LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
CHAIR FOR SOFTWARE AND COMPUTATIONAL SYSTEMS

T2R: Reduction of Termination of C Programs to Reachability-Safety Problem

Tian Xia

Bachelor Thesis

Supervisor: Prof. Dr. Dirk Beyer

Mentor: Marek Jankola

Submission Date: March 13, 2024

Statement of Originality

I confirm that this bachelor thesis is my own work. In addition to those sources and materials cited in the text, I only used ChatGPT to generate and improve wordings of single sentences and small paragraphs.

Munich, March 13, 2024 Tian Xia

Acknowledgments

I would like to express my sincere gratitude to my mentor, Marek Jankola, for his invaluable guidance and support throughout my thesis work. Especially, he was always available via Zulip to answer my questions. Further thanks to my girl friend for her emotional support durch this period. At the end, I would also like to thank the Software and Computational Systems Lab for providing me with this topic, which helped me gain valuable insights into software verification.

Abstract

Program termination analysis is crucial in various contexts, such as embedded systems and safety critical software. Traditionally, proving program termination involves searching for ranking functions and validating the ranking functions, which can be expensive. However, reachability algorithms are more tuned than termination analysis techniques, because of far more tasks in the software verification competition and more variety of usage in the practice. In this thesis, we propose a new approach, named T2R, which transforms the termination of C programs to a reachability-safety problem. In this way, we could leverage various fined-tuned reachability analysers. We evaluate T2R by combining it with the best tool for reachability analysis and compare the combination with the state-of-the-art tools in termination analysis. Our results show that the combination of T2R and the best reachability analyser is more efficient and at least 10% more effective than other tested termination techniques.

Contents

1 Introduction

Program termination analysis is necessary in many settings, such as embedded systems and safety critical software [4]. Non-termination bugs can lead to performance problems or denial-of-service attacks [12]. The general approach for proving program termination is to search ranking functions and validate them. However, the search of ranking functions can be costly, whether in space or in time. For example, when dealing with complex systems, some techniques may exhaust the available memory before a ranking function can be found [6]. Moreover, in many cases, we must search for lexicographic ranking functions instead of simple linear ranking functions because they are not powerful enough [7]. This makes the search even harder. On the other hand, the reachability algorithms are more tuned than termination analysis techniques because of far more tasks in SV-COMP [1] and wider variety of usage in practice. For example, in SV-COMP 2024, there are 11305 tasks for reachability analysis, but only 2354 tasks for termination proving. Based on the theory proposed from the paper *Livess Checking as Safety Checking* [3], we propose a novel approach for termination proving, which can leverage various fined-tuned reachability analysers and avoid constructing expensive ranking functions.

This approach works by reducing the termination of C programs to a reachability-safety problem. In more concrete terms, it takes a C program and instrument every loop within it, including adding an assertion to each loop, so that the original program does not terminate iff the newly added assertions in its instrumented version can be violated. For example, in Figure 1, the left side shows an C program for termination analysis, while the right side presents its instrumented version. The non-termination of the original program is equivalent to the existence of an input for which the assertion in its instrumented version does not hold.

For implementation, we first extract the loop locations and the variables inside these loops from the program to be verified, and then instrument the program based on the information obtained. As an attempt, this approach deals with C programs that only contain data types: char, int, short, long, float, double, and pointers, as well as only the while loop as loop structure.

We evaluated our tool by answering two research questions:

- RQ1: Which tool for reachability analysis is the best for instrumented

[1] `https://sv-comp.sosy-lab.org/2024/index.php`

```c
int main() {
  int x = nondet_int();

  while (x > 0) {
    x = x - 1;
  }

  return 0;
}
```

(a) Original C program

```c
int main() {
  int x = nondet_int();

  int x1;
  int i1 = 0;
  while (x > 0) {
    assert(i1 == 0 || x != x1);
    if (i1 == 0 && nondet_int()) {
      x1 = x;
      i1 = 1;
    }
    x = x - 1;
  }

  return 0;
}
```

(b) Instrumented version

Figure 1: Simple example for T2R

program?

- RQ2: Is the combination of T2R tool and the best tool for reachability analysis of instrumented program more effective and efficient than current termination analysis techniques?

Our results on a set of 355 instrumented programs showed that UAutomizer [10] with approach trace abstraction is the best tool for reachability analysis of instrumented program. It has the highest number of correct results, with 298 (84%). The second highest number of correect results is 294 (83%), achieved by CPAchecker [1] with approach predicate Analysis with linear arithmetics, and its average performance of each task is even approximately 10 s better than UAutomizer with trace abstraction, but it has 10 more incorrect results.

In the termination analysis, which is conducted on the original ones of those 355 instrumented programs, the combination of our T2R tool and UAutomizer with trace abstraction is more effective and efficient than CPAchecker and UAutomizer with termination analysis approaches. It has the highest number of correct results with 298 (32%), which is 57 (16%) more than CPAchecker and 36 (10%) more than UAutomizer.

Therefore, we believe that our approach demonstrates a good direction for solving program termination analysis.

2 Related Work

As mentioned in the introduction, the general approach for termination analysis includes the search of ranking functions and the validation of the ranking functions. Scientific work *Synthesis of Linear Ranking Functions* [6] shows a concrete approach to do it this way. First, it extracts a set of linear expressions bounded inside the loop from some loop invariant. Second, it derives a set of linear expressions that decrease discretely around the loop from the loop's transition relation. The third step then is to compute the intersection of these two sets. Any expression in the intersection acts as a ranking function, and thus proves termination of the loop. This has advantage that simple programs are easily handled, but programs with complex control flows often exhaust the available memory before a ranking function can be found.

Other than finding ranking functions, Scientific work *Loopster: Static Loop Termination Analysis* [12] proposes a lightweight analysis-based approach to prove program termination. It employs a divide-and-conquer approach:

1. It derives individual paths from a target multi-path loop and analyze the termination of each path.

2. It analyzes the dependencies between every two paths.

3. It determines the termination of the entire loop based on the relations among paths.

Surprisingly, it shows 20×+ performance improvement compared to the state-of-the-art tools, even if only considering those correctly analyzed programs. This owes to its static analysis. However, for the same reason, it cannot handle the loops that contain complex data structures(e.g., arrays, heaps), function calls, and variables that are updated by complex computation(e.g., bitwise calculator). By the way, these obstacles, except for the first one, can be easily overcome by our T2R tool.

