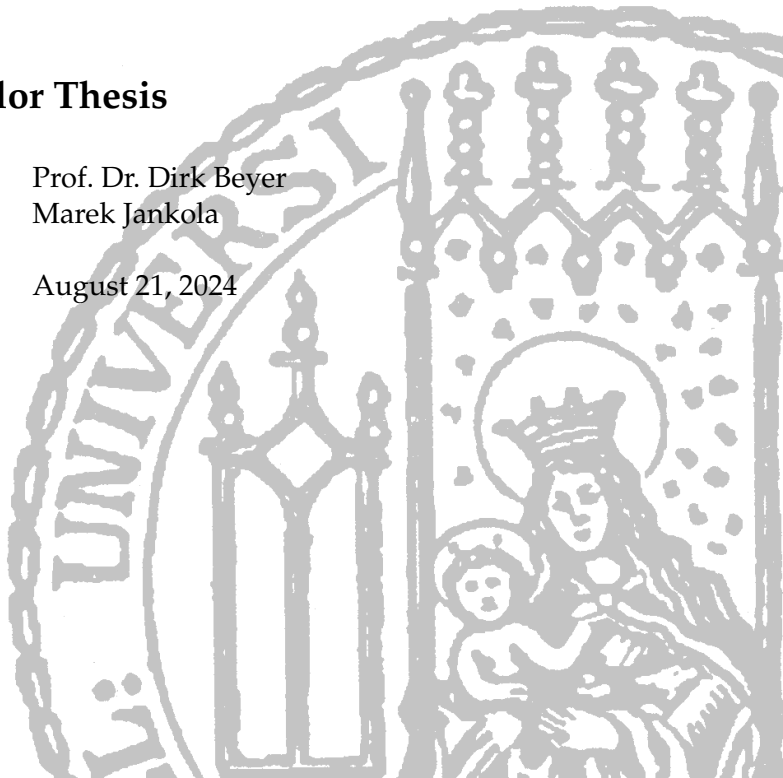
INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

Transformation of Reachability YAML Witnesses to No-Overflow YAML Witnesses

Tim Kriegelsteiner



Bachelor Thesis

Supervisor Prof. Dr. Dirk Beyer
Mentor Marek Jankola

Submission Date August 21, 2024

b

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.

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.

München, August 21, 2024 Tim Kriegelsteiner

Acknowledgements

Code fragments written in the repository of the witness transformation algorithm, that are generated by or with the help of `ChatGPT`[1] are marked in the respective code file or the `README` file of the repository.[2] Furthermore, `ChatGPT` was used to generate `LateX` commands and tables, that were otherwise tiresome to produce manually. It was also utilized to enhance the style of single sentences.

Thanks are due to the Software Systems Lab of the LMU Munich for providing a thesis subject. Furthermore, I am indebted to Marek Jankola for his mentoring. Greetings and gratitude are also extended to the elements of my social environment, particularly to my neighbor's dogs, whose constant barking could only be interpreted as their way of cheering me on during the writing of this thesis. Lastly, I am much obliged to the deities of our modern pantheon: Superman, Snoopy, Spongebob and their like. May they guide you, *dear reader*, through this thesis.

[1] https://chatgpt.com.

[2] https://gitlab.com/sosy-lab/software/specification-transformation.

Abstract

One important aspect of improving formal verification software is to improve its decisiveness, that means, its yielding of more results that are either true or false. With the possibility of using the reachability analysis to check no-overflow properties, this and other performance can be improved. Though this can be achieved by transforming the program to be verified by instrumenting it with assertions, the resulting witness contains verification information on this transformed program and not the original one. As the intent of verification is to have information on the latter, there is a need for adaption of the resulting witnesses to maintain its information and validation value regarding the original program. This was achieved by writing an algorithm that converts the witness to a version that is the same type as a witness produced by the usual no-overflow property. The validations, which are using the transformed witnesses, demonstrate the effectiveness of the property conversion and the potential of a property transformation approach.

Contents

Contents iii

List of Figures v

List of Tables vi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Overall Approach . . . 2
- 1.3 Example . . . 3
- 1.4 Contributions . . . 5
- 1.5 Related Work . . . 5

2 Background 7
- 2.1 CPAchecker . . . 7
- 2.2 Formal verification and Model checking . . . 7
- 2.3 Properties . . . 8
- 2.4 Program Transformation . . . 10
- 2.5 Witnesses . . . 10

3 Witness Transformation Algorithm 17
- 3.1 Preparation . . . 17
- 3.2 Line Matching . . . 19
  - Summary . . . 20
  - Details . . . 21
- 3.3 Realignment . . . 23
- 3.4 Replacement . . . 24
  - Replacement Algorithm For Correctness Witnesses . . . 25
  - Replacement Algorithm For Violation Witnesses . . . 26

4 Application 28
- 4.1 Research Questions . . . 28
- 4.2 Experimental Setup . . . 28
  - Program Transformation . . . 30
  - Verification . . . 31
  - Witness Transformation . . . 36

Validation 36
4.3 Results 38
Validation of reachability correctness witnesses of transformed programs 38
Validation of transformed no-overflow correctness witnesses 38
4.4 Evaluation 38
Research Question 1 39
Research Question 2 39

5 Conclusion **40**
5.1 Witness Transformation Algorithm 40
5.2 Verification And Validation Results 40

Bibliography **42**

List of Figures

1.1 Overall workflow 2
1.2 Transformed program 3
1.3 Witness 4
1.4 Original program 5
1.5 Witness before and after transformation 6

2.1 Simple workflow 7
2.2 Property example: no-overflow.prp 8
2.3 Specification example: overflow.spc 9
2.4 Property example: unreach-call.prp 9
2.5 Reachability assertion 9
2.6 Old and new assertion format 11
2.7 Full witness example 12
2.8 Witness structure snippet 13
2.9 Program and corresponding correctness witness 15
2.10 Program and corresponding violation witness 16

3.1 Connections between line matching data 18
3.2 Snippets from Ackerman-1.c and its transformed version 18
3.3 Line data object 19
3.4 Index data of the original program 19
3.5 Transformation types example 1 20
3.6 Transformation types example 2 21
3.7 Line match data 22
3.8 Line match data object 22
3.9 Line matching pseudo code 23
3.10 Realignment 24
3.11 Witness template snippet before and after transformation 25
3.12 Invalid witness column 26
3.13 Multi-operation handling in comparison 27
3.14 Assertion and operation listing 27

4.1 Steps of the experiment 29
4.2 Used benchmark files 29
4.3 Program transformation step 30
4.4 Task definition file example 31
4.5 Transform program command 31

4.6 Verification step . . . 31
4.7 XML file for benchmark verification . . . 32
4.8 Cloud benchmark verification command . . . 33
4.9 CPU time comparison of new and old program transformation . . . 34
4.10 Memory comparison of new and old program transformation . . . 34
4.11 CPU time comparison filtered by the expected result: "true" . . . 35
4.12 CPU time comparison filtered by the expected result: "false" . . . 35
4.13 Witness transformation step . . . 36
4.14 Witness transformation command . . . 36
4.15 Validiation step . . . 37
4.16 Single file validation command . . . 37
4.17 Comparison step . . . 39

List of Tables

3.1 Final index data . . . 24

4.1 Setup for verification . . . 32
4.2 Performance of old and new program transformations . . . 33
4.3 Verification results of the (new) program transformations . . . 35
4.4 Results of the transformed correctness witness verification by relevance 35
4.5 Results of the reachability correctness witness verification by relevance . 38
4.6 Results of the reachability correctness witness validation . . . 38
4.7 Results of the reachability correctness witness validation by relevance . 38
4.8 Results of the transformed correctness witness validation . . . 38
4.9 Results of the transformed correctness witness validation by relevance . 38

1 Introduction

1.1 Motivation

Software verification is an integral method of error detection and avoidance in computer software engineering. Software verification, here synonymously used with "formal verification", is the practice of proving the correctness of a computer program with the use of mathematical logic [10]. Automated formal verification, which means verification done by a verification program (also called “verifier”), is a feasible option to prove the fail-safety of code, especially as the codebase grows larger. Improving and developing such verifiers is a research field to which this thesis aims to contribute.

A verification can mainly result in one of the three evaluations: true, false or unknown [2] [6]. If the result is the latter, a decision was not possible due to the complexity of the program or the condition, their inherent undecidability, or the insufficient resources of the algorithm to compute the verification task. Verifying consists of checking a program for specified conditions called "properties". A no-overflow property for example does not apply to a program, if an overflow error is detected. A verifier uses different tools and algorithms depending on which property is analysed. Nevertheless, some properties are similar in kind and therefore convertible. This also means that if one analyzer performs better than another, a conversion can lead to better performances checking a property. Zheng 2024 showed the possibility of a reachability analysis of overflow errors by transforming the programs before they are verified [13]. It was found that this transformation approach has the potential to compete with the standard approach [13]. Therefore, it should be further explored and developed.

Besides generating a verification result, a verifier also produces a witness. A witness is a protocol holding information on the verification, which can be used to validate the result manually or automatically [7]. While an algorithm was contributed by [13] to instrument programs for a reachability analysis of a no-overflow property, the witness produced by verifying the instrumented program is referring to the very program. This is of course the intentional behavior of the verifier, but for information on the original program, this witness will not be sufficient. Therefore, the thesis examines the possibility of transforming the witness produced by the verification of an altered program to a version equal to a no-overflow analysis witness.

1.2 Overall Approach

For the task of witness transformation, `CPAchecker`[1] is used as verification and validation tool [8]. Furthermore, the programs verified are written in `C`. From generating a transformed program to obtaining the results of the validation of the new witness, the process can be summarized as follows:

1. The program is transformed.
2. The transformed program is verified.
3. The resulting witness is transformed into an altered witness fitting the original program.
4. The transformed witness can be used for validation and information.

To get the no-overflow result and information by using reach-safety analysis, the transformed version of the program that should be verified is needed. If it is not there, it needs to be generated. This program has to be verified resulting in a witness. The witness then needs to be transformed with the original program, the transformed program and the witness used as input. Providing the new witness and the original program to validate the proof and getting the results, completes the process (see Figure 1.1).

