesis, it is also crucial to consider the possibility of reusing or expanding of witness backtransformation.

```
1   void main() {
2     int x = 0;
3     int i = 0;
4     while (i++ < 10) {
5         x = x + 1;
6     }
7     while (i >= 10) {
8         i--;
9     }
10  }
```

Listing 1: C program before transformation

```
- invariant:
    type: "loop_invariant"
    location:
        line: 7
        column: 9
        function: "main"
    value: "(i == (10) ...)"
    format: "c_expression"
- invariant:
    type: "loop_invariant"
    location:
        line: 4
        column: 9
        function: "main"
    value: "(i == (1) ...)"
    format: "c_expression"
```

Listing 2: Witness.yaml before transformation

Table 1.1: State before transformation

```
1   void main() {
2     int x = 0;
3     int i = 0;
4     while (i < 10) {
5         i = i + 1;
6         x = x + 1;
7     }
8     while (i >= 10) {
9         i--;
10    }
11  }
```

Listing 3: C program after transformation

```
- invariant:
    type: "loop_invariant"
    location:
        line: 8
        column: 9
        function: "main"
    value: "(i == (10) ...)"
    format: "c_expression"
- invariant:
    type: "loop_invariant"
    location:
        line: 4
        column: 9
        function: "main"
    value: "(i == (1) ...)"
    format: "c_expression"
```

Listing 4: Witness.yaml after transformation

Table 1.2: State after transformation

Evaluation. The main purpose of the evaluation is to check the functionality of the algorithm. To do this, we will analyse whether a witness after execution of the backtransformation algorithm is valid for the validation process with the original C program and contains correct information about it.

2 Related work

After reviewing existing approaches, I was unable to find a tool that fully satisfies the problem of witness backtransformation. The transformation of witnesses is a specific task related to a specifically implemented verification witness structure. Therefore, I considered other related tools that partially refer to my approach.
Despite the large number of refactoring and compiling tools for C programming, only some of them are focused on program transformations, and even fewer on removing side -effects.

2.1 Program transformation tools

One program transformation approach is "A Unifying Approach for Control-Flow-Based Loop Abstraction", published by Dirk Beyer, Marian Lingsch Rosenfeld and Martin Spiessl in 2022 [7]. The paper presents a framework that allows the implementation of various loop abstraction techniques in one common environment with the ability to dynamically switch between them, selecting different abstractions using them. Loop abstractions are a technique used to abstract the behaviour of a program that contains a loop. This technique draws a parallel between the correctness of the abstract program and the original program. As part of my approach I want to achieve a similar relation to the original program by transforming it to one without side-effects using various techniques depending on the program specification. However, the adjusting framework does not meet the requirement of witness backtransformation, which is a necessary aspect according to my thesis requirements.

Another related tool is "A Haskell Implementation of a Rule-Based Program Transformation for C Programs" [20]. This algorithm translates Semantic Transformation Meta-Language (SMPL) rules into Haskell language to perform C program transformation. It has a number of instruments that are similar to those that are required in our approach. One of them is the SMPLanguage, that takes care of the syntactic and semantic conditions required to apply a given transformation [21]. Additionally, Haskell provides SYB (Scrap Your Boilerplate) libraries with powerful functions such as 'everywhere' and 'everything', which allow users to operate with complex data structures, such as abstract syntax trees or recursive data types. This approach makes process of transformation more reliable than changing code by pattern matching. However, the instrumentation of this tool does not satisfy the requirement of witness backtransformation, which distinguishes it from the one implemented in this thesis.

The most similar approach is a "A Post-Placement Side-Effect Removal Algorithm" [8], which introduces an algorithm for side-effect removal considering side-effects: a pure expression and a state-changing. This program uses syntax-directed transformations and symbolic execution to modify programs. The suggested implementation partially meets our requirements, which is the creation of a semantically equivalent program free of side-effects. However, only assignments to variables that occur during expression evaluation are considered. Additionally, semantic transformations need to be considered for witness backtransformation and should be implemented as a type of transformation. This related work does not consider them.

2.2 CIL

CIL (C Intermediate Language) is an infrastructure for the analysis and transformation of C programs [17], [16]. It has a set of tools that allow source-to-source transformation of C programs. It compiles all valid C programs into simpler C syntax with clean semantics. The main advantage of CIL is that it compiles all valid C programs into a few core constructs with very clean semantics. CIL also has a syntax-driven type system that makes it easy to analyse and manipulate C programs. So if we look at the process of compiling C to CIL we can see a lot of simplifying transformations, such as eliminating declarations of unused entities or relocation of global variables declarations from local to global scopes. One of the most significant features is sorting out of expressions, instructions and statements that contain side-effects. Additionally, compilation to CIL includes other transformations, such as loop transformations or removing unused declarations, variables, and inline functions.

CIL only performs analysis and transformation of C programs, which is similar to the transformation part of this thesis. It could be used for the transformation of C programs. However, while using the CIL side-effects removal algorithm, the information required for witness backtransformation is not provided.

3 Background

This chapter provides an overview of the theoretical aspects necessary for the implementation of the thesis.

3.1 Side effects in C programming

In contrast to functional languages such as Haskell, imperative languages traditionally used in high performance computing, such as C, can have side-effects [19]. The presence of these effects can influence the execution of parallel tasks, as well as the maintenance, understanding and debugging of programs. In order to detect them, it is essential to understand the concept of side-effects in relation to C programs, since we are only considering C programs in this thesis.
The official definition of side-effects is provided in the 'The GNU C Reference Manual' documentation [18]. Side-effects are essentially the externally-visible effects of running a program. According to the aforementioned source, a side-effect can be one of the following:

- accessing a *volatile* object
- modifying an object
- modifying a file
- a call to a function which performs any of the above side-effects

These are essentially the externally visible effects of running a program. They are called side-effects because they are effects of expression evalation beyond the expression's actual resulting value. In C, the order of computation is not fixed, which means that the compiler can perform the operations of the program in an order different to the order implied by the source of the program, provided that in the end all the necessary side-effects actually take place [18]. For example, in a statement like `x = f() + g()`, if `f()` or `g()` changes a variable on which the other depends, then the value of `x` may depend on the order of evaluation [10]. The compiler is also allowed to entirely omit some operations; for example, it is allowed to skip evaluating part of an expression if the compiler can be sure that the value will not be used and evaluating that part of the expression will not produce any necessary side-effects.
An approach to removing side-effects is presented in the C Intermediate Language [17], since this tool provides expressions without side-effects. The core idea of side-effect removal is to break down complex expressions with side-effects into statements without side-effects. For example, side-effects such as assignment, increment or decrement are moved out of the expressions into separate statements [16].