3 Background

3.1 Theoretical Background

3.1.1 Termination and Reachability Analysis

T2R reduction stands for reduction from termination analysis to reachability analysis. *Termination analysis* is a program analysis that determines whether a program halts for every input, while *reachability analysis* determines whether an assertion condition holds at a concrete location of the program for every input.

3.1.2 Transition System

When explaining the theory behind T2R reduction, we will introduce a mathematical model known as transition system. This model allows the termination condition of a program to be transformed first into an intermediate equivalent condition, and then into a reachability problem.

A *transition system* [5] is a pair $(S, \rightarrow)$, where S is a set of states that respectively consist of a program location, representing the current value of the program counter (e.g., the line number of the next operation to be performed), and a valuation of the program variables declared so far, and $\rightarrow \subset S \times S$ is a set of transitions. It is considered finite if the set S is finite. A trace t of it with initial state s_0, is a (possibly infinite) sequence of states $s_0, s_1, s_2, ...$ such that for each s_{i+1} with $i \in \mathbb{N}$, it holds $s_i \rightarrow s_{i+1}$. Every program possesses a distinct transition system that models all potential executions of the program and we call it underlying transition system. In Figure 2, there is a non-terminating C program in 2a and the visualization of its corresponding transition system in 2b.

Last but not the least, T2R reduction operates under the assumption that the program to be verified has only variables with finite domain so that its underlying transition system is finite.

3.1.3 Control Flow Automaton

For instrumentation, we need information about every loop in the program to be verified. However, identifying all the loops in the source code can be challenging, especially when dealing with complicated loop structures such as nested loops, loops embedded within functions, or recursion. Therefore,

we parse the program as CFA, which models the control flow of the program, making it easier to identify loops.

A *control flow automaton* (CFA) [5] is a tuple $G = (L, l_0, E)$. It consists of a finite set $L = \{l_0, ..., l_n\}$ of program locations, a program entry $l_0 \in L$, and a set $E \subset L \times Ops \times L$ of edges, where Ops denotes the set of all operations within the program. Figure 2c provides a graphical representation of a CFA constructed from the C program in 2a.

3.2 Practical Background

3.2.1 assert

To add diagnostics to programs, we employ the `assert` [11] macro in C. It is declared as follows: `void assert(int condition)`. If the condition evaluates to zero, the macro `assert` will print an error message on `stderr` (standard error stream to display error messages and diagnostics) and aborts program execution. Otherwise, it will do nothing.

3.2.2 CPAchecker

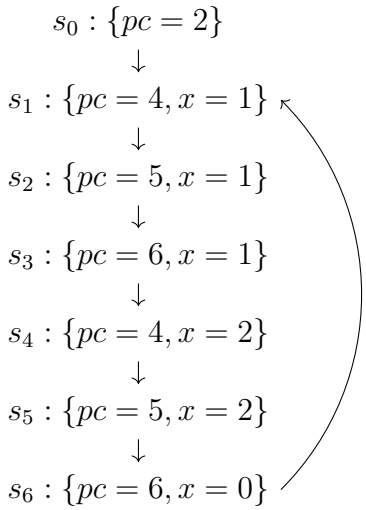
CPAchecker is a tool and framework for configurable software verification. It can perform both reachability and termination analysis. For reachability analysis, its approaches include predicate analysis (PD), predicate analysis with linear arithmetics (PDL), bounded model checking (BMC), interpolation-based model checking (IMC), and kInduction (KID). For termination analysis, it uses a lasso-based approach (LSB_C) which uses the tool LassoRanker. Before a program analysis starts, the input program is firstly transformed into a syntax tree, and further into a set of CFAs, which is the central data structure of CPAchecker. Therefore, we plan to use CPAchecker to obtain the CFA of the program to be verified.

3.2.3 UAutomizer

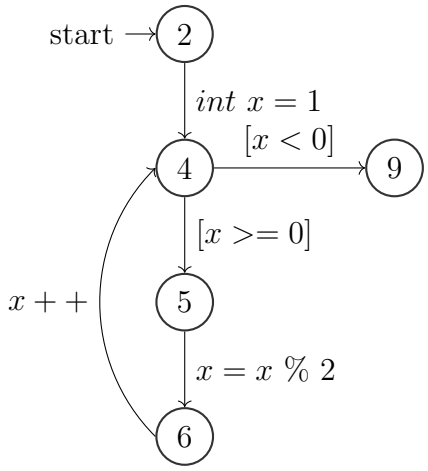
UAutomizer is a software verification tool for C Programs. Similar to CPAchecker, it can also perform reachability and termination analysis, but with noticeably different performance. For reachability analysis, it uses trace abstraction (TAS). For termination analysis, it also uses a lasso-based approach (LSB_D), but with a different algorithm from CPAchecker's.

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

(a) C program



(b) Transition system



(c) CFA

Figure 2: Example for representing C program by transition system and CFA

3.2.4 CPV

CPV is a circuit-based program verifier for C,[2] which can perform reachability analysis. It functions by converting a C program into a circuit and then running a hardware verifier (HWV) on it. The hardware verifier (HWV) can

[2] `https://zenodo.org/records/10203472`

be ABC[3] or AVR [9].

3.2.5 SV-COMP Benchmark Set

For evaluation of our approach, we use verification tasks from the SV-COMP benchmark set.[4] It is constructed and maintained as a common benchmark for evaluating the effectiveness and efficiency of state-of-the-art verification technology. The verification tasks for C programs in `c/` are classified into (sub)categories as defined by SV-COMP.[5] Each (sub)category is defined by a `.set` file that contains patterns specifing a set of programs.