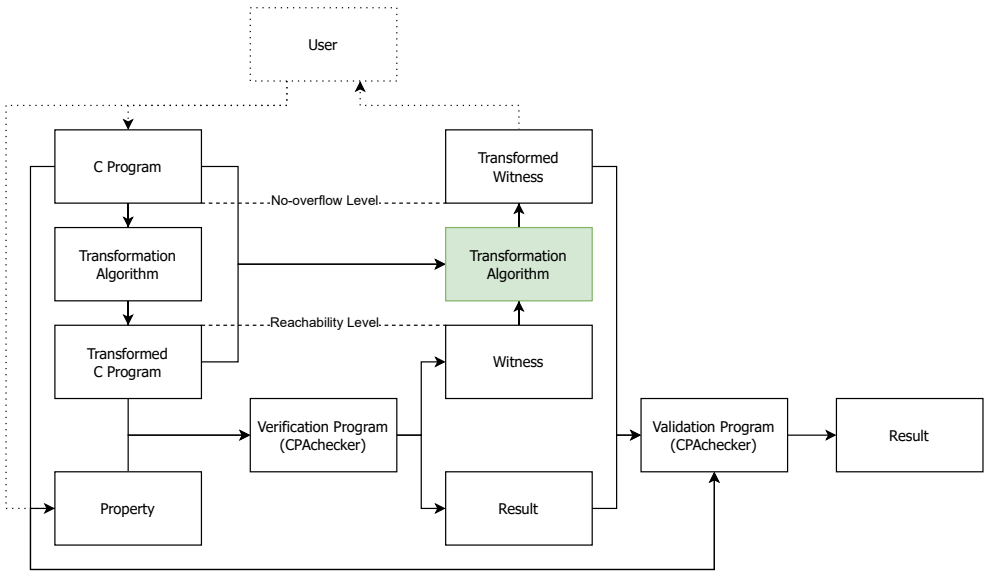


Figure 1.1: Overall workflow

[1] https://gitlab.com/sosy-lab/software/cpachecker.

1.3 Example

The program transformation alters the code with additions (`line 1-9, 23, 28-34`) and deletions (`line 23/24`) (see Figure 1.2).[2]

```
1  extern void __assert_fail(const char *, const char *, unsigned int,
       const char *) __attribute__((__nothrow__, __leaf__)) __attribute__
       ((__noreturn__));
2  extern void abort(void);
3  void reach_error() { __assert_fail("0", "bin-suffix-5.c", 3,
       "reach_error"); }
4  void __VERIFIER_assert(int cond) {
5    if (!(cond)) {
6      ERROR: {reach_error();abort();}
7    }
8    return;
9  }
10 /*
11  * Date: 2012-08-10
12  * Author: leike@informatik.uni-freiburg.de
13  *
14  * This program has the following 2-nested ranking function:
15  * f_0(x, y) = y + 1
16  * f_1(x, y) = x
17  */
18
19 typedef enum { false, true } bool;
20
21 extern int __VERIFIER_nondet_int(void);
22
23 int main() {
   {
24   int x, y;
25   x = __VERIFIER_nondet_int();
26   y = __VERIFIER_nondet_int();
27   while (x >= 0) {
28     if (!((y >= 0 && x <= (2147483647 - y)) ||
       (y <= 0 && x >= -2147483648 - y))) {
29       ERROR1: {reach_error();abort();}
30     }
31     x = x + y;
32     if (!((1 >= 0 && y >= -2147483648 + 1) ||
       (1 <= 0 && y <= 2147483647 + 1))) {
33       ERROR2: {reach_error();abort();}
34     }
35     y = y - 1;
36   }
37   return 0;
38 }
39
```

Figure 1.2: Transformed program

[2] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/termination-crafted/2Nested-2.c.

```yaml
- entry_type: "violation_sequence"
  metadata:
    format_version: "2.0"
    uuid: "da12266b-0c9f-4c5f-a264-1bdf812b554b"
    creation_time: "2024-07-09T21:04:21+02:00"
    producer:
      name: "CPAchecker"
      version: "2.3.1-svn-77
          fd7b5113039273790bb136467e329a9c19d72d"
      configuration: "svcomp24"
    task:
      input_files:
      - "./transformed_2Nested-2.c"
      input_file_hashes:
        "./transformed_2Nested-2.c": "544
            efbc0324ad7239b54d3089e6cbd7e59072532b93e64f2f2d08fb9407cc9db
            "
      specification: "G ! call(reach_error())"
      data_model: "LP64"
      language: "C"
  content:
  - segment:
    - waypoint:
        type: "assumption"
        action: "follow"
        constraint:
          value: "(x == 2147483647)"
          format: "c_expression"
        location:
          file_name: "./transformed_2Nested-2.c"
          file_hash: "file_hash"
          line: 27
          column: 10
          function: "main"
  - segment:
    - waypoint:
        type: "target"
        action: "follow"
        location:
          file_name: "./transformed_2Nested-2.c"
          file_hash: "file_hash"
          line: 29
          column: 16
          function: "main"
```

Figure 1.3: Witness

The transformed program is verified resulting in a witness, here with an error detection, referring to the transformed program (see Figure 1.3). The locations one (`line 27, column 10`) and two (`line 29 and column 16`) are pointing to the code positions where the error occurred.

The corresponding lines of the original and transformed program are matched to get the data, including the new locations (in the program Figure 1.2: `line 19, column 5 & line 20, column 15`, in the witness Figure 1.3: `line 28, 29 & line 37, 38`) to transform the witness (see Figure 1.4 and Figure 1.5).

```
1  /*
2   * Date: 2012-08-10
3   * Author: leike@informatik.uni-freiburg.de
4   *
5   * This program has the following 2-nested ranking function:
6   * f_0(x, y) = y + 1
7   * f_1(x, y) = x
8   */
9
10 typedef enum {false, true} bool;
11
12 extern int __VERIFIER_nondet_int(void);
13
14 int main()
15 {
16     int x, y;
17         x = __VERIFIER_nondet_int();
18         y = __VERIFIER_nondet_int();
19         while (x >= 0) {
20                 x = x + y;
21                 y = y - 1;
22         }
23         return 0;
24 }
```

Figure 1.4: Original program

1.4 Contributions

The thesis contibutes the follwing:

- In the context of no-overflow to reachability transformation, an algorithm to transform witnesses is provided.
- The success of the programmed witness transformation opens up the possibility of testing no-overflow errors by reach-safety analysis with having information concerning the original program.
- The successful validation of the transformed witnesses also shows that the program transformation is correct.

1.5 Related Work

Literature on the transformation of witnesses is scarce. The only publication found was the recently finished bachelor's thesis of Anna Ovezova which is a case study in transforming witnesses to fit the original program, if the program was altered and therefore the produced witness was referring to the altered version of the program [11]. While it is the same task as the one in this thesis, it is not focused on a specific

```
content:
- segment:
  - waypoint:
      type: "assumption"
      action: "follow"
      constraint:
        value: "(x ==
            2147483647)"
        format: "
            c_expression"
      location:
        file_name:
            "./transformed_
            2Nested-2.c"
        file_hash: "
            file_hash"
        line: 27
        column: 3
        function: "main"
- segment:
  - waypoint:
      type: "target"
      action: "follow"
      location:
        file_name:
            "./transformed_
            2Nested-2.c"
        file_hash: "
            file_hash"
        line: 29
        column: 16
        function: "main"
```

```
content:
- segment:
  - waypoint:
      type: "assumption"
      action: "follow"
      constraint:
        value: "(x ==
            2147483647)"
        format: "
            c_expression"
      location:
        file_name:
            "./2Nested-2.c"
        line: 19
        column: 5
        function: "main"
- segment:
  - waypoint:
      type: "target"
      action: "follow"
      location:
        file_name:
            "./2Nested-2.c"
        line: 20
        column: 15
        function: "main"
```

Figure 1.5: Witness before and after transformation

software, but follows a systematic approach. The main reason the paper was not considered was that it was published after the algorithm presented in this thesis had already been completed. Another reason is that the implementation of the approach presented there would have been more time-consuming than the solution given here, which was tailored to the specific problem being addressed.

2 Background

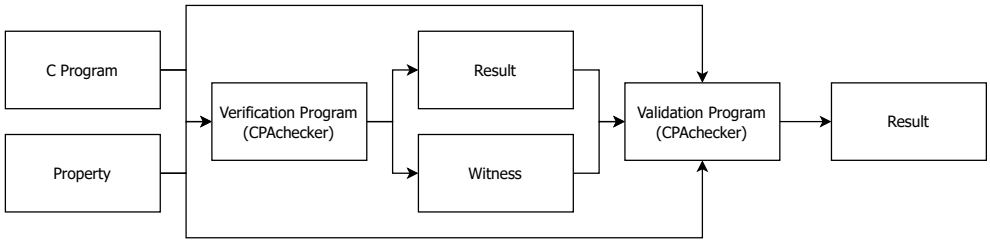


Figure 2.1: Simple workflow

2.1 CPAchecker

Basic insights about verification were formulated in Section 1.1. To deepen the understanding of it, the following sections are provided. To start with, a presentation of `CPAchecker`, the tool used for verification and validation experiments in this thesis, is given. It should serve as an example for the sections that come after.
`CPAchecker` is a software verification program developed by the SoSy-Lab at the LMU Munich.[1] It is based on configurable program analysis (CPA) and control-flow automata (CFA) data structure [8]. The latter is generated from parsing the verified program into an abstract syntax tree [8]. Though it can verify programs written in Java, its focus lies on C programs [3]. The workflow of `CPAchecker` without the transformation steps gives a better overview regarding the basic verification process and is therefore shown in Figure 2.1.

2.2 Formal verification and Model checking

Formal verification can be done by model checking, which is also used by verifiers. Modelling and checking can be described based on Emerson et al. 2009 [10] as follows:

1. Model:
   A model is an abstract representation of the system (or computer program) to be verified.

[1] `https://www.sosy-lab.org/.`

2. Specification:
   A specification is the representation of the intended behavior of the system. It is expressed through temporal logic for example Computation Tree Logic (CTL) or Linear Temporal Logic (LTL) (see also Piterman and Pnueli 2018 [12])
3. Checking:
   When checking a system, every state of it is explored to check whether the specification holds: Given a model M, a state s of M, and a formula (specification) f, the checking of one state can be written as

$$M, s \models f$$

In summary, when model checking, a verification program checks if a program does what it should do following its specification.

2.3 Properties

"Properties" is another word for "specifications". `CPAchecker` has implemented these under the term "properties" as well as "specifications".[2] Properties are formulated in a type of LTL, while specifications use a type of CTL, seen in Figure 2.2 and Figure 2.3 and taken from the `CPAchecker` repository.[3]

```
1  CHECK( init(main()), LTL(G ! overflow) )
```

Figure 2.2: Property example: no-overflow.prp

The two for this thesis important properties are no-overflow and reachability. They are defined as follows.
No-overflow:

> "It can never happen that the resulting type of an operation is a signed integer type but the resulting value is not in the range of values that are representable by that type." [4]

Reachability:

> This property holds, if a specified function is not called or rather a condition will never be reached that leads directly to this function [2] [4].