3.2 Software Verification Witnesses Version 2.0

In the context of program verification, a witness is an object explaining thr verdict of the verification task. They can be independently analyzed by witness validators to confirm or refute verification results [1]. The current witness format supports two types of witnesses: violation and correctness witness. Violation witnesses represent error paths that violate a specification. Correctness witnesses are witnesses for the cases when a verifier decides that a given program satisfies a given specification [3]. In my bachelor thesis I focus on corrctness witnesses.
The existing format of verification witnesses based on GraphML [5] has been identified as having certain deficiencies, including readability in text form and length. In response, a new format based on YAML [1] has been developed. This new format is more concise and well-structured, making it easier to fully support it by validators. Verification Witnesses Version 2.0 are represented by entries [1]. Each entry contains three key-value pairs: `entry-type`, `metadata` and `content`. The value of `entry-type` correspondents to the type of the witness: `invariant_set` for correctness witness and `violation_sequence` for violation witness. The key `metadata` describes the context of the witness, including the format version, creation time, information about witness producer and producer task description. The key `content` contains zero ore more `invariant` keys, which content is described in Table 3.1. The order of invariants in `content` is not important.

Key	Value	Description
`type`	`location_invariant`	The invariant type for arbitrary statements
	`loop_invariant`	The invariant type for iteration statements
`location`	mapping	The location of the invariant. Explained in Table 3.2
`value`	scalar	The actual invariant string, which is a side-effect-free C expression over variables in the scope where the invariant is placed
`format`	`c_expression`	Invariant is a c_expression

Table 3.1: Structure of the `content` part

The key `location` of the `loop_invariant` points to the first character of an iteration statement. The `location` of the `location_invariant` points to the first character of the statement or a declaration that is within a compound statement.

3.3 Coccinelle

Coccinelle is a program matching and transformation tool for C code. It will be used to remove side-effects according to predefined pattern matchings written in the semantic patch language [13].
This tool was first released in 2008 to facilitate the specification and automation in the evolution of the Linux kernel code [11]. As a tool, designed for updating the Linux

Key	Value	Description
`file_name`	scalar	The name of verificating file
`file_hash`	scalar	Hash value of the verificating file
`line`	scalar	Starting line of the invariant
`column`	scalar	Column number where invariant starts
`function`	scalar	Function in which invariant is located

Table 3.2: Structure of the `location` part

kernel device drivers, it supports a very large portion of the C language. Coccinelle was built around the existing notation for describing changes, the *patch* [14]. A patch is an extract of source code in which some lines are annotated with '-' or '+' symbols. It indicates whether the line should be removed or added, respectively. Coccinelle performs changes expressed as a semantic patch, an abstract form of patch that is not tied to specific lines in source code but rather to all relevant locations in the entire code base. Consequently, Coccinelle enables transformations to be expressed in semantic patch syntax, utilising pattern-matching rules, which may include scripts written in Python or OCaml, thus affording greater expressiveness.
Coccinelle provides a transformation language, SmPL (Semantic Patch Language), and an engine for applying SmPL semantic patches to C code [11]. These allow large-scale transformations to be performed, including changes to function, structure, variable names, function arguments, types of variables, the addition and removal of comments or other program structures, and many more.

Syntax of Coccinelle. Each SmPL semantic patch consists of a series of rules and a code pattern to match or transform. The rule name, surrounded by @ symbols, can be used for further use in other rules and is shown on the first line of Listing 6. By default, each rule is independent of one another; however, this can be altered by using the *depends on* keyword like shown in Listing 5. One rule can also contain more matching options using *disjunctions*, written as (... | ...) (Lines 7-11 of Listing 7). After the rule name the metavariable declarations are listed. There are different keywords to label metadata such as: `identifier`, `expression`, `statement`, `type`, `position`, `constant`, `binary operator`. They represent different program structures such as variable or function names, loop condition and loop body, type of the variable or binary operators. The examples of such declaration are shown in lines 2-4 of Listing 6. After metavariable declaration, separated with the @@ symbol begins the part with code pattern. Code patern part is shown in lines 7-9 of Listing 5. The part with transformations pattern consist of source language patterns, which identify the source language constructions to be altered, insertions and deletions(lines 10, 11, 22 of Listing 5), which mark the changes to be made [9]. The matching and transformation process is independent of any whitespace in the semantic patch, so there is no need of the additional rule declaration to handle unformatted cases in source code.

```
1       @rule1@
2       identifier p, v;
3       constant C;
4       type t;
5       @@
6       func() {
7           <+...
8           v = C;
9           ...+>
10          - t p;
11          + int p;
12      ...
13      }
14
15      @rule2 depends on rule1@
16      identifier r;
17      constant rule1.C;
18      identifier func;
19      @@
20      func() {
21          ...
22              + r = C;
23          ...
24      }
```

Listing 5: Coccinelle semantic patch for changing a variable type and assignment an existing variable a new value

```
1           @rule1@
2           identifier p;
3           identifier func;
4           expression E, X;
5           @@
6           func() {
7           <...
8           - for (E; X; p++)
9           + for (E; X; )
10          {
11          ...
12          + p = p + 1;
13          }
14          ...>
15          }
```

Listing 6: Coccinelle semantic patch for increment removing inside the *for* loop condition