A C verification task contains a `.c` or `.i` file and a `.yml` file as shown in Figure 3. The `input_files` specifies the subject program `example.c`. The `properties` section lists the properties that should be checked for this program. Each property is associated with a property file containing its definition and an expected verdict. The property file `termination.prp` relates to termination analysis, while `unreach-call.prp` relates to reachability analysis. In this example, it is exepcted that the subject program does not terminate and an assertion condition in it does not hold for certain inputs.

```
format_version: '2.0'
input_files: 'example.c'
properties:
  - property_file: ../properties/termination.prp
    expected_verdict: false
  - property_file: ../properties/unreach-call.prp
    expected_verdict: false

options:
  language: C
  data_model: ILP32
```

Figure 3: Example of `.yml` file

[3] `https://people.eecs.berkeley.edu/~alanmi/abc/`
[4] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks`
[5] `https://sv-comp.sosy-lab.org/2024/benchmarks.php`

4 T2R Reduction

In this section, we describe T2R reduction at a high level. First, we elucidate its underlying theory. Then, we present four examples of real instrumentation to solidify understanding.

4.1 Theory

A program does not terminate iff there exists an infinite trace in its transition system. Since we restrict the transition system to be finite, such a trace can always be assumed to be lasso-shaped, denoted as $s_0, s_1, ..., s_{l-1}, (s_l, s_{l+1}, ..., s_{l+k})^\omega$, where $l, k \in \mathbb{N}_0$ with $l \leq k$, and ω indicates that the sequence $s_l, s_{l+1}, ..., s_{l+k}$ repeats infinitely. Figure 4 illustrates a sub-part of a transition system, on which such a trace can be executed. Thus, whether a program terminates is equivalent to the existence of a lasso-shaped trace in its transition system. To detect it, it is sufficient to find a trace in which a state occurs, or in other words, is encountered, at least twice. This is because if a program can reach the same state twice, it can reach it infinitely many times, although not necessarily in the same execution. Since only the states with program locations within loops can potentially be encountered twice, we only check each loop to see if there is a state that can be revisited.

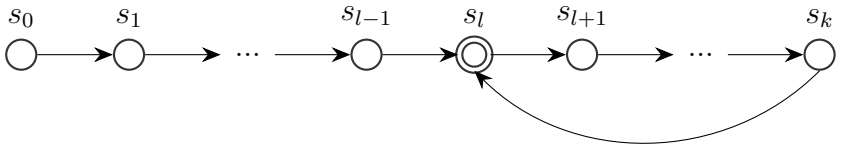


Figure 4: Example for a generic lasso-shaped trace in transition system

For this purpose, we employ the method *State Recording Translation*, whose basic idea is to detect a previously seen state by saving it beforehand. It is illustrated by a generic instrumentation of a given loop in CFA, as dipicted in Figure 5. Firstly, we look at the loop to be instrumented in Figure 5a, which, for simplicity, contains variables only of integer type. In order to find a state that can be seen twice, it is sufficient to detect it only at the location l_1. This is because if the program passes through l_1 infinitely many times and each time reaches a different state, then its underlying trantistion system would not be finite. Since we do not know whether the current state

at l_1 will be encountered again later, we use an oracle to tell us whether to save it. After saving, each time the program passes through l_1, we check if the current state is the same as the saved one. If yes, we have found such a state.

Now we look at the instrumented loop in Figure 5b. The integer variables $x_0^L, ..., x_n^L$ are used to save the values of the variables $x_0, ..., x_n$ inside the loop, respectively. The boolean variable $saved_L$ indicates whether the saving has been performed. The oracle is represented by the function $NDBool()$. If $saved_L == false$ and $NDBool() == true$, meaning that we have yet to save the values of the variables inside the loop and the oracle instructs us to do so, we perform the operation $op_{s=s'}$. Otherwise, we do nothing. After saving, we verify if a state at l_1 is reachable, where the values of variables $x_0, ..., x_n$ are equal to those of $x_0^L, ..., x_n^L$, respectively, as asserted by π_L. If so, we transition to the error state l_e. Thus, the termination analysis of a program has been transformed into a reachability analysis.

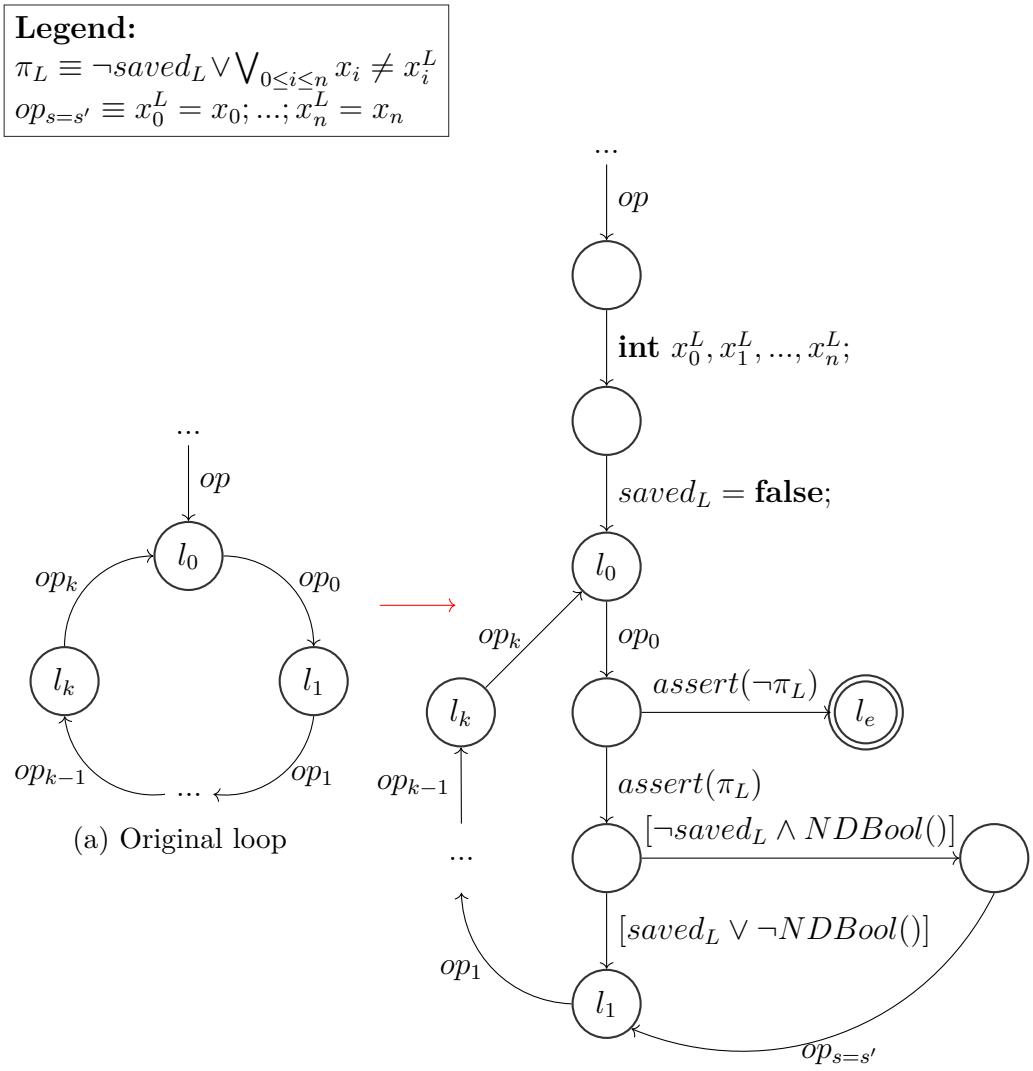
4.2 Examples of C Program Instrumentation