To understand, how no-overflow checking can be substituted by reachability checking, it is worth noting the following: Verifiers use different analyzers to check different properties: A reachability property is analyzed by a reachability analyzer, a no-overflow property is analyzed by a no-overflow analyzer and so on. While no-overflow analyzers check cues the program code already inherits, reachability analysis checks, if a specified reachability function was called (see Figure 2.4). That means, a reachability function and its calls are added to the code. In practice, reachability assertions are placed in the code that call the reachability function as seen in Figure 2.5. The `__VERIFIER_assert()` function (`line 4 & 39`) calls

[2] https://gitlab.com/sosy-lab/software/cpachecker/-/tree/trunk/config.
[3] Explanations of the languages can be found at Beyer 2015 [2] and https://gitlab.com/sosy-lab/software/cpachecker/-/blob/trunk/doc/SpecificationAutomata.md.

```
1  // This file is part of CPAchecker,
2  // a tool for configurable software verification:
3  // https://cpachecker.sosy-lab.org
4  //
5  // SPDX-FileCopyrightText: 2007-2020 Dirk Beyer
   <https://www.sosy-lab.org>
6  //
7  // SPDX-License-Identifier: Apache-2.0
8
9  // This automaton contains the specification of the
10 // category Overflows of the
11 // Competition on Software Verification.
12 CONTROL AUTOMATON Overflows
13
14 INITIAL STATE Init;
15
16 STATE USEFIRST Init :
17   CHECK("overflow") -> ERROR("no-overflow: integer overflow in
   $location");
18
19 END AUTOMATON
```

Figure 2.3: Specification example: overflow.spc

```
1  CHECK( init(main()), LTL(G ! call(reach_error())) )
```

Figure 2.4: Property example: unreach-call.prp

```
1  extern void __assert_fail(const char *, const char *, unsigned int,
       const char *) __attribute__ ((__nothrow__ , __leaf__))
       __attribute__ ((__noreturn__));
2  extern void abort(void);
3  void reach_error() { __assert_fail("0", "bin-suffix-5.c", 3, "
       reach_error"); }
4  void __VERIFIER_assert(int cond) {
5    if (!(cond)) {
6      ERROR: {reach_error();abort();}
7    }
8    return;
9  }

...

39     __VERIFIER_assert((1 >= 0 && x >= -2147483648 + 1)
           || (1 <= 0 && x <= 2147483647 + 1));
40     x--;
```

Figure 2.5: Reachability assertion

an error function (line 3 & 6), if the assertion condition (line 39) does not hold. As the condition leading to the function call is not specified by the reachabiltiy property, it is possible to let these conditions represent no-overflow conditions and thereby enable reachbility analysis to check overflow errors.

2.4 Program Transformation

The first nine lines shown in Figure 2.5 plus the assertions before every signed integer operation represent the core part of the transformation of C programs presented in the work by Xiyue Zheng [13]. Her thesis revolves around the task of transforming C programs to make them suitable for overflow checking by reachability analysis. This was done by extracting CFA data generated by `CPAchecker` and analyzing it [13]. Apart from that, some original code had to be commented out that would otherwise have affected the functionality or compilation of the transformed code [13]. The results regarding the performance did not lead to a preference of one analysis other the other: While reachability analysis performed better under some circumstances, no-overflow analysis did under others [13]. This is a positive result as it shows that the transformational approach can compete with the original one. Thus, improvement and further development is fruitful.
The witness transformation outlined here is based upon the witnesses that were produced along the transformed programs generated by the algorithm presented in Zheng 2014 [13]. This algorithm was not written with the task in mind to transform the witnesses. Therefore, it was necessary to alter the algorithm as shown in Figure 2.6.
The former assertion is a function `void __VERIFIER_assert(int cond)` (`line 3 & 22`) defined in the beginning and called later. This caused a problem with transforming violation witnesses, because the line reference for every error was `line 5`, which is a function call from within the assertion function: `ERROR: {reach_error();abort();}` (`line 53`). It is not possible from this line to deduce, at which state of the code the error actually happens. Therefore, instead of calling the assertion function, the function body (`line 4`) had to be put at every assertion check separately (`line 22`) along with creating new "`ERROR`" labels by numerating them (`line 23`).
The performance of this new transformation is discussed in Section 4.2.

2.5 Witnesses

The procedure of analyzing transformed programs has some side effects regarding the additional output of the verification. The witnesses, which contain information about the program and the verification process, are witnesses belonging to the transformed programs. Because of fixed sets of test files and the simple practical reason, that one wants to have information on the program, which will be used in ones project, and not some transformed version of it, the witnesses have to be transformed to fit the original programs. To understand, which parts have to be changed, the following selective description of witnesses is given (see Figure 2.8). For a full description, see Ayaziová et al. 2024 [1]. For a full example, see Figure 2.7.[4]

[4]Witness in Figure 2.7 is the verification result of this program: `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/termination-crafted-lit/AliasDarteFeautrierGonnord-SAS2010-easy1.c`.

Old assertion:

```
1  extern void abort(void);
2  void reach_error() { __assert_fail("0", "bin-suffix-5.c", 3, "
       reach_error"); }
3  void __VERIFIER_assert(int cond) {
4    if (!(cond)) {
5      ERROR: {reach_error();abort();}
6    }
7    return;
8  }
9  extern void __assert_fail(const char *, const char *, unsigned int,
       const char *) __attribute__ ((__nothrow__ , __leaf__))
       __attribute__ ((__noreturn__));

                                    ...

21 if (m == 0) {
22     __VERIFIER_assert((1 >= 0 && n <= (2147483647 - 1))
           || (1 <= 0 && n >= -2147483648 - 1));
23     return n + 1;
24 }
```

New assertion:

```
21 if (m == 0) {
22     if (!((1 >= 0 && n <= (2147483647 - 1))
           || (1 <= 0 && n >= -2147483648 - 1))){
23         ERROR1: {reach_error();abort();}
24     }
25     return n + 1;
26 }
```

Figure 2.6: Old and new assertion format

Witnesses are protocols of automated verification processes and are produced with data of the verification process. They follow a certain format, which was agreed upon in the formal verification community, and contain information on the verification, including locations of invariants or error traces in the code [1].

> "The validation of verification results, in particular, verification witnesses, becomes more and more important for various reasons: verification witnesses justify and help to understand and interpret a verification result, they serve as exchange object for intermediate results, and they allow to make use of imprecise verification techniques (e.g., via machine learning)." [3]

The usability of a witness is twofold: On one hand, it is designed to help understand the verification process on user side [6]. On the other hand, it can be used for validation of the verification process, that means as input for an automatized validator,

```yaml
- entry_type: "invariant_set"
  metadata:
    format_version: "2.0"
    uuid: "d4bf7e44-5a8e-481f-ab5b-d7acf27b9216"
    creation_time: "2024-07-09T15:11:46+02:00"
    producer:
      name: "CPAchecker"
      version: "2.3.1-svn-77
          fd7b5113039273790bb136467e329a9c19d72d"
      configuration: "svcomp24"
    task:
      input_files:
      - "./transformed_AliasDarteFeautrierGonnord-SAS2010-easy1
          .c"
      input_file_hashes:
        "./transformed_AliasDarteFeautrierGonnord-SAS2010-easy1
            .c": "
            f9b56e72ba5eb010016cf0e8b7b4aef4a2c4952c0cf4cc40345b
             6187bc21dea8"
      specification: "G ! overflow"
      data_model: "LP64"
      language: "C"
  content:
  - invariant:
      type: "loop_invariant"
      location:
        file_name: "./transformed_AliasDarteFeautrierGonnord-
            SAS2010-easy1.c"
        line: 13
        column: 1
        function: "main"
      value: "y == (100) && (x == (12) || x == (34) || [...])"
      format: "c_expression"
```

Figure 2.7: Full witness example

that tries to reconstruct the proving of the verification program with the information the witness contains [3]. Witnesses are categorized into violation witnesses and correctness witnesses [5]. This separation is rooted in the different requirements and efforts needed to prove that something is correct or incorrect. Subsequently, the proving tools are implemented differently regarding the result, which leads to different information needed in the witness [5].

The current version (2.0) of witnesses is in `YAML`[5] format and the only version dealt with in this thesis [1]. The structure of the witness in `YAML` format can be described as an array of objects.[6] Although this is the case, I also will refer to these objects or their names as "keys". The YAML witness has three main keys: `entry_type`,

[5]Also called `YML`, with the file endings `.yml` or `.yaml`.

[6]See https://sosy-lab.gitlab.io/benchmarking/sv-witnesses/yaml/violation-witnesses.html.

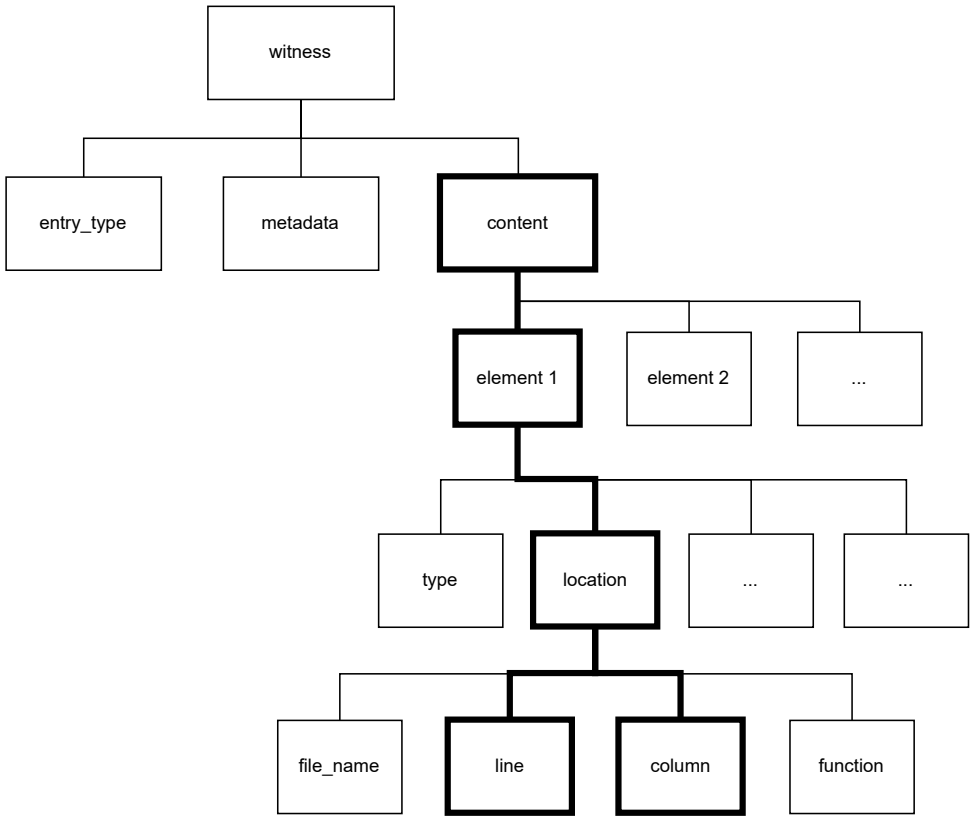


Figure 2.8: Witness structure snippet

`metadata` and `content`.[7] The value of `entry_type` determines the witness type. It is either `invariant_set` for correctness witnesses, or `violation_sequence` for violation witnesses. `Metadata` contain, as the name says, metadata which is not relevant regarding the transformation of witnesses. The `content` section is a list of elements called `invariants` (with their types `loop` and `location`), when dealing with correctness witnesses and `segments`, when dealing with violation witnesses. `Segments` consist of one or more `waypoints`. Every `waypoint` and every `invariant` hold information on a program state that is relevant for showing correctness or violation. This information is found at `location` and divided into four objects: `file_name`, `line`, `column` and `function`. While `function` does not need to be altered and `file_name` does not impact the automated validation (see also Ayaziová et al. 2024 [1]), but is substituted in the witness transformation presented in this thesis, `line` and `column` need to be changed. The values of both are dependent on the witness type. For each `type` of `waypoint` or `invariant`, there is a definition regarding the location:

[7] Every description of the witness format given is found in Ayaziová et al. 2024 [1].