As previously stated, semantic patches may include Python or OCaml scripts to enhance performance. Listing 7 shows an example for storing the variable name and line to a dictionary. The python script starts in line 13 and uses identifiers imported from the semantic patch as a variable.

```
@rule1@
type T;
identifier i;
constant C;
position p;
@@
(
 T i@p;
|
 T i@p=C;
)

@script:python@
p << rule1.p;
i << rule1.i;
t << rule1.T;
@@
import json
line = int(p[0].line)

result = {'type':t,
          'line':line}
```

Listing 7: Coccinelle semantic patch for saving of information about the variable

4 Implementation

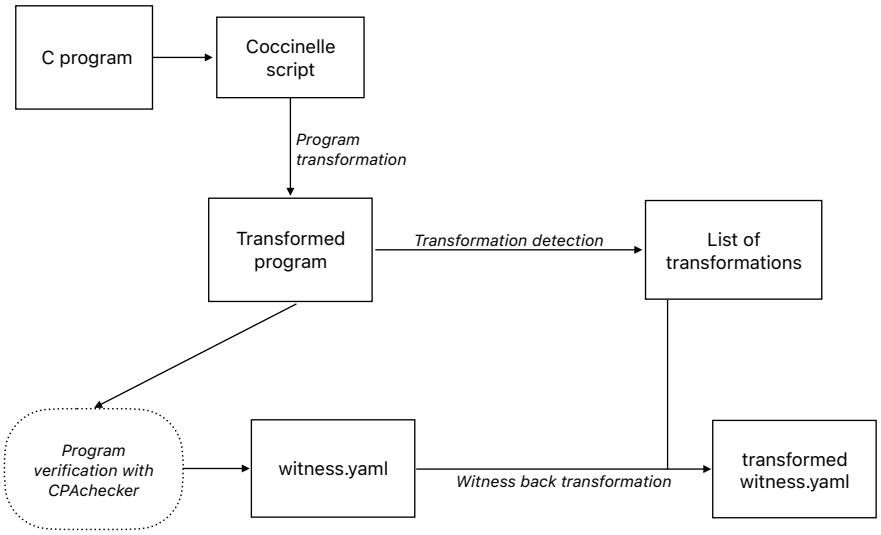


Figure 4.1: Workflow of witness backtransformation

Figure 4.1 illustrates a detailed overflow of verifying programs using witness backtransformation. It shows the steps that should be implemented in this section. In the first phase, *program transformation*, we detect side-effects in C programs and remove them using Coccinelle. The second phase is *difference detection*: during the transformation process, each modification is exported, including the type of the transformation and the order in which they were applied. The transformed program is then verified by CPAchecker, resulting in the creation of a witness version 2.0. The last phase is *witness backtransformation*: the list of exported transformations and the produced witness are used to handle differences, caused by program transformations in the first phase. The following sections will provide a detailed examination of each step.

4.1 Program transformations

The first step in the process is the removal of side-effects, which serves to establish a foundation for modification of a witness. We restrict ourselves to only certain but frequently occurring kinds of side-effects. Basing on the information, mentioned in

Sect. 3.1, we define the scope of side-effects, that will be removed to demonstrate the usefulness of the witness backtransformation algorithm. The overview of them is shown in Table 4.1.

Side-effect	Example
Increment operator	`int main() { int x = 0; int i = 0; while (i++ < 10) { x = x + i; } }`
Decrement operator	`int main() { int y = 0; int x = 10; while (x-- > 0) { y = y + i; } }`
The logical OR operator	`int main() { int x = 10; int y = 15; if (x == 10 || y == 15) { x = x + 1; y = y + 1; } }`
Assignment operator	`int main() { int a = 5; int b = 10; int c = 0; c = a + (b = 20); }`

Table 4.1: Side-effects scope

Increment, decrement or assignment operators modify the variable as a part of evaluation, so they may ocure state changing. In the case of the logical OR operator, the second operand will never be evaluated due to the *short-circuit evaluation* strategy, which is supported by the C language. Short-circuit is an evaluation strategy for the boolean operators, in which the second argument is evaluated only if the first argument does not suffice to determine the value of the expression [2].
Let us examine the algorithm for removing each type of side-effect from the selected scope. The choice of the scope of side-effects mentioned above is conditioned by the need to demonstrate different types of information that will be used during the witness backtransformation. This can lead to different approaches to witness backtransformation.

Increment and decrement. The algorithm for removing increments and decrements is dependent on the control flow constructs in which they occur, as well as their location in relation to the variable. In the case of the *while* loop, the algorithm is applied as shown in the Table 4.2, where `v` is the variable.

Before transformation	After transformation
`while (++v...) { ... }`	`v = v + 1; while (v...) { v = v + 1; ... }`
`while (v++...) { ... }`	`while (v...) { v = v + 1; ... }`

Table 4.2: Transformation strategy for pre- and postincrement within while loop

A comparable approach is used for the decrement in *while* loops.
In the case of the *for* loop, the location of the increment or decrement symbol in relation to variable inside the loop condition, is of no consequence, as the condition check occurs before the condition step. In contrast to *while* loop, in the *for* loop the condition step is executed after the execution of the loop body [10]. Hence, the algorithm for removal of increment and decrement in *for* loop is slightly different from the one used in the *while* loop. It is presented in the Table 4.3, where `E1`, `E2`, `I`, `D` are expressions with variable `v`, `I` is of the form `v++` or `++v` and `D` is of the form `v−` or `−v`.

Before transformation	After transformation
`for (E1; E2; I) { ... }`	`while (E1; E2; ) { ... v = v + 1; }`
`for (E1; E2; D) { ... }`	`while (E1; E2; ) { ... v = v − 1; }`

Table 4.3: Transformation strategy for in- and decrement within for loop

In all algorithms for increment and decrement removal, lines are added or deleted, resulting in one transformations type. We call them *syntactic transformations*. A further discussion of this topic will be presented in the following section.

Short-circuit evaluation. In the case of a logical *OR* operator, it is necessary to modify the structure in such a way that the second expression is evaluated. The algorithm was developed in the way illustrated in the Table 4.4, where `E1`, `E2` are expressions, and `S` is a statement.