Now, let's take a look at the four examples of real instrumentation. The first two examples represent typical cases, while the last two corner cases.

4.2.1 Typical Cases

The program to be verified in Figure 6a contains two consecutive while loops. Thus, instead of detecting a state that will recur later only at one location, we need to search for such a state at two different places: the line of code $x = x-1$ and the line of code $y = y+1$. Now, we look at the instrumentation in Figure 6b. For the first loop, the variable x_1 is declared to save the value of the variable x, and the variable $i1$ signifies whether the saving has occurred. A value of 0 means false, otherwise true. Moreover, the oracle is represented by the function *nondet_int*() in the if construct at line 9, and it returns non-deterministic integer value. A value of 0 means that we will not save the current value of x, otherwise we do. The instrumentation of the second loop is the same.

In Figure 7, the program to be verified, however, contains two nested loops. In this case, we detect a state that will be encountered later at the lines of codes $x = x - 1$ and $y = y + 1$. Moreover, the outer loop has two variables used inside, namely x and y, while the inner loop only has y. This



(a) Original loop

(b) Instrumented loop

Figure 5: Example for a generic instrumentation of a given loop in CFA. Further, NDBool() is a function that returns nonterministic boolean value, and $x_0, ..., x_n$ are the variables used in the original loop.

implies that when instrumenting the outer loop, we must record the values of two variables, whereas for the inner loop, only the value of y needs to be recorded.

```c
int main() {
  int x = nondet_int();
  int y = nondet_int();

  while (x > 0) {
    x = x - 1;
  }

  while (y < 0) {
    y = y + 1;
  }

  return 0;
}
```

(a) Original C program

```c
int main() {
  int x = nondet_int();
  int y = nondet_int();

  int x1;
  int i1 = 0;
  while (x > 0) {
    assert(i1 == 0 || x != x1);
    if (i1 == 0 && nondet_int()) {
      x1 = x;
      i1 = 1;
    }
    x = x - 1;
  }

  int y2;
  int i2 = 0;
  while (y < 0) {
    assert(i2 == 0 || y != y2);
    if (i2 == 0 && nondet_int()) {
      y2 = y;
      i2 = 1;
    }
    y = y + 1;
  }

  return 0;
}
```

(b) Instrumented version

Figure 6: Typical case 1 of instrumentation

4.2.2 Corner Cases

The original program in Figure 8a closely resembles that of Figure 1a, but with the difference that the condition of the while loop has side effect. At first glance, it may cast doubt on the effectiveness of the previous instrumentation approach for this program; however, it is indeed equally effective. The reason is as follows. The line of code $x = x-1$ is the location to detect the state that can be encountered twice. If the loop does not terminate and we cannot find such a state at the location, it means that every time the program reaches the location, it has a different state. This would result in the underlying

```
1  int main() {
2    int x = nondet_int();
3    int y = nondet_int();
4
5    while (x > 0) {
6      x = x - 1;
7
8      while (y < 0) {
9        y = y + 1;
10     }
11   }
12   return 0;
13 }
```

(a) Original C program

```
1  int main() {
2    int x = nondet_int();
3    int y = nondet_int();
4
5    int x1;
6    int y1;
7    int i1 = 0;
8    while (x > 0) {
9      assert(i1 == 0 ||
10     !(x == x1 && y == y1));
11     if (i1 == 0 && nondet_int()) {
12       x1 = x;
13       y1 = y;
14       i1 = 1;
15     }
16     x = x - 1;
17
18     int y2;
19     int i2 = 0;
20     while (y < 0) {
21       assert(i2 == 0 || y != y2);
22       if (i2==0 && nondet_int()) {
23         y2 = y;
24         i2 = 1;
25       }
26       y = y + 1;
27     }
28   }
29   return 0;
30 }
```

(b) Instrumented version

Figure 7: Typical case 2 of instrumentation

transition system of the program not being finite.

In the second corner case, the previous instrumentation approach, however, needs to be adjusted for the program shown in Figure 9a because there is a variable y declared inside the while loop. If we instrumented this program in the same manner as before, we would save the value of y in the body of the if construct, but at this point, y has yet to be defined. Therefore, a question arises: can we ignore the variables declared inside the loop when

```
1  int main() {
2    int x = nondet_int();
3
4    while (x++ > 0) {
5      x = x - 1;
6    }
7    return 0;
8  }
```

(a) Original C program

```
1  int main() {
2    int x = nondet_int();
3
4    int x1;
5    int i1 = 0;
6    while (x++ > 0) {
7      assert(i1 == 0 || x != x1);
8      if (i1 == 0 && nondet_int()) {
9        x1 = x;
10       i1 = 1;
11     }
12     x = x - 1;
13   }
14   return 0;
15 }
```

(b) Instrumented version

Figure 8: Corner case 1 of instrumentation

instrumenting? For instance, a loop includes three variables: a, b, and c. a and b are defined outside the loop, while c is defined only inside the loop. Is it possible that, at a location within the loop, a and b return to the same state but c never visits the same value twice? If so, then we would claim the program as non-terminating if we do not consider c in the assertion condition and the if construct. Fortunately, such a scenario can never happen because c can get only its value from a, b, some constants or non-deterministic choice. If c depends on:

- a and b, then c will behave the same as before, since they are set to the same value.
- constant, then c behaves the same in every iteration.
- non-deterministic choice, then c can be set the same as in the previous iterations where a and b reached the same state.