Correctness witness (see Figure 2.9):[8,9]

`loop_invariant`:

"[The] location of a loop_invariant must point to the first character of a keyword at the beginning of a loop (i.e., for, while, or do)." [1]

`location_invariant`:

"The location of a location_invariant must point to the first character of a statement". [1]

Violation Witness (see Figure 2.10):[10]

`assumption`:

"The location has to point to the first character of a statement." [1]

`branching`:

"The location points to the first character of a branching keyword like if, while, switch, or to the character ? in the ternary operator (?:)." [1]

`function_enter`:

"The location points to the right parenthesis after the function arguments of a function call." [1]

`function_return`:

"The location points to the right parenthesis after the function arguments at the function call." [1]

`target`:

"[T]he location points at the first character of the statement or full expression whose evaluation is sequenced directly before the violation occurs, i.e., there is no other evaluation sequenced before the violation and after the sequence point associated with the location. This also implies that it can point to a function call only if it calls a function of the C standard library that violates the property or if the function call itself is the property violation." [1]

```c
int main(void) {
  int A[2048];
  int i;

  for (i = 0; i < 1024; i++) {
    A[i] = i;
  }

  __VERIFIER_assert(A[1023] != 1023);
}
```

```yaml
  content:
  - invariant:
      type: "loop_invariant"
      location:
        file_name: "./array_1-1.c"
        line: 16
        column: 3
        function: "main"
      value: "i == (455) || i == (558) || ..."
      format: "c_expression"
  - invariant:
      type: "location_invariant"
      location:
        file_name: "./array_1-1.c"
        line: 17
        column: 5
        function: "main"
      value: "i == (83) || i == (236) || ..."
      format: "c_expression"
```

Figure 2.9: Program and corresponding correctness witness

The main part of the algorithm presented in this thesis revolves around the replacement of the `lines` and `columns` pointing to the code positions of the transformed program with the numbers pointing to the positions of the original program (see Figure 2.8).[11] Currently, the violation witnesses generated by the transformed program verification only have the types `assumption` and `target`. Therefore, examples of the other `types` can not be provided. Nonetheless, a description of the handling of these cases by the witness transformation is presented in Section 3.4.

[8] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/loop-acceleration/array_1-1.c.

[9] The `value` section was shortened ("...") for readability.

[10] Excerpt already used in Figure 1.2

[11] To better subsume the witness types, "element" stands for `segment` and `waypoint` regarding violation witnesses, while for correctness witnesses it represents `invariant`.

```c
int main()
{
    int x, y;
        x = __VERIFIER_nondet_int();
        y = __VERIFIER_nondet_int();
        while (x >= 0) {
                x = x + y;
                y = y - 1;
        }
        return 0;
}
```

```yaml
  content:
  - segment:
    - waypoint:
        type: "assumption"
        action: "follow"
        constraint:
          value: "(x == 2147483647)"
          format: "c_expression"
        location:
          file_name: "./2Nested-2.c"
          line: 19
          column: 5
          function: "main"
  - segment:
    - waypoint:
        type: "target"
        action: "follow"
        location:
          file_name: "./2Nested-2.c"
          line: 20
          column: 15
          function: "main"
```

Figure 2.10: Program and corresponding violation witness

3 Witness Transformation Algorithm

The algorithm written to transform the witnesses is mainly focused on the replacement of the line and column references and can be split into four steps: Preparation, Line Matching, Realignment, Replacement (see Figure 3.1): (I) The Preparation step includes acquiring the program line data and transforming it, to enable its processing, including the removing of whitespaces, in order to make the programs comparable. (II) With the prepared data, it is then possible to match the lines of the original program with the corresponding lines of the transformed program. The result is a data object with mappings of columns and lines of the program and its transformed version. (III) As the first step removes whitespaces and therefore alters the lines, which leads to indices and other data referring to these alterations, it is necessary to realign them with the unaltered lines. (IV) In the final step, the `lines` and `columns` found in the witness are replaced with corresponding line and column numbers of the original C program, by using the acquired data. To illustrate the transformation process, snippets from the benchmark file `Ackermann-1.c`[1] are used (see Figure 3.2).[2]

3.1 Preparation

All required data from the C program and its transformation are stored in a data object, including different versions of the program lines, the program names, the object that holds the matching data, and the column indices of the programs (see Figure 3.3).
Because whitespaces are altered in the transformed programs, not only at the edges of a line, but also in the middle of it, it was necessary to delete them in order to make a program and its transformed version comparable. Another problem were the tabs which were replaced by whitespaces in the transformed program. Therefore, tabs were replaced by four whitespaces before whitespaces were stripped from the lines. After that, line indices data is stored for both program versions. Line indices data is a list of a list of every non-whitespace character with its `column` numbers from

[1] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/termination-crafted/Ackermann-1.c.

[2] The "ASSERTION" label in Figure 3.2 is a placeholder for an equation and used for readability. The coloring in this figure should only facilitate the comparison and help to understand the structure of changes in the example.

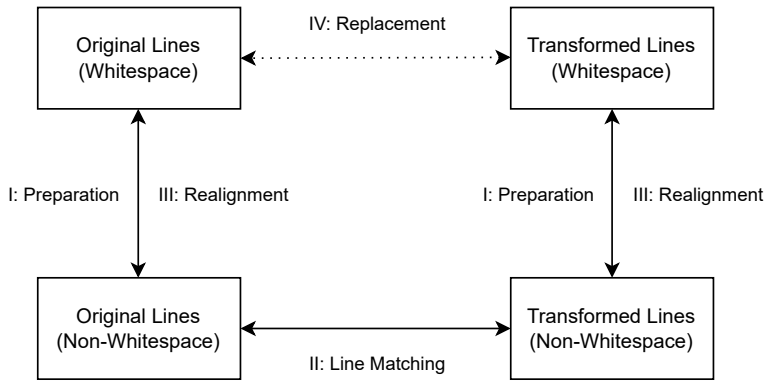


Figure 3.1: Connections between line matching data

Original program:

```
8  int Ack(int m, int n)
9  {
10   if (m == 0) return n+1;
11   else if (n == 0) return Ack(m-1, 1);
12   else return Ack(m-1, Ack(m, n-1));
13 }
```

Transformed program:

```
20 int Ack(int m, int n) {
21   if (m == 0) {
22     if (!(ASSERTION1)) {
23       ERROR1
24     }
25     return n + 1;
26   } else if (n == 0) {
27       if (!(ASSERTION2)) {
28         ERROR2
29       }
30       return Ack(m - 1, 1);
31   } else {
32       if (!(ASSERTION3)) {
33         ERROR3
34       }
35       if (!(ASSERTION4)) {
36         ERROR4
37       }
38       return Ack(m - 1, Ack(m, n - 1));
39   }
40 }
```

Figure 3.2: Snippets from Ackerman-1.c and its transformed version

the whitespace version and the non-whitespace version (see Figure 3.4). This data is later used for realignment.

```
class LineData:
    original_program_name: str
    transformed_program_name: str
    original_program_lines: list[str]
    transformed_program_lines: list[str]
    original_program_lines_wo_tabs: list[str]
    transformed_program_lines_wo_tabs: list[str]
    original_program_lines_wo_whitespaces: list[str]
    transformed_program_lines_wo_whitespaces: list[str]
    original_index_data: list[list[[IndexData]]
    transformed_index_data: list[list[[IndexData]]
    line_matcher: LineMatcher
```

Figure 3.3: Line data object

Original program with whitespaces:

```
int Ack(int m, int n)
{
  if (m == 0) return n+1;
  else if (n == 0) return Ack(m-1, 1);
  else return Ack(m-1, Ack(m, n-1));
}
```

Original program without whitespaces:

```
intAck(intm,intn)
{
if(m==0)returnn+1;
elseif(n==0)returnAck(m-1,1);
elsereturnAck(m-1,Ack(m,n-1));
}
```

List of index data objects:

character	i	f	(	m	=	=	0	)	r	e	t	u	r	n	n	+	1	;
with whitespaces	3	4	6	7	9	10	12	13	15	16	17	18	19	20	22	23	24	25
without whitespaces	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Figure 3.4: Index data of the original program

3.2 Line Matching

The key to writing the line matching algorithm was to understand in which ways the transformed lines can differ from its original counterparts.
The following alterations could occur (see Figure 3.5 and Figure 3.6):[3]

1. Curly brackets were added.
2. Breaks were removed.
3. Breaks were added.
4. Whitespaces were added.
5. Combinations of the four above.
6. Code was commented out.

[3]Second example from: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/memsafety-ext3/getNumbers4-1.c.

7. Assertion conditions were added above lines containing operations.
8. Additions to `while`, `for` and `if` statements.
9. Indentation was changed from a tab or four whitespaces to two whitespaces.
10. The first nine lines were new code lines added (does always occur). (see Figure 2.5)

Original program:

```
8  int Ack(int m, int n)
9  {
10   if (m == 0) return n+1;
11   else if (n == 0) return Ack(m-1, 1);
12   else return Ack(m-1, Ack(m, n-1));
13 }
```

Transformed program:

```
20 int Ack(int m, int n) {
21  if (m == 0) {
22     if (!(ASSERTION1)) {
23       ERROR1
24     }
25     return n + 1;
26   } else if (n == 0) {
27       if (!(ASSERTION2)) {
28         ERROR2
29       }
30       return Ack(m - 1, 1);
31   } else {
32       if (!(ASSERTION3)) {
33         ERROR3
34       }
35       if (!(ASSERTION4)) {
36         ERROR4
37       }
38       return Ack(m - 1, Ack(m, n - 1));
39   }
40 }
```

Figure 3.5: Transformation types example 1

The main obstacle was to handle the combination of changes in a clean and comprehensible way. The result was the following.

Summary

The line matching algorithm is inside a line matcher object, thus uses its global variables, and works as follows (see Figure 3.9): There is a nested while loop: The outside loop goes through the lines of the original program (o-lines) (`line 13`) and the inner loop iterates through lines of the transformed program (t-lines) (`line 17`), both are freed from whitespaces. Both loops start with checking if the line, which is the line at the current line pointer of the respective lines is an empty line. If it is, the line index is increased, and the code jumps to the next iteration (`line 14-16`, `22-24`). If not, the current iteration proceeds. Regarding the outer loop that is all

Original program:

```
43 int main (void) {
44
45     int *numbers = getNumbers4();
46     for (int i = 0; i < 10; i++ ) {
47     printf( "%d\n", *(numbers + i));
48
```

Transformed program:

```
88 int main(void) {
89
90   int *numbers = getNumbers4();
91   int i = 0;
92   if (!(ASSERTION9)) {
93   ERROR9: {reach_error();abort();}
94   }
95   for (int i = 0; i < 10; i++) {
96     //         printf( "%d\n", *(numbers + i));
```

...