Before transformation	After transformation
`if (E1 || E2) { S }`	`if (E1) { S } else if (E2) { S }`

Table 4.4: A transformation strategy for the addition of a new structure

In this case we observe the addition of a new program structure, indicating a *structurally relevant transformation*.

Assignment. In order to keep the state of the program unchanged without modifying the value of the variable, a new temporary variable is created and used at the assignment position. The detailed algorithm for different program structures is presented in the Table 4.5. In the abstract code `v`, `v_temporary` and `w` are variables of numeric type `T`; `C1`, `C2` and `C3` are constants. `structure()` can be a function, while or for loop. By creating a new variable `v_temporary` for local usage, the value of the global variable remains untouched. The implemented algorithm is not activated when the value of a local or loop variable is changed using the assignment operator. In this case we observe the addition of a new variable, which also indicate a *structurally relevant transformation*.

Before transformation	After transformation

`T v = C1;`
`T w = C2;`
`structure(...) {`
`  w = w + (v = C3);`
`}`
	`T v = C1;`
	`T w = C2;`
	`structure(...) {`
	`  T v_temporary = C3;`
	`  w = w + v_temporary;`
	`}`

Table 4.5: A transformation strategy for the addition of a new variable

4.2 Differences detection

To perform a proper transformation of the witness version 2.0, it is necessary to obtain the following information: the sequence in which the transformations were applied, the type of alterations made to the program, and specific details regarding location and nature of the transformations. To accomplish this, after each use of the transformation script, we check the modified program whether any changes have been made. Subsequently, in accordance with the category of changes to which the Coccinelle script belongs, the transformations are written out. All scripts are divided into several categories:

- *syntactic transformations* which track the insertion of new lines;
- *structurally relevant transformations* which track the addition of new structures;
- *structurally relevant transformations* which track the addition of new variables.

All modifications are defined by the scope of the C program transformations. This implies that each modification type requires its own method to export them. However, the difference detection methods sufficiently flexible to permit the number of these algorithms to be expanded. This can be relevant when adding new transformation types and related necessity to export new modifications. There are different approaches to the detecting of syntactic and structurally relevant changes. Both approaches are based on the comparison of the original C program with the transformed one, and storing needed attributes from them. A detailed explanation of the algorithms for detecting differences is presented in the next subsections.

Despite the fact that each transformation uses a different detection method, all exported modifications are saved in a single file in JSON format, which is an important artefact for further backtransformation algorithms. This file is called the *transformation protocol*. Each protocol consists of the list of sequence numbers, representing the order of the transformation performance, and additional information about the applied modifications. Currently, the transformation type can take one of the following values: `syntactic_transformations` for adding new lines without structural relevance, `new_variable_transformations` for adding new variables and `new_structure_transformations` for adding new structures.

Syntactic transformations. In order to create an algorithm for detecting differences, we consider the syntactic transformations from Table 4.2 and Table 4.3. As can be observed, two distinct types of line addition are declared: the addition of a single line above the existing structure and addition of the line within the structure itself. As mentioned in Sect. 3.2, a starting line of the loop indicates a location of the loop invariant in witness version 2.0. This makes the starting line of the loop that can be shifted to the most significant data modified by this changes.
The identification of differences is performed with the Coccinelle script for syntactic transformations. This is implemented in a way that the line numbers where modification should occur are stored in an external document. This information is used to maintain the correspondence between the lines that have been added to the modified program and the lines that have been deleted from the original program. To showcase the modification principle in details, a simple example of differences after syntactic transformation of the C program was created with `difflib` library in Python and shown in Fig. 4.2 and Fig. 4.3.

Line replacement				
Line before	Line after	Difference	Deletion	Addition
3	3		int i = 0;	
4	4			
-5	++5	5, 6 -> 5	- while (i++ < 10) {	+ while (i < 10) {
	++6			+ i = i + 1;
6	7		x = x + i;	
7	8		}	
8	9			
9	10		int r = 0;	
10	11		int t = 0;	
11	12			
-12	++13	13, 14 -> 12	- while (r++ < 10) {	+ while (r < 10) {
	++14			+ r = r + 1;
13	15		t = t + r;	
14	16		}	
15	17		}	

Figure 4.2: Syntactic transformations - line adding inside the structure

Line adding with invariant shifting				
Line before	Line after	Difference	Deleting	Addition
2	2		int x = 0;	
3	3		int i = 0;	
4	4			
-5	++5	-> 5	- while (++i < 10) {	+ i = i + 1;
	++6	6 -> 5		+ while (i < 10) {
6	7		x = x + i;	
	++8	-> 8		+ i = i + 1;
7	9		}	
8	10			
9	11		int r = 0;	
10	12		int t = 0;	
11	13			
-12	++14	-> 14	- while (++r < 10) {	+ r = r + 1;
	++15	15 -> 12		+ while (r < 10) {
13	16		t = t + r;	
	++17	-> 17		+ r = r + 1;
14	18		}	
15	19		}	

Figure 4.3: Syntactic transformations - line adding above the structure

The locations where the loop invariants could be are highlighted in green, as these are the loop start lines. In both cases, it is the process of counting down of new lines, increasing or decreasing the line number depending on whether the line has been added or replaced. The amount of new lines is also important information that should be stored. When adding two lines the line that points to invariant location is shifted.
The results are stored to the transformation protocol as `syntactic_transformations` type. An example of transformation protocol for syntactic transformations is shown in Listing 8. Next to the transformation type we see the list of line changes stored as a dictionary. The key represents the line number after the transformation, the value represents the line number before the transformation. In the example we can see that line 24 after the transformation corresponds to line 24 before the transformation. Line 36 corresponds to line 0, which means that it is a new line. This information is used in the witness backtransformation to check the invariant place transformations.

```
{
    "1": {
      "type": "syntactic_transformations",
      "info": {
        "line_changes": {
          "24": 24,
          "36": 0,
          "30": 29,
          "35": 0
        },
        "shift": 1
      }
    }
  }
```