Thus, if a and b visit the same state twice, c can have the same behavior as it had in the previous iterations and therefore also visit the same value twice.

```c
int main() {
  int x = nondet_int();

  while (x > 0) {
    x = x - 1;

    int y =
    nondet_int();
    if (y == 1) {
      break;
    }
  }
  return 0;
}
```

(a) Original C program

```c
int main() {
  int x = nondet_int();

  int x1;
  int i1 = 0;
  while (x > 0) {
    assert(i1 == 0 || x != x1);
    if (i1 == 0 && nondet_int()) {
      x1 = x;
      i1 = 1;
    }
    x = x - 1;

    int y = nondet_int();
    if (y == 1) {
      break;
    }
  }
  return 0;
}
```

(b) Instrumented version

Figure 9: Corner case 2 of instrumentation

5 Implementation

In this chapter, we delve into the implementation of our approach. We begin by enumerating the assumptions of the program to be verified and the dependencies required for running our implementation. Next, we explain how to run the implementation. Finally, we provide an overview of the implementation process, followed by a discussion of the major challenges encountered during the implementation.

5.1 Requirements

5.1.1 Assumptions of Program to be Verified

The program to be verified is coded in C and only supports the following data types: char, int, short, long, float, double, and pointers. Additionally, it exclusively contains the while loop as its loop structure.

5.1.2 Dependencies for Running Implementation

Before running the implementation, we need to have four dependencies installed:

- Python 3.10.12 with standard library
- clang-format 14.0.0 (a C formatter)
- CPAchecker (based on the revision 45859 of the *locate-loop-and-live-variables* branch, which is hosted at `https://svn.sosy-lab.org/software/cpachecker/branches/locate-loop-and-live-variables/`)
- Java 17 or later

5.2 Usage of Implementation

The project **specification-transformation** is responsible for running our implementation. It serves as a versatile framework designed to transform a program's property using a specified algorithm. To use it, we execute the Python script `specification-transformation.py` in `src`. Its detailed usage is outlined in Table 1. This table describes the command syntax for running the script, with explainations for each component provided below. For instance, if we are in the project's directory and wish to instrument the C program `test_program.c` located in the subdirectory `tests` using our `t2R_algorithm.py` algorithm, we should set the argument PROGRAM to `tests/test_program.c`, the options –from-property to `termination`, –to-property to `reachability`, and –algorithm to `T2RAlgorithm`, as shown in Figure 10. Additionally, since we do not specify the output directory, the transformed program is outputted to the default output directory `output`, rataining the same name as before.

```
tian@Tian:~/Projects/Bachelor_Project/specification-transformation$ python3
src/specification-transformation.py tests/test_program.c --from-property
termination --to-property reachability --algorithm T2RAlgorithm
```

Figure 10: Example of running specification-transformation.py to instrument a C program using our approach

Table 1: Usage of specification-transformation.py. Note: irrelevant options to our instrumentation are ignored.

Command Syntax	
`specification-transformation.py PROGRAM --algorithm ALGORITHM [--from-property FROM-PROPERTY] [--to-property TO-PROPERTY] [-h] [--output-dir OUTPUT_DIR]`	
Argument	**Description**
PROGRAM	The program to be transformed
Option	
–algorithm	The algorithm to be used for the transformation
–from-property	The property from which the program should be transformed
–to-property	The property to which the program should be transformed
-h, –help	Show help message and exit
–output-dir	The output directory

5.3 Implementation Flow

The implementation consists of two algorithms: `LocateLoopAndLiveVariableAlgorithm.java` and `t2R_algorithm.py`. The first algorithm is integrated in CPAchecker and located in `src/org/sosy_lab/cpachecker/core/algorithm`. It is used to parse C program as CFA, extracting the necessary loop information for later instrumentation. Specifically, for each while loop, we collect the line number of the loop head (consisting of the keyword **while**, the condition, and the opening bracket),[6] and the name and type of every variable used within the loop but declared outside of it. Figure 11 shows the loop information extracted from two previous programs, respectively. The second, and main, algorithm `t2R_algorithm.py` is integrated in the project specification-transformation and located in `src/algorithms`. Firstly, it checks for any unsupported data type or loop structure in the input C program. Secondly, it formats the program by clang-format so that every while loop strictly adheres to the structure depicted in Figure 12, with the loop head components placed on a single line and the closing bracket

[6]The C program received by CPAchecker is formatted beforehand so that it is guaranteed that the loop head components are on the same line.

on a separate line. Thirdly, it calls CPAchecker to extract the necessary loop information from the formatted program. Lastly, it instruments the formatted program with the help of the extracted loop information. Figure 13 illustrates the implementation flow described above.

4 {x=int}

(a) Loop information of the program in Figure 9a

5 {x=int}
9 {y=int}

(b) Loop information of the program in Figure 6a

Figure 11: Example for loop information

```
while (condition) {
  ...
}
```

Figure 12: Allowed while loop structure

5.4 Major Challenges

5.4.1 Extract Variable Names within a Loop

In the beginning, we intended to obtain variable names via CFA edges using the `getPartitionForEdge(CFAEdge edge)` method from the `VariableClassification` class. However, instead of solely retrieving variables from an edge, the method also returns variables upon which they depend. This can result in transformed loops appearing redundant, although it may not necessarily invalidate our reduction. So we adjusted the approach and decided to obtain variable names in an edge through its corresponding AST node [8].