Figure 3.6: Transformation types example 2

that happens besides the inner loop. If the line is not empty, the nested loop then tries to find a match. If there is a match, data on the match is stored. If the current o-line is completely matched after the latest match, all relevant data regarding this o-line is firstly stored in a temporary data object. Line and column pointers are updated depending on the case. A list of line match data objects is returned (see Figure 3.7 and Figure 3.8).

Details

The possible states of a transformed line, which are represented as conditions in the code are listed here (see Figure 3.9):

1. Transformed line == original line
2. Transformed line in original line
3. Transformed line without added bracket in the beginning in original line
4. Transformed line without added bracket in the end in original line
5. Transformed line without added brackets at both edges in original line
6. Transformed line without comment keyword at the beginning in original line

If there is an exact match, the line match data with the lines and indices given, is saved and the loop iterates forward. If not, parts of the t-line have to be compared with the current part of the o-line (`line 18, 19`). For identifying the match, there are column pointers, which mark the remaining part of the lines, that have not been matched yet. In order to compare the lines, different versions for the t-line were created (`line 20`). That means, depending on which addition is checked, characters were cut. T-lines with added brackets at the start are separated from their first character, t-lines with added brackets in the end are separated from their last character and so on. For every t-line version, it is checked, whether the remaining

Original program without whitespaces:

```
8  intAck(intm,intn)
9  {
10 if(m==0)returnn+1;
11 elseif(n==0)returnAck(m-1,1);
12 elsereturnAck(m-1,Ack(m,n-1));
13 }
```

Transformed program without whitespaces:

```
20 intAck(intm,intn){
21 if(m==0){
22 if(!(ASSERTION1)){
23 ERROR1
24 }
25 returnn+1;
26 }elseif(n==0){
27 if(!(ASSERTION2)){
28 ERROR2
29 }
30 returnAck(m-1,1);
31 }else{
32 if(!(ASSERTION3)){
33 ERROR3
34 }
35 if(!(ASSERTION4)){
36 ERROR4
37 }
38 returnAck(m-1,Ack(m,n-1));
39 }
40 }
```

Line match data:

```
{o-line 10: "if(m==0)returnn+1;"} :
[{t-line 21: "if(m==0)"}, {t-line 25: "returnn+1;"}]

["i": {t-line 21: o-column 1}, "r": {t-line 25: o-column 9}]
["i": {t-line 21: t-column 1}, "r": {t-line 25: t-column 1}]
```

Figure 3.7: Line match data

```
1 class LineMatchData:
2     original_program_line: str
3     original_program_line_number: int
4     line_matches: dict[int, str]
5     original_line_starting_points: dict[int, int]
6     tranformed_line_starting_points: dict[int, int]
```

Figure 3.8: Line match data object

o-line has fewer characters left than them, or not. If so, the t-line versions were reduced to the length of the remaining o-line (`line 21`). With these modifications,

```
1   class LineMatcher:
2       t-line_pointer: int
3       o-line_pointer: int
4
5       t-column_pointer: int
6       o-column_pointer: int
7
8       match_is_found: bool
9
10      match_data_list: list[LineMatchData]
11
12      def match_lines(o-lines, t-lines) -> list[LineMatchData]
13          while o-line_index < len(o-lines):
14              if o-line is empty:
15                  o-line_index++
16                  continue
17              while ! match_is_found:
18                  t-part = t-line[t-column_pointer:]
19                  o-part = o-line[o-column_pointer:]
20                  t_parts = create_versions()
21                  t_parts = reduce_to_o-part_length()
22                  if t-line is empty:
23                      t-line_index++
24                      continue
25                  if o-line == t-line:
26                      handle_exact_match()
27                  else if t-part in o-part:
28                       handle(regular part)
29                  else if (t-part without frist character) in o-part:
30                       handle(leading bracket)
31                  else if (t-part without last character) in o-part:
32                       handle(tailing bracket)
33                  else if (t-part without edge characters) in o-part:
34                       handle(surrounding brackets)
35                  else if (t-part without first two characters) in o-part:
36                       handle(comment)
37          return match_data_list
```

Figure 3.9: Line matching pseudo code

the cases are checked (`line 25-36`). If the algorithm works correctly, every original line has its transformed counterparts at the end of the loops.

3.3 Realignment

After the line matching, the data of the matched lines are referring to the program lines without whitespaces. With the help of the indices data created along data preparation, it is possible to correct the line mapping data (see Figure 3.10). To update the integers (indices), one has just to search for them in the indices data and assign the corresponding integer. To update the strings (= lines and line parts), differences between whitespace and non-whitespace columns of a character are used to calculate the missing whitespaces that are then added to recreate the unaltered lines. After the realignment, new indices data is generated that links every single character in the o-line with the corresponding character in the matched t-part (see Table 3.1).

Before: Line match data:

```
{o-line 10: "if(m==0)returnn+1;"} :
[{t-line 21: "if(m==0)"}, {t-line 25: "returnn+1;"}]


["i": {t-line 21: o-column 1}, "r": {t-line 25: o-column 9}]
["i": {t-line 21: t-column 1}, "r": {t-line 25: t-column 1}]
```

Index data lookup:

character	i	f	(	m	=	=	0	)	r	e	t	u	r	n	n	+	1	;
with whitespaces	3	4	6	7	9	10	12	13	15	16	17	18	19	20	22	23	24	25
without whitespaces	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
character	i	f	(	m	=	=	0	)	r	e	t	u	r	n	n	+	1	;
with whitespaces	3	4	5	7	8	9	10	11	5	6	7	8	9	10	12	13	14	15
without whitespaces	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	10

After: Line match data with whitespaces:

```
{o-line 10: "  if (m == 0) return n+1;"} :
       [{t-line 21: "  if (m == 0)"},
        {t-line 25: "    return n + 1;"}]


["i": {t-line 21: o-column 3}, "r": {t-line 25: o-column 15}]
["i": {t-line 21: t-column 3}, "r": {t-line 25: t-column 5}]
```

Figure 3.10: Realignment

character	i	f	(	m	=	=	0	)	r	e	t	u	r	n	n	+	1	;
original line nr.	10																	
original column nr.	3	4	6	7	9	10	12	13	15	16	17	18	19	20	22	23	24	25
transformed line nr.	21								25									
transformed column nr.	3	4	5	7	8	9	10	11	5	6	7	8	9	10	12	13	14	15

Table 3.1: Final index data

3.4 Replacement

Besides the replacement of `line` and `column`, which was prepared up to this point, other little value changes have to be undertaken. The specification should be overflow (`G ! overflow`) instead of reachability (`G ! call(reach_error())`) (see also the SV-COMP website [4]). The `input_file` value and `input_file_hash` keys both contain the name of the checked program and are also changed. The current naming convention of transformed programs is adding a "`transformed_`" prefix, thus removing it for the new witness could be easily done. But as this convention is very loose, the file name is derived from the original program name. A template snippet of which parts of the witness are transformed is shown in

Figure 3.11. A more challenging part comes with the line and column number

```
10 task:
11     input_files:
12     - "./transformed_program.c"
13     input_file_hashes:
14       "./transformed_program.c":
             "43616e20796f7520
             726561642074686973 3f"
15     specification: "G ! call(
           reach_error())"

22 location:
23     file_name:
           "./transformed_progam.c"
24     line: 25
25     column: 13
```

```
10 task:
11     input_files:
12     - "./program.c"
13     input_file_hashes:
14       "./program.c": "43616
           e20796f752072656164
            20746869733f"
15     specification: "G !
           overflow)"

22 location:
23     file_name:
           "./program.c"
24     line: 10
25     column: 23
```

Figure 3.11: Witness template snippet before and after transformation

changes. The algorithm for the `line` and `column` replacement can be sketched as follows: All `line` and `column` pairs are retrieved from the witness. Then, depending on the type of witness, different replacement algorithms are used.

Replacement Algorithm For Correctness Witnesses

For correctness witnesses, the following is done: For every `line` and `column` pair, the corresponding pair is searched via indices data, which was stored inside the line map object. If no match is found, there are two reasons, why:

1. The column number of the witness is invalid.
2. The line number points to a line which is not in the original program.

The first cause is handled by checking if the witness line number is found in a line map object. If the line number is matching, but no matching column number is found (`column 18 > column 15`) and the line number is not found in the next line map object (`line 26 > line 25`), it has to be an invalid `column` (see Figure 3.12).
The algorithm responds with replacing the column number with the first column number, which yields a character that is not a whitespace, thereby considering the `location` definitions for `invariant types`.
Otherwise, it handles the second cause and searches for the next t-part that has a link to the original program. This is done because it is likely that the line number points to an assertion. As assertions are always put the nearest above the respective operation and the line of this operation is the line we want it to point to, it will always be the next line with a match (because other assertions, which could be beneath the one pointed to, have no match in the original program). If the line number is pointing to line five or another of the first nine lines, it is clear that it can not have a match in the original program and can also be substituted by pointing to

Line and column of witness:

```
22 location:
23     file_name:
           "./transformed_progam.c"
24     line: 25
25     column: 18
```

Index data for original line nr. 10:

transformed line nr.	character	i	f	(	m	=	=	0	)		
21	original column nr.	3	4	6	7	9	10	12	13		
	transformed column nr.	3	4	5	7	8	9	10	11		
transformed line nr.	character	r	e	t	u	r	n	n	+	1	;
25	original column nr.	15	16	17	18	19	20	22	23	24	25
	transformed column nr.	5	6	7	8	9	10	12	13	14	15

Index data for original line nr. 11:

transformed line nr.	character	e	l	s	e	i	f	(	n	=	=	0	)					
26	original column nr.	3	4	5	6	8	9	11	12	14	15	17	18					
	transformed column nr.	5	6	7	8	10	11	13	14	16	17	19	20					
transformed line nr.	character	r	e	t	u	r	n	A	c	k	(	m	-	1	,	1	)	;
30	original column nr.	20	21	22	23	24	25	27	28	29	30	31	32	33	34	36	37	38
	transformed column nr.	5	6	7	8	9	10	12	13	14	15	16	18	20	21	23	24	25

Figure 3.12: Invalid witness column

an assertion. Because of this and therefore the redundancy, these `line` and `column` pairs are just deleted. The result is the transformed witness.

Replacement Algorithm For Violation Witnesses

The line matching between witness data and line match data follows the same approach as for the correctness witness. The `assumption` and `branching` `waypoint types` can also be completely handled by the correctness transformation procedure. The handling for `function_enter` and `function_return` checks, if the found column number points to a right parenthesis. If it does, continue. if not, the matched line is checked against right parenthesis'. If more than one parenthesis is found, the line is analysed by `C` function syntax cues. Else if no parenthesis is found, `-1` is inserted to show the invalid result.
As the `target` locations of a violation witness generated by verifying a transformed program point to the added assertion, the new column pointer cannot be deduced by the old one, but has to be computed by analysing the line, the new line pointer refers to. [4] To determine, which operation causes the overflow, when having multiple operations in the line, is supported by the new transformation instrumentation of the programs, which clearly identifies, where the problem occurs, as shown in

[4] As it was discussed with my mentor, the locations of `target` point to the operation that causes the overflow, despite the definition given in Section 2.5. My apologies, if there was an misunderstanding. If the pointing is handled as defined in [1], it is the same as for `assumption` and `branching`. With the following description, it is assumed, that the column number must point to the operation that causes the error.