Listing 8: Example of transformation protocol after syntactic transformations

Insertion of new variable. To create an algorithm for detecting differences, we consider the transformations from the Table 4.5. As this is one of the structurally relevant transformations, we need to keep track of changes, affecting the semantic of the program. To achieve this we implement a format to store structurally relevant information. It is represented as list of structures of the program, such as *variables*, *while* loops, *for* loops, *if* statements which contain *OR* operator in the condtion, *if* statements which contain *AND* operator in the condition, *if* statements with a simple condition and *if-else* statements. These are stored in JSON format before and after transformation. An example of such file is shown in Listing 9.

```
{
"file name": "file_name",
"constructs": {
    "variables": [],
    "while_loops": [],
    "for_loops": [],
    "if_else_statements": [],
    "if_statements": {
        "single": [],
        "or": [],
        "and": [] }
    }
}
```

Listing 9: Example of the contents of a program structure file

The key `variables` is representade as a list of variables of a C program. For each variable we export an additional information: type, value and declaration line. The implemented script recognises the following cases of variable declarations:

- variables of types:
  - integer
  - character
  - floating-point

- variables, declared with keywords:
  - const
  - volatile
  - auto
  - static
  - extern
  - unsigned
  - register

Due to time constraints some cases remain unimplemented: variables of type struct, arrays and pointers with initialisation.
To find a temporary variable created by *new variable transformation*, a list of variables should be compared in the program before and after the transformation. Once detected, information about the new variable is exported to the transformation protocol. An example of transformation protocol after assertion of new variable is shown in Listing 10.

```
{
    "1": {
        "type": "new_variable_transformations",
        "info": [
        {
            "new_variable": "b_temporary",
            "line": 24,
            "shift": 1,
            "new_value": "20",
            "old_value": "10"
        }
        ]
    }
}
```

Listing 10: Example of transformation protocol after insertion of new variable

Insertion of new structure. Since inserting a new structure is one of the structurally relevant transformations, the strategy for exporting changes is similar to that for inserting a new variable. The structure of the C program is saved before and after the transformation.
In order to find a new structure, created after the *new structure transformation*, we compare the structures of the original program, which is subject to change - information under the key `or` in the `if_statements`(Listing 9), and the structure of the transformed program correspondingly - the value of `if_else_statements`(Listing 9). As we know from the Chapter 3, invariant can be created when a loop is present. So we save the starting lines of the loops, which occur inside the `if` statement with `or` operator. In a similar way, the starting lines of loops inside the `if_else_statements` are saved as new structures. When inserting new structures, the column number is also taken into account. An example of the transformation protocol after inserting a new structure is shown in Listing 11. As we can observe, the

loop with condition `(r == 0)` has an additional starting line after the program transformation, which is the result of insertion of a new structure.


```
{
    "1": {
      "type": "new_structure_transformations",
      "info": {
        "before": {
          "(r == 0)": {
            "line": 24,
            "column": 9
          }
        },
        "after": [
          {
            "expression": "(r == 0)",
            "lines": [
              24,
              33
            ]
          }
        ],
        "shift": {
          "38": 9
        }
      }
    }
  }
```


Listing 11: Example of transformation protocol after insertion of new structure

Shift. After each transformation, lines are added to the program. If the modified construct is followed by other invariant checks that have not been modified, the line values at their locations will also not match those in the original program. Therefore, after each change, we write an additional `shift` key to transform back the locations of the unchanged invariants, as can be observed in Listing 8, Listing 10 and Listing 11. Despite careful testing of the algorithm's correctness, we cannot exclude that some edge cases were not considered. Also, due to the semantics of the language, there are many options for placing structures in relation to each other. My goal was not to cover all possible options, but those that are relevant to the scope of the chosen transformations.

4.3 Witness backtransformation

After saving the modifications, we can proceed to the final stage - backtransformation algorithm. In Fig. 4.4 we can see the detailed scheme of witness backtransformation.