5.4.2 Data Structure for Loop Instrumentation

To instrument a loop, we should insert three code snippets: one for variable declaration, another for variable value storage, and a third for assertion. These insertions are dependent on the concrete location of the loop head. Since during the inserting process it will definitely shift, we need to ensure constant reference to it. Therefore, we devised a new data structure. It is a

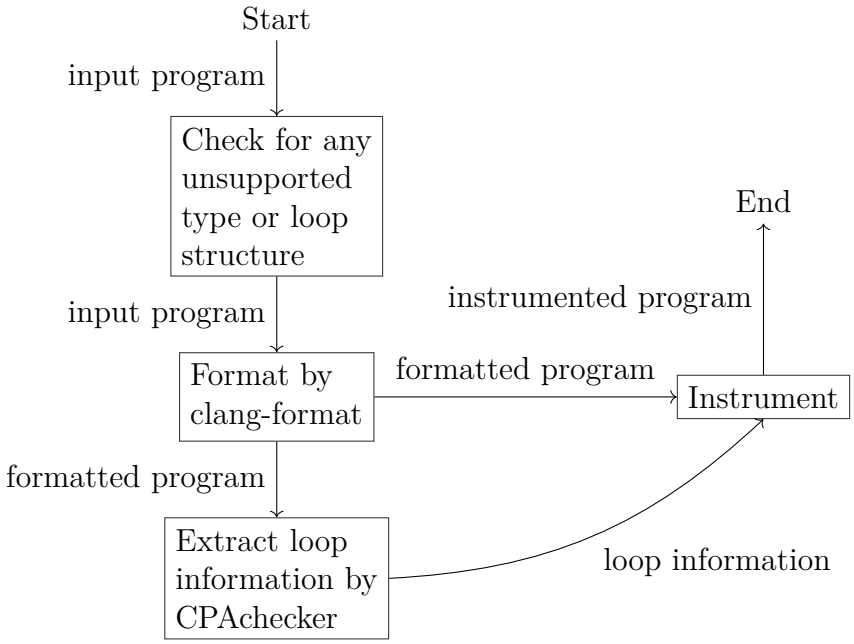


Figure 13: Implementation flow

list of pairs [(line number, code line **or** code snippet)]. We split the program to be instrumented into a list of code lines and store them into the structure properly. Furthermore, each inserted code snippet is assigned a universal line number. As a result, when inserting a code snippet into a loop, we can determine the location(index) of the loop head by referencing its original line number. Figure 14 depicts a generic example of inserting instrumentation code into a while loop.

6 Evaluation

To evaluate our approach, we conduct experiments based the two research questions stated in the introduction chapter.

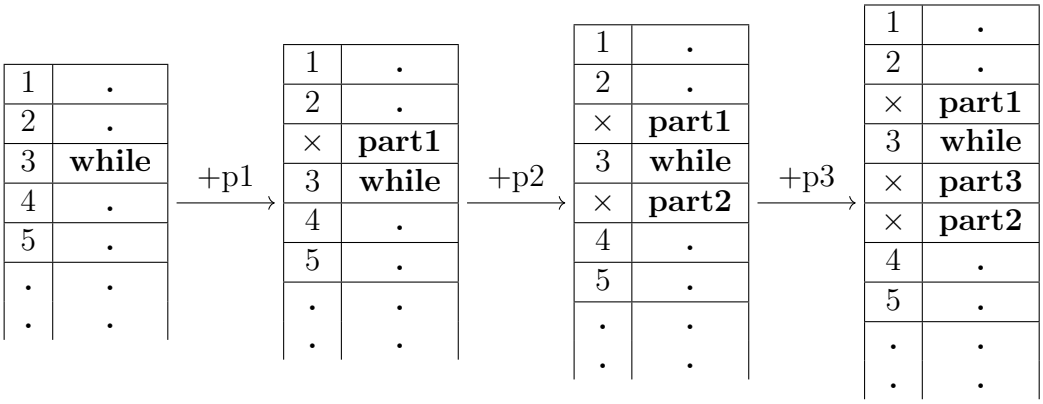


Figure 14: Generic example of inserting instrumentation code into a while loop. Note: The numbers in left column represent line numbers of program code. × denotes the universal line number for all inserted code snippets. The keyword **while** denotes a loop head. p1, p2 and p3 stand for part1, part2, and part3, respectively. part1 contains the declaration code, part2 the saving code, and part3 the assertion code.

6.1 Environmental Setup

6.1.1 Experimental Environment

- CPU: Intel Xeon E3-1230 v5 @ 3.40 GHz, cores: 8, frequency: 3800 MHz
- Operating system: Linux 5.15.0-92-generic
- RAM: 33467 MB

6.1.2 Limits on Experiments

- CPU time limit: 900 s
- CPU core limit: 4
- Memory limit: 15000 MB

6.1.3 Benchmark Tasks

To evaluate our approach, we selected four subcategories from the SV-Benchmark set:

- Termination-BitVectors.set
- Termination-MainControlFlow.set
- Termination-MainHeap.set
- Termination-Other.set

They contain a total of 4180 `.yml` files. However, only 2353 of these files include `termination.prp`, meaning they are the ones with expected result of termination analysis. Out of these 2353 files, only 591 have input files that meet the assumptions of T2R reduction. Among these 591 files, 282 have input files suitable for instrumentation, while the remaining 309 contain `unreach-call.prp`, indicating that their input files already have assertions which can influence our instrumentation. But some of these 309 files can be instrumented after commenting out the assertions within them. As a result, we are left with only 355 `.c` or `.i` files[7] which are suitable for instrumentation and termination analysis. For reachability analysis, we use their instrumented versions with correspondingly modified `.yml` files.[8]

6.1.4 Tools

For **reachability** analysis we used tools:

- CPAchecker 2.3[9] with approaches KID, BMC, IMC, PD, and PDL
- UAutomizer 0.2.4[10] with approach TAS
- CPV 0.4[11] with approach HWV

For **termination** analysis we used tools:

- CPAchecker 2.3 with approach LSB_C
- UAutomizer 0.2.4 with approach LSB_D

[7] `https://gitlab.com/sosy-lab/software/specification-transformation/-/tree/T2R_algorithm/files%20to%20be%20instrumented?ref_type=heads`

[8] `https://gitlab.com/sosy-lab/software/specification-transformation/-/tree/T2R_algorithm/instrumented%20files?ref_type=heads`

[9] `https://cpachecker.sosy-lab.org/download.php`

[10] `https://github.com/ultimate-pa/ultimate/releases`

[11] `https://zenodo.org/records/10203472`

We ran the benchmarks on the VerifierCloud via the webclient.[12] Firstly, we performed reachability analysis on those 355 instrumented tasks. Secondly, we conducted terminantion analysis on their original versions.