Old transformation of two operations in one statement

```
4  void __VERIFIER_assert(int cond) {
5    if (!(cond)) {
6      ERROR: {reach_error();abort();}
7    }
8    return;
9  }

...

50 __VERIFIER_assert(ASSERTION1);
51 __VERIFIER_assert(ASSERTION2);
52       v = v + 2 * Y;
```

New transformation of two operations in one statement

```
54 if (!(ASSERTION1)) {
55   ERROR3: {reach_error();abort();}
56 }
57 if (!(ASSERTION2)) {
58   ERROR4: {reach_error();abort();}
59 }
60   v = v + 2 * Y;
```

Figure 3.13: Multi-operation handling in comparison

List of assertions:

[line 55, line 58]

List of operations:

[column 9, column 13]

Match by index:

index	assertion	operation
0	line 55	column 9
1	line 58	column 13

Figure 3.14: Assertion and operation listing

Figure 3.13. Every `reach_error` call provoked by an overflow condition has its own location for every operation in question (`line 55, 58 & 60`). While transforming the program, the assertions are placed above the corresponding expression in order of appearance of its operations. To determine, which operation caused the overflow, every line of assertion, which refers to the expression, is listed, as well as the column number of the operations in the expression. The index of the assertion in the assertion list the witness points to, is the index of the operation in the list of operations, which was searched for (see Figure 3.14).

4 Application

4.1 Research Questions

To check, whether the transformed witnesses represent the verification result of an unaltered program, the following questions can be posed:

1. Can reach-safety violation witnesses of the transformed programs be validated?
2. Can reach-safety correctness witnesses of the transformed programs be validated?
3. Can transformed no-overflow violation witnesses be validated?
4. Can transformed no-overflow correctness witnesses be validated?

As the validation of witnesses of transformed programs was not undertaken by Zheng 2024 [13], it is a presupposing task. Because of the differences of violation and correctness witnesses (see Section 2.5), it is split into two tasks, as well as the validation of the transformed witnesses. The separation is mandatory, because the validation of violation witnesses in `YAML` format with `CPAchecker` has not been implemented yet (see also [3]). Consequently, the violation questions must be discarded. Two research questions remain:

RQ1:
Can reach-safety correctness witnesses of the transformed programs be validated?

RQ2:
Can transformed no-overflow correctness witnesses be validated?

To tackle these, the experimental requirements and its results are described in the next sections.

4.2 Experimental Setup

The path leading to the validation of the witnesses consist of several steps. This chapter will show which steps are necessary, and which experimental setup can be used and was used in order to gain comparable results. Along the descriptions of the steps, console commands will be presented to support the recreation of the experiment. First of all, `CPAchecker` is the main tool required. It can be found at: `https://gitlab.com/sosy-lab/software/cpachecker`. Though the benchmarking tool `benchexec` can be executed with `CPAchecker`, it might

be worth noting the documentation: https://gitlab.com/sosy-lab/software/benchexec. The verification tasks used are listed in Figure 4.2.[1] The steps of generating results, including the evaluation, consists of five steps: Program Transformation, Verification, Witness Transformation, Validation and Comparison. Every step is explained and illustrated below (see Figure 4.1).[2]

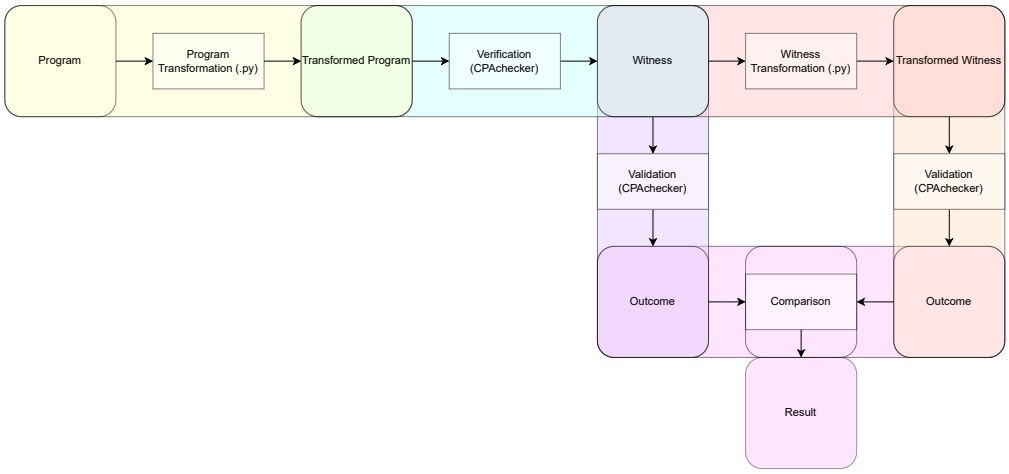


Figure 4.1: Steps of the experiment

- loop-acceleration
- loop-invgen
- loop-lit
- loops
- loops-crafted-1
- loop-simple
- loop-zilu
- memsafety
- memsafety-bftpd
- memsafety-ext3
- nla-digbench
- nla-digbench-scaling
- ntdrivers-simplified
- recursified_loop-simple
- recursive
- recursive-simple
- recursive-with-pointer
- seq-mthreaded
- signedintegeroverflow-regression
- termination-crafted
- termination-crafted-lit
- termination-memory-alloca
- termination-memory-linkedlists
- termination-nla
- termination-numeric
- termination-restricted-15

Figure 4.2: Used benchmark files

[1] Every item in the list of Figure 4.2 is linked to the corresponding folder in the SV-Benchmarks repository.

[2] Note, that not every input and output is shown in Figure 4.1. The purpose is to illustrate the workflow and the relevant results regarding the experiment.

Program Transformation

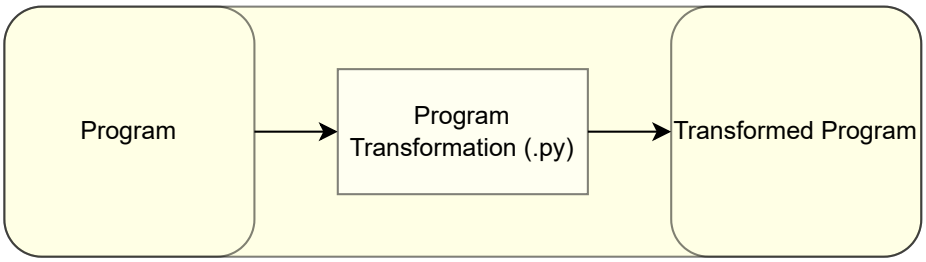


Figure 4.3: Program transformation step

The first step consists of taking the C programs in question and instrument them for reachability analysis (see Figure 4.3). The program transformation algorithm has not been integrated into `CPAchecker` yet. Currently, it can be found at: `https://gitlab.com/sosy-lab/software/specification-transformation`. The latest version can be pulled form the branch called "`transform_witness`". Though it is not integrated, it requires `CPAchecker` and especially its version on the `locate-candidates-to-overflow` branch in order to work properly. The execution command can be found beneath (see Figure 4.5) and requires some notes to consider:

- The argument of `--program` has to be a folder containing the program files and `.yml` task definitions (see Figure 4.4).
- To match the program with the witness, there are currently three patterns that can be used:
  - ***<file name>***
  - ***<original_<file ending>_<file name>>***
  - ***<intrumented_original_<file ending>_<file name>>***
- The whole algorithm includes the `CPAchecker` verification of the original program for processing reasons. Due to the restrictions of the script, it is not possible to generate multiple witnesses and therefore not suitable for mass verification
- The output contains all the results of a default configured verification, plus the transformed programs and task definition files as well as the original program files and its task definitions, provided that the file can be transformed.

```yaml
format_version: '2.0'

# old file name: array_false-unreach-call1_true-termination.c
input_files: 'array_1-1.c'

properties:
  - property_file: ../../../config/properties/no-overflow.prp
    expected_verdict: true
  - property_file: ../../../config/properties/unreach-call.prp
    expected_verdict: true

options:
  language: C
  data_model: ILP32
```

Figure 4.4: Task definition file example

```
cd "<repository>"
python3 src/specification-transformation-with-yml.py \
--from-property overflow \
--to-property reachability \
--algorithm Transform_Overflow_Algorithm \
--program \
"<folder of verifiable programs and task definitions>" \
--output-dir "<output directory>"
```

Figure 4.5: Transform program command

Verification

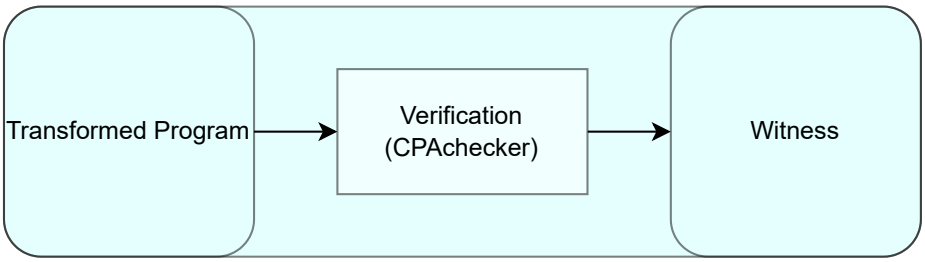


Figure 4.6: Verification step

Acquired by step one, the transformed programs are now verified with `CPAchecker` to get the witnesses (see Figure 4.6). New and old program transformations are compared. As the necessity of a new transformation algorithm is rooted in the old violation witness, whose line and column pointers do not provide useful information on where the code fails, it might be optional to only use the new transformation for files expected to result in false, while the old algorithm can be kept for correct programs. The results will show, if this is possible.

Bechmark Run

For comparable verification results of `CPAchecker`, it is recommended to use `benchmarkexec` or another reliable benchmark tool. It is possible to use `BenchCloud` for this[3] as it was done here with the setup shown in Table 4.1. For comparable testing, see also [9]. The command used for this experiment is shown in Figure 4.8.

resource limits:		**hardware requirements**:	
memory:	15000.0 MB	cpu model:	E3-1230
time:	900 s	cpu cores:	2
cpu cores:	2	memory:	15000.0 MB

Table 4.1: Setup for verification

Some notes regarding the reproduction of results:

- The set file in the required XML file contains the path to YML task definition files (see Figure 4.7).
- For further information on the XML file, see: https://gitlab.com/sosy-lab/software/benchexec/-/blob/main/doc/benchexec.md.
- An account might be required when using `Benchcloud`.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE benchmark PUBLIC "+//IDN sosy-lab.org//DTD BenchExec
       benchmark 1.9//EN" "https://www.sosy-lab.org/benchexec/
       benchmark-2.3.dtd">
3  <benchmark tool="cpachecker" timelimit="15 min" hardtimelimit="
       16 min" memlimit="15 GB" cpuCores="2">
4
5  <require cpuModel="Intel Xeon E3-1230 v5 @ 3.40 GHz"/>
6  <option name="-svcomp24"/>
7  <option name="-preprocess"/>
8  <option name="-setprop">counterexample.export.yaml=
       violation_witness.yml</option>
9  <option name="-setprop">cpa.arg.yamlProofWitness=
       corectness_witness.yml</option>
10
11 <rundefinition name="<custom name>">
12 <tasks name="<custom name>">
13 <includesfile><path to .set file></includesfile>
14 <propertyfile><path to>/unreach-call.prp</propertyfile>
15 </tasks>
16 </rundefinition>
17 <resultfiles>**/*witness*</resultfiles>
18
19 </benchmark>
```

Figure 4.7: XML file for benchmark verification

[3]Also called "VerifierCloud": https://vcloud.sosy-lab.org/cpachecker/webclient/master/info.