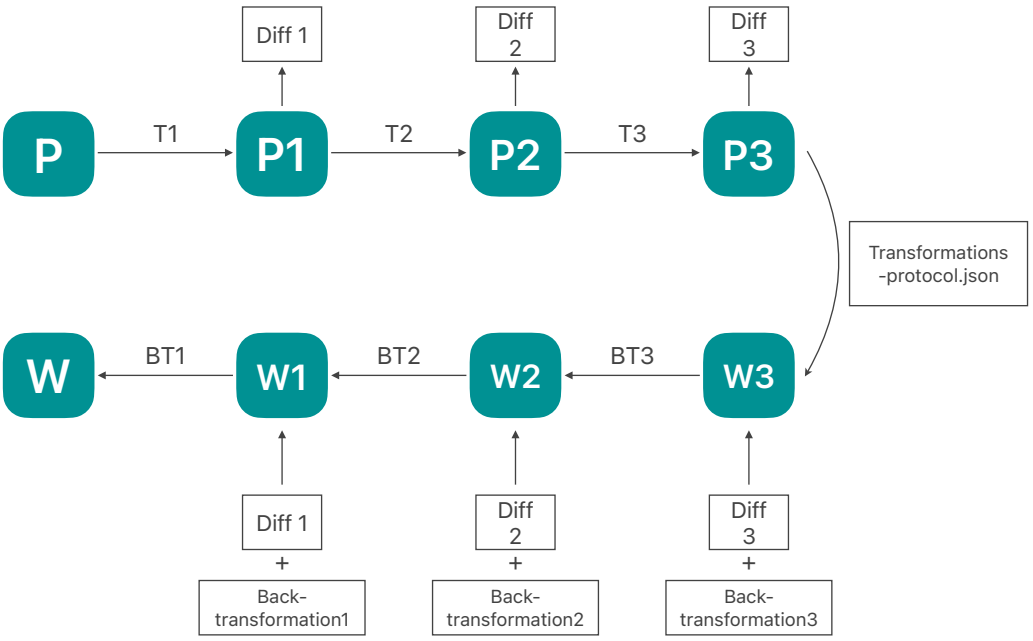


Figure 4.4: Witness backtransformation overflow

The process of backtransformation starts with parsing of invariants from the witness version 2.0. It is also necessary to parse the content of the transformation protocol. By reading the transformations in reverse order, we need to match the all invariants to each transformation. Each type of program modifications has its own backtransformation script.
For syntactic type changes, the invariant lines are updated according to the information in transformation protocol file.
In case of the insertion of the new structure, it may happen that, as a result of addition, a new invariant will be added to the existing one. During the backtransformation process it will be verified whether the invariant lines correspond to those stored in `after`(Listing 11) section of the transformation protocol. In the positive case, which means that the invariant was evolved in the transformation process, and the invariant lines will be changed to the lines stored in the `before`(Listing 11) section of the transformation protocol. Otherwise, this means, that the invariant was not involved in the exported transformation and only the shift value should be handled. The expression of the loop serves as an indicator that the lines that change belong to the same loop.
For modifications where a new variable is inserted, it should be taken into account that it is not only the location of the invariant that is changed, but also the value of the invariant. To avoid reassigning the value of variable `x` in the original program, a new variable `x'` was introduced during transformation. As a result, possible values of `x'` will appear in the value of the invariant. But the new variable `x'` does not exist in the original program, so it must be removed from the invariant. The value

of the invariant is then checked and the occurrence of the variable `x'` is changed to the value of the variable `x` from the original program. Also, since the original variable `x` could have two values due to the reassignment, it should be handeled in the transformed invariant value.
Lets consider the variable `v=C` was reassigned to `v=C'`. To avoid it, we create a `v_temporary=C'`. So, inspecting the invariant value after transformation, the occurance of `(v_temporary==C')` will be modified to `(v==C')`. Also, each occurance of `(v==C)` will be extended to `(v==C || v==C')`, as well as each `(v==C')` to `(v==C' || v==C)`.

4.4 Technical details

The project is avaliable as external `zip` folder that accompanies this thesis.
The algiorithm is implemented in Python programming language, version 3.9 with usage of following libraries:

- standard libraries of Python `subprocess` and `os` for commmand line executions;
- `json version 2.0.9` and `ruamel.yaml version 0.18.6` for exchange file parsing and changing;
- `re version 2.2.1` for Regular Expressions;
- standard library of Python `shutil` for file management.

The transformations of C program is executed with Coccinelle 1.2 compiled with OCaml version 5.1.0.
The project is structured in the manner illustrated above:

```
project_root/
├── evaluation/
├── c_trunk/
├── output/
├── temporary/
├── expected_c_trunk/
├── expected_witness/
├── src/
│   ├── coccinelle_util/
│   ├── coccinelle/
│   ├── CoccinelleUtils.py
│   ├── DifferencesUtils.py
│   ├── TransformationsUtils.py
│   └── WitnessUtils.py
├── main.py
├── transform_program.py
├── witness_backtransformation.py
├── constants.py
└── ...+
```

The scripts for program transformation are stored in the folder `coccinelle`. The scripts for structure creation are stored in the folder `coccinelle_util`. The folder

`c_trunk` contains files, used for testing and evaluation purposes. `output` and `temporary` are utility folder, needed for algorithm internal processes. The folders labelled `expected_c_trunk` and `expected_witness` contain files that are pertinent to the evaluation process.
The witness backtransformation flow is comprised of the following steps:

1. Removal of side-effects in a C program. In order to achieve this, the `transform_and_safe_modification()` function from `util` module is used. The location and name of the C program must be specified as arguments of a function.
2. After the execution of the transformation method, the transformed C file is located in the `output/` folder next to `transformation_protocol.json`.
3. It is necessary to verify the transformed C program with CPAchecker in order to create a witness version 2.0. The witness should be placed in the `output/` folder under the name `witness-after.yaml` .
4. The `witness_back_transformation()` method should be invoked with the name of the witness as argument. The backtransformed witness will be stored in the `output/` folder under the `witness-back-transformed.yaml` name.

Additional information about the project can be found in the project's README file.

5 Evaluation

In this section, we evaluate the implemented algorithm. We also evaluate the efficiency with which the algorithm accomplishes its task and discuss any potential weaknesses. To accomplishe the evaluation we will answer the following research questions:

RQ1. Algorithm evaluation.

- Is it possible to verify and validate the program after side-effects removal?
- Is it possible to validate a witness after backtransformation with the original program?
- Are validation results differ before and after witness transformation?

RQ2. Algorithm correctness.

- Does the information about invariants in the witness after transformation correspond to the information about invariants in the witness before transformation?

5.1 Quantitative Analysis

To answer the research questions, we conducted a series of experiments to evaluate the correctness and performance of the tool. The experiment sets were executed on a MacOS(Ventura 13.6.4) with a 2,3 GHz Dual-Core Intel Core i5 and 8GB of memory. To generate witnesses in YAML format during development process I will use CPAchecker - a tool for the configurable software verification [6]. In addition, we used the following tools:

- Java: Version 20.0.2 TM (GraalVM)
- CPAchecker Version 2.3.1
- Python Version 3.9.1
- Coccinelle 1.2 compiled with OCaml version 5.1.0

We run a bench of experiments with a set of programs, which contain side-effects from the research scope: increment and decrement as part of the while and for loop expression, assigning of the global variable and OR operator inside the if-expression. The number of required side-effect removal transformations varies considerably, from one to four. The programs are structured in a way that places loops, expressions, and statements containing side-effects in different positions according to each other. In this way, we try to cover more different edge cases. All programs for the experimental set can be found in the `c_trunk` folder of the project folder.

Examples of the results of experiments execution are demonstrated in the Fig. 5.1.