6.2 Experimental results

6.2.1 RQ1: Which tool for reachability analysis is the best for instrumented program?

Table 2 highlights that PDL and TAS achieved the highest numbers of correct results, with 298 (84%) and 294 (83%), respectively, while the other methods yielded only around 100 correct results. However, TAS has 17 (5%) wrong proofs and PDL has 28 (8%). Upon inspection, we discovered some interesting things. Let's start with the incorrect results of TAS. Firstly, their corresponding subject programs are instrumented as expected.[13] Secondly, all of these programs can trigger integer overflow behavior, which is undefined in C and has influenced their verification results in various ways. Therefore, we think that these 17 flawed tasks should not be included as part of our benchmark tasks. Next, we examine the 28 incorrect results of PDL,[14] of which 16 share the same subject programs as the incorrect results of TAS. As for the remaining 12, we found no obvious issues. Therefore, it is probable that the 12 incorrect results were caused by potential algorithmic bugs in the PDL approach. In summary, TAS has the most correct results, with 298 (84%). Although it has 17 (5%) wrong results, the root cause was not TAS itself. Thus, we conclude that, in terms of effectiveness, TAS outperforms all other approaches.

Figure 15 depicts the distribution of CPU time consumption for correct results of the approaches in the above table. We can see that HWV is the fastest on the first approximately 85 tasks but then takes significantly more time for each subsequent task. It suggests that HWV is particularly suitable for verifying easily solvable programs. However, in terms of overall perfor-

[12] https://gitlab.com/sosy-lab/doc/-/wikis/Benchmarking-for-Students-with-SVN-account

[13] https://gitlab.com/sosy-lab/software/specification-transformation/-/tree/T2R_algorithm/incorrect_results_of_trace_abstraction_in_UAutomizer?ref_type=heads

[14] https://gitlab.com/sosy-lab/software/specification-transformation/-/tree/T2R_algorithm/incorrect_results_of_predicate_analysis_with_linear_arithmetics_in_CPAchecker?ref_type=heads

Table 2: Experimental results for reachability analysis

Verifier	CPAchecker					UAutomizer	CPV
Approach	KID	BMC	IMC	PD	PDL	TAS	HWV
Correct results	119	123	96	99	**294**	**298**	112
correct proofs	50	53	37	34	235	227	42
correct alarms	69	70	59	65	59	71	70
Incorrect results	0	0	0	0	28	17	0
wrong proofs	0	0	0	0	**28**	**17**	0
wrong alarms	0	0	0	0	0	0	0
Timeouts	233	229	180	250	5	39	229
Other inconclusive	3	3	79	6	28	1	14

mance on the benchmark tasks, PDL is the best, successfully completing more than 280 tasks, with each task taking around 7 s. The second best, TAS, also completed more than 280 tasks, but with each task taking roughly 17 s.

In conclusion, considering both effectiveness and performance, we believe that UAutomizer with approach TAS is the best tool for reachability analysis of instrumented program, because obtaining 4 more correct results and 10 fewer errors is more important than an average performance difference of approximately 10 s per task.

6.2.2 RQ2: Is the combination of T2R tool and the best tool for reachability analysis of instrumented program more effective and efficient than current termination analysis techniques?

Now, we compare the combination of T2R tool and TAS with current termination analysis techniques, as shown in Table 3. We can see that T2R+TAS has the highest number of correct results with 298 (32%), which is 57 (16%) more than LSB_C and 36 (10%) more than LSB_U. Moreover, based on the discusstion about the first research question, we can ignore the 17 wrong proofs of T2R+TAS. Therefore, T2R+TAS demonstrates better effectiveness over the other two termination analysis techniques.

For efficiency, let's look at Figure 16. It is evident that, on the first approximately 75 tasks, LSB_C is the fastest. However, considering overall performance, T2R+TAS outperforms both LSB_C and LSB_D.

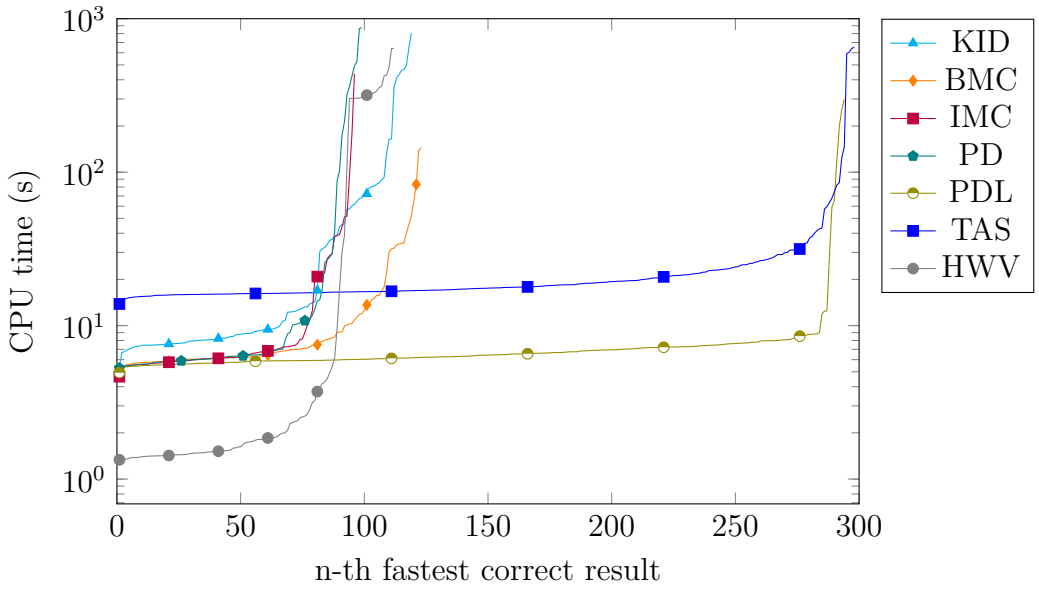


Figure 15: Quantile plot for CPU time consumption of correct results in reachability analysis