```
cd "<CPAchecker>"
scripts/benchmark.py \
--cloud \
--cloudUser "<username>" \
--cloudCPU 1230 \
--cloudPriority LOW \
--no-container \
--revision direct-witness-export:46261 \
--cloudMaster \
https://vcloud.sosy-lab.org/cpachecker/webclient/ \
"<XML file>"
```

Figure 4.8: Cloud benchmark verification command

Verification Results

To compare the performance of the transformation versions, two `BenchCloud` runs where executed, resulting in the numbers of Table 4.2. The results for the old transformed programs are slightly worse than the numbers presented in Zheng 2024 [13]. While the differences in correct and unknown results is negligible, the incorrect outcomes are a problem that has to be discussed.

run	total	correct	incorrect	unknown
witnesses presentend in Zheng 2024 [13]	856	687	2	167
witnesses of old program transformation	856	670	2	184
witnesses of new program transformation	856	660	5	191

Table 4.2: Performance of old and new program transformations

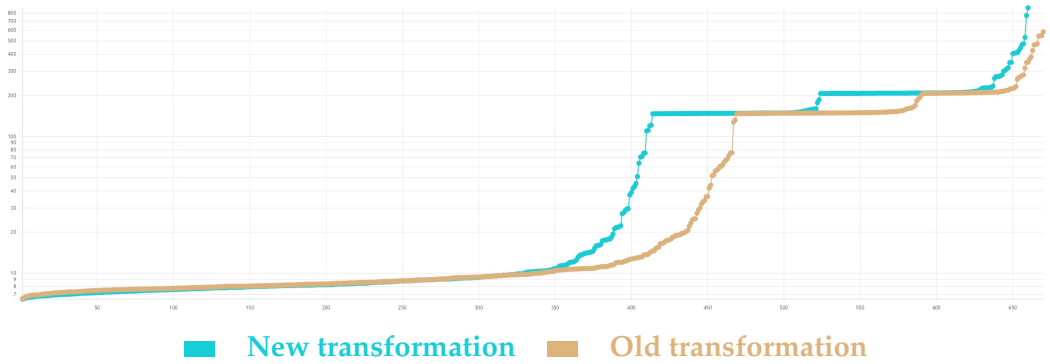
Examining the transformation of the three new incorrect files `hard2_unwindbound-50.c`[4], `hard2_unwindbound100.c`[5] and `b.05.c`[6], errors are easily spotted: the `unwindbound` files have a noticeably amount of assertion duplicates in a row. Some assertions themselves raise also suspicion like `(2 != 0 && (p != -2147483648 || 2 != -1))`. The transformed `b.05.c` code contains assertions statements inside another condition which is otherwise empty. It is unclear, why this is the case, as the changes to the transformation process should not have affected the core algorithm. They are in fact just substitutions at the end of it. Running the transformation in the current build without the substitution code lines, yields a comparable result. At some point of development, a side effect must have occurred. Unfortunately, the bug has not been detected yet. Because the old numbers are superior to the rerun of the old transformation, the former will be taken to be measured against in the

[4] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/nla-digbench-scaling/hard2_unwindbound50.c.

[5] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/nla-digbench-scaling/hard2_unwindbound100.c.

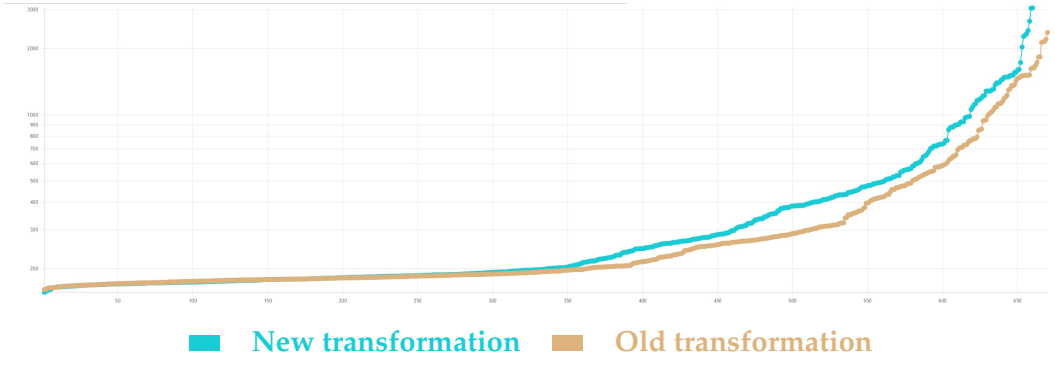
[6] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/termination-restricted-15/b.05.c.

validation step. For other benchmark comparisons with data only available for the rerun, these results are used, like in the following.



y-axis: *CPU time (s)* **x-axis:** *n-th fastest result*

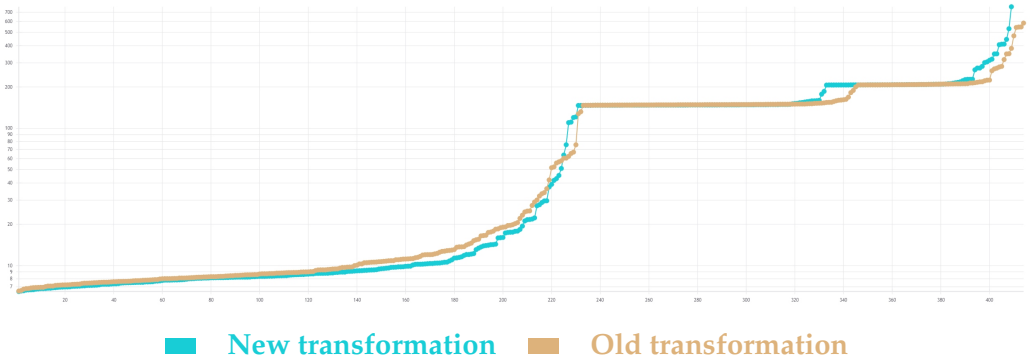
Figure 4.9: CPU time comparison of new and old program transformation



y-axis: *Memory consumption* **x-axis:** *n-th lowest consumption*

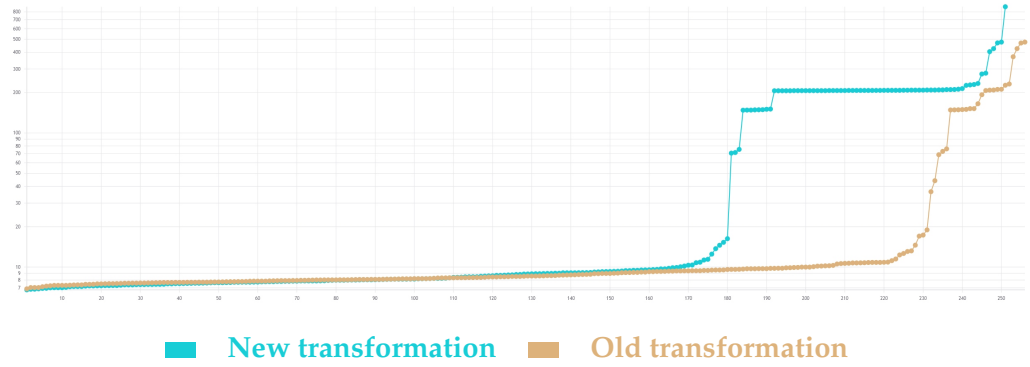
Figure 4.10: Memory comparison of new and old program transformation

Compared to the old transformation regarding other performance criteria, the new transformation has an observable negative offset in its graphs (see Figure 4.9 and Figure 4.10). Selecting by result type, it is possible to narrow down the cause of the problem (see Figure 4.11 and Figure 4.12). The results to be expected `false` show a steep increase in CPU time, while the results to be expected `true` only differ insignificantly between old and new. The option mentioned, to use the different transformation versions based on the expected result can thereby be excluded. If the violation witness problem is linked to the limitation on found `waypoint types`, it is a trace, which should be followed in the future for improvements. Though some analysis on the shortcomings of the results are required, the new transformation version is usable for the purpose of the experiment and future work in general.



y-axis: *CPU time (s)* **x-axis:** *n-th fastest result*

Figure 4.11: CPU time comparison filtered by the expected result: "true"



y-axis: *CPU time (s)* **x-axis:** *n-th fastest result*

Figure 4.12: CPU time comparison filtered by the expected result: "false"

total	**correct**	**incorrect**	**unknown**	*correct true*	*correct false*	*incorrect true*	*incorrect false*
856	660	5	191	409	251	2	3

Table 4.3: Verification results of the (new) program transformations

selection	**total**	**no content**	**value is "1"**	**relevant**
by relevance	530	360	1	169

Table 4.4: Results of the transformed correctness witness verification by relevance

It is useful to further examine the results of the new transformation in detail (see Table 4.3), because the number of correctness witnesses produced, differ from the results given. While there are 411 verifications resulting in `true`, 530 correctness witnesses were generated. To filter the relevant results, the correctness witnesses were sorted by relevance, which is measured by data of the `value` key of the `content` elements in the witnesses (see Table 4.4). Besides empty `content` sections that do not provide a proof and are therefore not relevant, `content` elements, where the `value` is `"1"` also yield diminished returns because an invariant is expected [1].

Due to the different verification results between old and new transformation, the amount of total witnesses is also different between them. Therefore, a comparison of relevance sorted witnesses is limited in significance. The relevance tables of reachability witness processing are shown in Section 4.3. As no generated violation witness has an empty `content` section, the measurement used for correctness witnesses can not be applied to violation witnesses. Despite the sorting by relevance, the whole witness set is used for transformation.

Witness Transformation

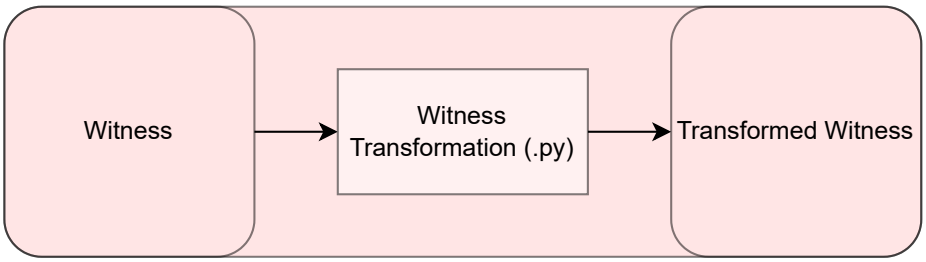


Figure 4.13: Witness transformation step

The witnesses produced by `CPAchecker` are now transformed (see Figure 4.13). To transform witnesses, the same repository and branch used for program transformation is needed.[7] For the transformation to work in the current state, the program files have to be named after the pattern ***<program file>*** and ***<transformed_<program file>>*** (see Figure 4.14).