Column 1	*Column 2*	*Column 3*	*Column 4*	*Column 5*	*Column 6*
Amount of invariants	**Amount of transformations**	**Transformations type**	**Validation before transformation**	**Validation after transformation**	**Validation after back transformation**
1	1	NV	TRUE	TRUE	TRUE
3	3	NV, S	TRUE	TRUE	TRUE
2	1	NV	TRUE	TRUE	TRUE
1	1	NV	TRUE	TRUE	TRUE
1	1	NV	TRUE	TRUE	TRUE
2	3	NS, S	TRUE	TRUE	TRUE
2	2	NV, NS	TRUE	TRUE	TRUE
3	4	NV, NS	TRUE	TRUE	TRUE
1	0	-	TRUE	TRUE	TRUE
2	2	NS, S	TRUE	TRUE	TRUE
2	1	NS	TRUE	TRUE	TRUE
2	1	NS	TRUE	TRUE	TRUE
1	1	S	UNKNOWN	UNKNOWN	UNKNOWN
2	2	S	TRUE	TRUE	TRUE
2	1	S	TRUE	TRUE	TRUE
5	2	S	TRUE	TRUE	TRUE
1	1	S	TRUE	TRUE	TRUE
3	3	S	TRUE	TRUE	TRUE
2	2	S	TRUE	TRUE	TRUE
2	2	S	TRUE	TRUE	TRUE
3	1	S	TRUE	TRUE	TRUE
2	1	S	TRUE	TRUE	TRUE
2	2	S	UNKNOWN	UNKNOWN	UNKNOWN
4	3	S	TRUE	TRUE	TRUE
2	2	S	TRUE	TRUE	TRUE
3	4	S, NS	TRUE	TRUE	TRUE
3	4	S, NV	TRUE	TRUE	TRUE
2	3	S, NS	TRUE	TRUE	TRUE
3	1	S, NS	TRUE	TRUE	TRUE
4	3	S, NS	TRUE	TRUE	TRUE
6	4	S, NS	TRUE	TRUE	TRUE
2	1	S	TRUE	TRUE	TRUE

Figure 5.1: Result set of evaluation experiments

Column 1 of the Fig. 5.1 represent the amount of Invariants and *Column 2* - amount of transformations in each experiment case. *Column 3* shows the type of transformation that were done during experiment. Here, `NS` - new structure transforma-

tion, `NV` - new variable transformation and `S` - syntactic transformation. We use `-valueAnalysis` with `-heap 10000M` and `-timelimit '900 s'`. The specification file is `config/properties/unreach-label.prp`.
Each experiment consists of following steps:

1. Program `original.c` verification with CPAchecker, which ends with creation of `witness-original.yaml`.
2. Validation of `witness-original.yaml` with `original.c` to proof a validation verdict. The result of this step is shown in *Column 4* of Fig. 5.1.
3. Transformation of the C program `program.c` to remove side-effects, which ends with creating of `transformed.c`.
4. Program `transformed.c` verification with CPAchecker, which ends with creation of `witness-transformed.yaml`.
5. Validation of `witness-transformed.yaml` with `transformed.c` to proof a validation verdict. The result of this step is shown in *Column 5* of Fig. 5.1.
6. Transformation of `witness-transformed.yaml` into `witness-back-transformed.yaml` using implemented algorithm.
7. Validation of `witness-back-transformed.yaml` with `program.c`. The result of this step is shown in the *Column 6* of the Fig. 5.1.

A comparison of the results of witness validation for programs at different stages of the experiment reveals that the results of validation are consistent. This evidence indicates that transformed witnesses can provide valid information about the original program and can be used as valid witnesses for its validation. Basing on this evidence, it can be concluded that the responses to RQ1 are positive.
However, the evaluation results do not show the explicit correctness of the witness backtransformation algorithm. To increase the strength of evidence we provide an additional checking, that transformation process yields the expected information about each invariant of the original program.

5.2 Qualitative Analysis

To validate that the witness back algorithm correctly modifies invariants, we provide a manual approvment of witnesses, generated during evaluation experiments execution. After each experiment the `witness-back-transformed.yaml` witnesses were stored as separate files for further comparison. Subsequently, a manual comparison of the invariants information to the `original.c` program was conducted. During the comparison process, the following information was verified:

- invariant location line and column in witness conform the start of the loop in the program;
- invariant location function in witness conform the function name in the program;
- invariant value in witness contain information about relevant variables in the program.

The order of the placement of invariants in the witness can be arbitrary.
Let us examine an example of the `original.c` program on Listing 12 and an example of the `witness-back-transformed.yaml` on Listing 13, which are part

of the experiment set from Sect. 5.1. During the manual comparison, the following information was verified:

- invariant location line and column in `witness-back-transformed.yaml` correspond to the `for` loop starting line on `original.c`;
- invariant location function in `witness-back-transformed.yaml` conforms to the function name `factorial` on `original.c`;
- invariant value in `witness-back-transformed.yaml` contains C expression over variables in the scope where the invariant is placed [1].

```
20          ...
21          unsigned int factorial(unsigned int n) {
22              int result = 1, i;
23
24              for (i = 2; i <= n; i++) {
25                  result *= i;
26              }
27
28          return result;
29          }
30          ...
```

Listing 12: Example of original.c program

```
- invariant:
type: "loop_invariant"
location:
    file_name: "../output/original.c"
    line: 24
    column: 5
    function: "factorial"
value: "(i == (3) && n == (5U) && result == (2))
    || (i == (5) && n == (5U) && result == (24))
    || (i == (6) && n == (5U) && result == (120))
    || (i == (4) && n == (5U) && result == (6))
    || (i == (2) && n == (5U) && result == (1))"
format: "c_expression"
```

Listing 13: Example of witness-back-transformed.yaml

It has been demonstrated that all transformed invariants are in alignment with the data present in the original program. This provides further evidence that the data produced by the algorithm is valid in the transformed witness, and that the witness backtransformation algorithm functions correctly. Basing on this evidence, it can be concluded that the RQ2 is answered in the positive.
All components of the experiments are available in the `evaluation` folder of the project folder.

5.3 Threats to Validity

Threats to Internal Validity. One potential threat of validity can be an inappropriate choice of techniques or incorrect experimental design. The experimental set of programs was self-defined, and it is therefore possible that the programs may have been formed incorrectly or may have contained some invalid cases. Another potential threat to the validity of the evaluation is the possibility that errors may have been made during the process of conducting, interpreting, or planning the experiments. These threats can affect the validity of the findings and may result in incomplete conclusions, making the obtained experimental results questionable.