Table 3: Experimental results for termination analysis

Verifier	CPAchecker	UAutomizer	
Approach	LSB_C	LSB_U	T2R+TAS
Correct results	241	262	**298**
correct proofs	163	199	227
correct alarms	78	63	71
Incorrect results	7	0	17
wrong proofs	1	0	**17**
wrong alarms	6	0	0
Timeouts	38	6	39
Other inconclusive	69	87	1

6.3 Threats to Validity

6.3.1 External Validity

Our benchmark tasks include only programs with certain data types and the while loop as loop structure, so some verifiers may outperform others solely

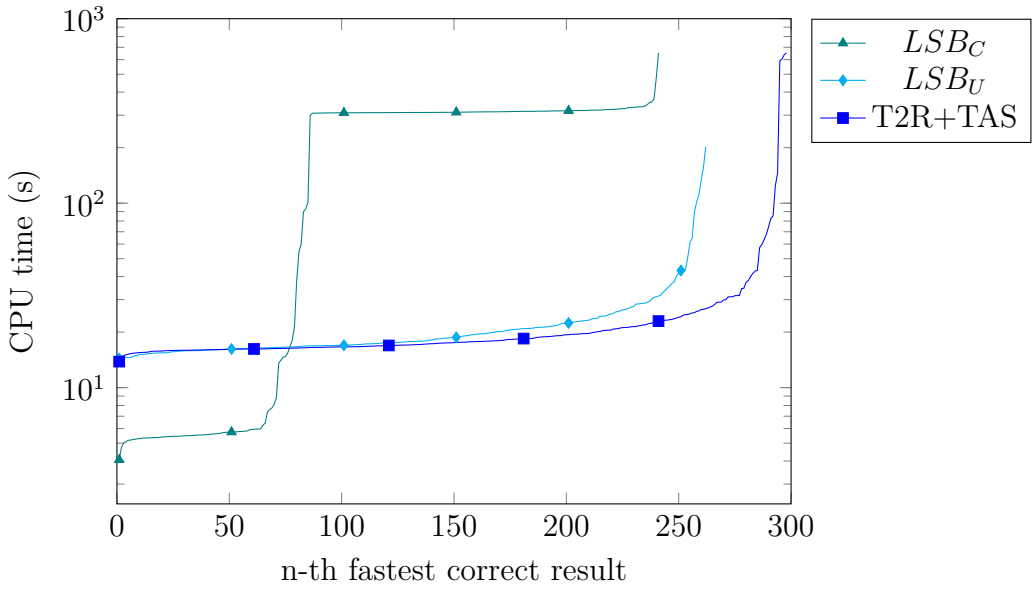


Figure 16: Quantile plot for CPU time consumption of correct results in termination analysis

on these tasks, whether in termination or reachability analysis. Therefore, our answers to the two research questions may change, if we include more programs with additional data types or loop structures, given that our T2R tool can instrument them.

6.3.2 Internal Validity

The implementation of the T2R tool may contain bugs, and thus, instrument some programs wrongly. We would expect that such a bug could also lead to incorrect results in the reachability analysis of wrongly instrumented programs. However, since we inspected the incorrect results in reachability analysis and found that the instrumentations of the programs were correct, we assume this is unlikely.

7 Future Work

7.1 Practical: Support More Data Types and Loop Structures

Our current implementation does not support while loop without brackets, which can have at most one statement in its loop body. So we could fix this in the future. Also, we could extend our implementation by supporting additional data types such as struct or array, as well as additional loop structures like for loop or recursion.

7.2 Theoretical: Prove Termination from Witness of Reachability Analysis

The reachability analysis of instrumented programs sometimes produce incorrect results, which can be a false alarm or a wrong proof. [2] To increase the reliability of verification results, we accompany their answers by witnesses, which can be validated by witness validators. However, a witness obtained from the reachability analysis of a instrumented program does not provide any information about the termination of the original program. Therefore, we need to devise a method for proving the termination of a program based on the witness obtained from the reachability analysis of its instrumented version.

8 Conclusion

The main goal of this thesis was to devise a tool for reducing the termination of C programs to a reachability-safety problem. This tool allows us to leverage various reachability algorithms, which are more fine-tuned than termination analysis techniques. Our implementation strategy begins with extracting the necessary loop information from a C program using CPAchecker, followed by instrumenting each loop within it using state recording translation. To evaluate our approach, we first selected the best tool for reachability analysis of instrumented program by testing various configurations of CPAchecker, UAutomizer, and CPV on a set of 355 instrumented tasks. Subsequently, we compared the combination of the T2R tool and the selected tool against the termination analysis techniques of CPAchecker and UAutomizer on the

original versions of those 355 instrumented tasks. The results demonstrated that UAutomizer with approach TAS is the most effective tool for reachability analysis of instrumented program, although its overal performance is not the best. However, the combination of it and our T2R tool is noticeably more effective and efficient than the tested termination analysis techniques. Therefore, we believe that our approach is worth extending in the future so that it could be more comprehensive and implement more programs.

List of Figures

List of Tables

References

[1] 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.

[2] D. Beyer and J. Strejcek. Case study on verification-witness validators: Where we are and where we go. In G. Singh and C. Urban, editors, *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings*, volume 13790 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2022.

[3] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In R. Cleaveland and H. Garavel, editors, *7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems, FMICS 2002, ICALP 2002 Satellite Workshop, Málaga, Spain, July 12-13, 2002*, Electronic Notes in Theoretical Computer Science, pages 160–177. Elsevier, 2002.

[4] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 491–504. Springer, 2005.

[5] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. *Handbook of Model Checking*. Springer, 2018.

[6] M. Colón and H. Sipma. Synthesis of linear ranking functions. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2001.

[7] B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms*

for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, volume 7795 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2013.

[8] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Katey Birtcher, 2023.

[9] A. Goel and K. A. Sakallah. AVR: abstractly verifying reachability. In A. Biere and D. Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I*, volume 12078 of *Lecture Notes in Computer Science*, pages 413–422. Springer, 2020.

[10] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. Ultimate automizer with smtinterpol. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 641–643, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[11] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[12] X. Xie, B. Chen, L. Zou, S. Lin, Y. Liu, and X. Li. Loopster: static loop termination analysis. In E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 84–94. ACM, 2017.