```
cd "<repository>"
python3 src/entrance/witness_transformation_main.py \
--witness_directory "<witness directory>" \
--original_program_directory "<original program directory>" \
--program_diectory "<program directory>" \
--output_directory "<output directory>" \
```

Figure 4.14: Witness transformation command

To get insights about the quality of the transformed witnesses, they must be validated.

Validation

Both, the "original" and the transformed witnesses are now validated (see Figure 4.15). When validating witnesses of transformed programs, it might be necessary to delete their `file_hash` keys first. For now, this step is already included in the

[7] "`transform_witness`" from: https://gitlab.com/sosy-lab/software/specification-transformation.

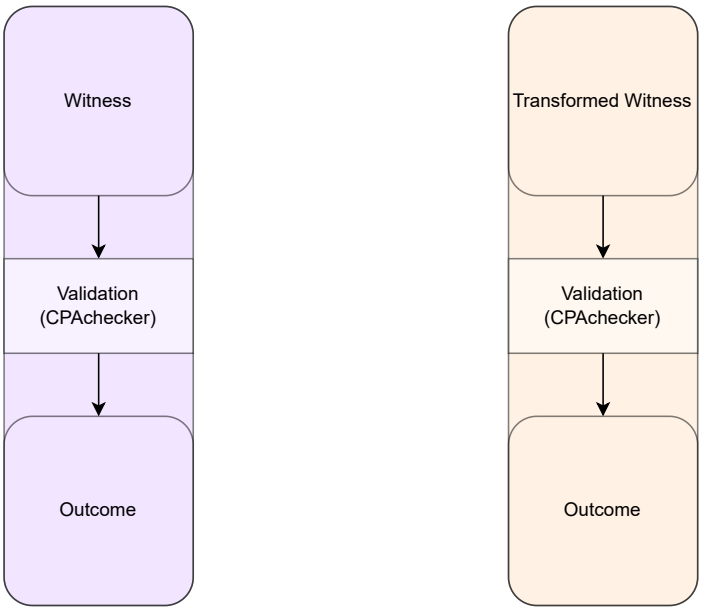


Figure 4.15: Validiation step

```
cd "<CPAchecker>"
./bin/cpachecker "<file to be validated>" \
--preprocess \
--option witness.checkProgramHash=false \
--heap 5000m \
--option cpa.predicate.memoryAllocationsAlwaysSucceed=true \
--timelimit 900s \
--witness "<witness>" \
--<validation property>

# Validation property options:
# No-overflow:     <witness type>-witness-validation--overflow
# Reachability:     <witness type>-witness-validation
```

Figure 4.16: Single file validation command

witness transformation procedure. This might change in the future. As in the verification step, the validation can be done on a single file or with benchmark grouping. For comparable validation results, its again recommended to use `benchexec` or a similar tool. Unfortunately, student accounts have no permission to execute validation benchmark runs on BenchCloud. Therefore, a local single validation of the files was necessary (see Figure 4.16).

4.3 Results

Validation of reachability correctness witnesses of transformed programs

selection	total	no content	value is "1"	relevant
by relevance	548	327	10	211

Table 4.5: Results of the reachability correctness witness verification by relevance

run	total	true	false	unknown	timeout
reachability correctness witness validation	548	477	0	35	36

Table 4.6: Results of the reachability correctness witness validation

	true	false	unknown	timeout	total
no content	337	0	13	4	354
value is "1"	0	0	0	0	0
relevant	140	0	22	32	194
total	477	0	35	36	548

Table 4.7: Results of the reachability correctness witness validation by relevance

Validation of transformed no-overflow correctness witnesses

run	total	true	false	unknown	timeout
transformed correctness witness validation	530	480	0	2	48

Table 4.8: Results of the transformed correctness witness validation

	true	false	unknown	timeout	total
no content	357	0	1	9	367
value is "1"	1	0	0	0	1
relevant	122	0	1	39	162
total	480	0	2	48	530

Table 4.9: Results of the transformed correctness witness validation by relevance

4.4 Evaluation

The results (see Table 4.6, Table 4.7, Table 4.8 and Table 4.9) are now evaluated (see Figure 4.17).

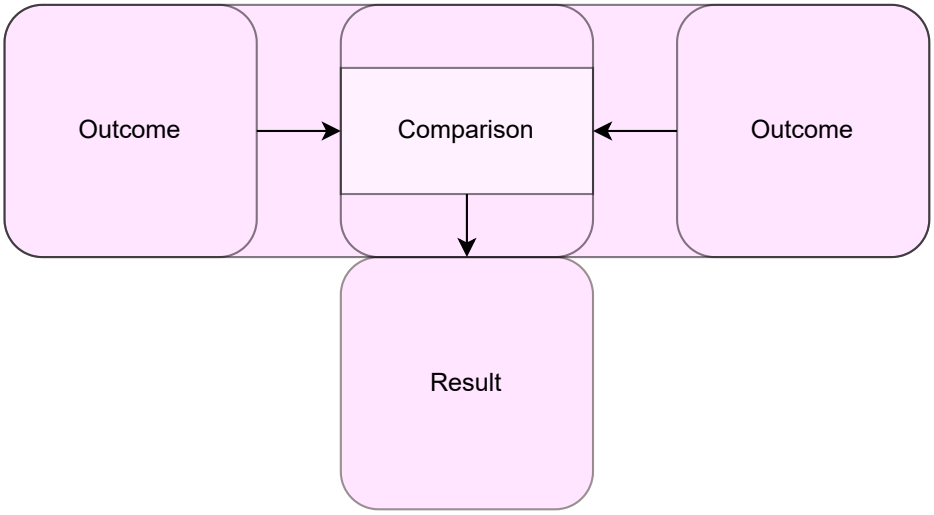


Figure 4.17: Comparison step

Research Question 1

The numbers of both validation runs can affirm the first research question.

Reachability witness validation:
About 87% (477 of 548) of all reachability correctness wintess tasks were validated as true. Ignoring the irrelevant items in Table 4.5, 66% (140 of 211) of all relevant witnesses produced were correct. 72% of the relevant witnesses were validated as correct.
Transformed no-overflow witness validation:
About 90% (480 of 530) of all no-overflow correctness witness tasks were validated as true. Taking the erroneously generated or empty witnesses into account by opposing the relevant witnesses, reveals a 72% success rate (122 of 169) regarding the verified relevant correctness witnesses and a 75.3% rate (122 of 162) regarding the total validated relevant correctness witnesses.

Research Question 2

Comparing the validations of the correctness witnesses, their difference is insignificant. The numbers of the validation of the reachability witnesses should be the upper boundary for the transformed witness validation. As this does not seem to be the case, the `unknown` results, for example, may be caused by the unreliable validation process. But even if they should be validated as true, there is no remarkable gap in between the numbers. This can be evaluated as a positive result regarding the witness transformation algorithm and the second research question.

5 Conclusion

5.1 Witness Transformation Algorithm

A witness transformation algorithm was provided that can transform both correct and violation witnesses. Though the algorithm works, there are some aspects that are still open for discussion and improvement. The important ones are listed here:
The matching of lines which consist of one single closing curved bracket could be one off, if brackets were added with the transformation. A fix is easy to implement, but has not been done yet, because the brackets are never a pointer target and thus are not relevant.
The tabs inside the program are not consistent across its original and transformed versions. This might be a problem when dealing with the column references, as they differ, depending on whether there is one tab or four whitespaces, because the first counts as one character while the latter counts as four. Therefore, when comparing the witness of a standard overflow verification and a transformed witness, where the tabs are substituted by four whitespaces, the column values can differ.
The absence of `waypoint types` in the violation witnesses leads to a lack of real world test cases for handling these cases. If the absence is an unavoidable side effect of reachability conversion, the handling becomes irrelevant. If not, tackling this issue becomes a future task.
A clarification should succeed this thesis, whether the `target waypoint location` points to an operation or a full expression or statement. Though this does not affect the code writing, because the cases are covered, it will lead to wrong column numbers in the witness, if the algorithm is not changed accordingly.
Integrating the algorithm in a bigger framework or refactoring it to raise the level of abstraction and modularization is a conceptual change that is possible future work (see Ovezova 2024 [11]).

5.2 Verification And Validation Results

Being currently restricted to the limited tool features, which can handle the new `YAML` witness format, leads to open tasks regarding the validation of transformed programs and witnesses, mainly validating correctness and violation witnesses with reliable benchmark runs.
The limitations on `waypoint types` observed in the violation witnesses is a finding, which has to be discussed in the aftermath of this thesis.

The undesired aspects of the current program transformation build should be analysed and fixed, in order to identify, whether they are the cause of the incorrect verification results.
Though there are bugs in the code that might affect the results as well as the mentioned shortcomings in execution, the outcome shows an overall working of both, the program and the witness transformation. They also confirm that a reachability analysis of overflow errors is possible. With these conclusions a contribution to the approach of converting properties is given that can be build upon in future works.

Bibliography

[1] P. Ayaziová, D. Beyer, M. Lingsch-Rosenfeld, M. Spiessl, and J. Strejček. Software verification witnesses 2.0. In *Proc. SPIN*, LNCS . Springer, 2024.

[2] D. Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 2015.

[3] D. Beyer. State of the art in software verification and witness validation: SV-COMP 2024. In B. Finkbeiner and L. Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*, volume 14572 of *Lecture Notes in Computer Science*, pages 299–329. Springer, 2024.

[4] D. Beyer. Sv-comp 2024: Call for participation — procedure. `https://sv-comp.sosy-lab.org/2024/rules.php`, 2024. Accessed: 2024-07-16.

[5] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Correctness witnesses: exchanging verification results between verifiers. In T. Zimmermann, J. Cleland-Huang, and Z. Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 326–337. ACM, 2016.

[6] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig. Verification witnesses. *ACM Trans. Softw. Eng. Methodol.*, 31(4), sep 2022.

[7] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In E. D. Nitto, M. Harman, and P. Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 721–733. ACM, 2015.

[8] 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.

[9] D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transf.*, 21(1):1–29, 2019.

[10] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.

[11] A. Ovezova. Witness modifications for program transformations: A case study on side-effect removal. Bachelor's thesis, Ludwig-Maximilians-Universität München, Software Systems Lab, 2024.

[12] N. Piterman and A. Pnueli. Temporal logic and fair discrete systems. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 27–73. Springer, 2018.

[13] X. Zheng. O2r: Reduction of no-overflow property of c programs to unreach-call property. Bachelor's thesis, Ludwig-Maximilians-Universität München, Software Systems Lab, 2024. Unpublished.