Threats to External Validity. The experimental set is based on the research scope and therefore cannot be considered a general approach for all transformation use cases. The experiments conducted cannot be generalized. The selected experimental set was created within the framework of this study. Conducting a similar experiment with a number of programs from the real world may potentially show different results or be biased towards the purpose of this thesis. This may be due to a limited sample of side-effect examples, the choice of analysis for verification, or edge cases that were not considered during the development process due to the time constraints. Additionally, the physical limitations of the development environment influenced the choice of experimental methods and general approaches to the creation of the experimentation set.

6 Future work

This section is devoted to the evaluation of the potential development of the problem-solving approach and the algorithm presented within this thesis.

6.1 Improvement of existing state

The developed algorithm can handle the required tasks. However, there are a number of points that could be improved. For example, when adding new structures, we read while and for loops that can be added as part of changes. The list of such loops could be extended. It is also possible to optimise the process of replacing a variable in the C program with a new one, as this process is now done with the help of two scripts - Coccinelle and Python. It would also be useful to extend the examples of possible occurance for already existing transformations. During the development of the witness back algorithm, different criteria had to be taken into account, such as where in the program the changes occur, how they behave within different semantic structures, and how they affect other elements of the program. Extension of the list of these criteria can make the algorithm more flexible and efficient.

6.2 Adding new components

An effective way to improve the functionality of the existing tool is to add more components to it. The existing algorithm was implemented on the selected rescticted scope of side-effects. This can be changing of global variable value, increment and decrement operators as part of *while* and *for* loop expressions, *OR* operator inside the *if* expressions. By increasing the number of different side-effects, the algorithm can be applied to more different C programs, increasing its versatility and efficiency.

7 Conclusion

In this thesis, we address the problem of backtransformation of Software Verification Witnesses Version 2.0, which can occur when transforming the program to improve the software verification process. We achieve this goal by using the case study of removing side-effects in C programs to produce a subject for witness modifications. In order to eliminate various side-effects during program modification, we developed different approaches to export transformations. During the development process, we created a format for exporting different types of modifications: syntactic and structurally relevant types of transformation. This allowed us to create a witness backtransformation for each type of modification.
Our contribution plays an important role in the development of Software Verification Witnesses Version 2.0, as it demonstrates that witness backtransformation is possible and it is not an interference for transformation of the program for improving of the software verification process. The algorithm can be used in its current form. However, the approaches developed for detecting and handling differences provide an opportunity to further extend the capabilities of the algorithm.

Bibliography

[1] P. Ayaziová, D. Beyer, M. Lingsch-Rosenfeld, M. Spiessl, and J. Strejcek. Software verification witnesses 2.0.

[2] J. A. Bergstra, A. Ponse, and D. J. C. Staudt. Non-commutative propositional logic with short-circuit evaluation. *Journal of Applied Non-Classical Logics*, 31:234 – 278, 2021.

[3] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Correctness witnesses: exchanging verification results between verifiers. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 326–337, New York, NY, USA, 2016. Association for Computing Machinery.

[4] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig. Verification witnesses. *ACM Trans. Softw. Eng. Methodol.*, 31(4):57:1–57:69, 2022.

[5] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 721–733, New York, NY, USA, 2015. Association for Computing Machinery.

[6] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[7] D. Beyer, M. L. Rosenfeld, and M. Spiessl. A unifying approach for control-flow-based loop abstraction. In B. Schlingloff and M. Chai, editors, *Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings*, volume 13550 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2022.

[8] M. Harman, L. Hu, R. M. Hierons, M. Munro, X. Zhang, J. J. Dolado, M. C. Otero, and J. Wegener. A post-placement side-effect removal algorithm. In *18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3-6 October 2002, Montreal, Quebec, Canada*, pages 2–11. IEEE Computer Society, 2002.

[9] N. Jones and R. Hansen. The semantics of "semantic patches" in coccinelle: Program transformation for the working programmer. volume 4807, pages 303–318, 11 2007.

[10] B. W. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[11] J. Lawall and G. Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In H. S. Gunawi and B. C. Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 601–614. USENIX Association, 2018.

[12] J. Lawall and G. Muller. Automating program transformation with coccinelle. In J. V. Deshmukh, K. Havelund, and I. Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 2022.

[13] J. L. Lawall and G. Muller. Automating program transformation with coccinelle. In *NASA Formal Methods*, 2022.

[14] D. MacKenzie, P. R. Eggert, and R. M. Stallman. Gnu diffutils reference manual. 2015.

[15] B. R. Maxim and M. Kessentini. Chapter 2 - an introduction to modern software quality assurance. In I. Mistrik, R. Soley, N. Ali, J. Grundy, and B. Tekinerdogan, editors, *Software Quality Assurance*, pages 19–46. Morgan Kaufmann, Boston, 2016.

[16] G. C. Necula, S. Mcpeak, M. Harren, W. Weimer, and B. Liblit. Cil: Infrastructure for c program analysis and transformation. 2009.

[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, 2002.

[18] T. Rothwell and J. Youngman. The gnu c reference manual. *Free Software Foundation, Inc*, page 86, 2007.

[19] T. Süß, L. Nagel, M.-A. Vef, A. Brinkmann, D. Feld, and T. Soddemann. Pure functions in c: A small keyword for automatic parallelization. *International Journal of Parallel Programming*, 49:1 – 24, 2020.

[20] S. Tamarit, G. Vigueras, M. Carro, and J. Mariño. A haskell implementation of a rule-based program transformation for C programs. In E. Pontelli and T. C. Son, editors, *Practical Aspects of Declarative Languages - 17th International Symposium, PADL 2015, Portland, OR, USA, June 18-19, 2015. Proceedings*, volume 9131 of *Lecture Notes in Computer Science*, pages 105–114. Springer, 2015.

[21] G. Vigueras, M. Carro, S. Tamarit, and J. Mariño. Towards automatic learning of heuristics for mechanical transformations of procedural code. *CoRR*, abs/1603.03022, 2016.