UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Software Engineering Curriculum

Karoliine Holter

Adapting an Alarm Repositioning Algorithm to Data Races

Master's Thesis (30 ECTS)

Supervisors: Vesal Vojdani, PhD
Simmo Saan, MSc

Tartu 2023

Adapting an Alarm Repositioning Algorithm to Data Races

Abstract:

This master's thesis addresses the challenge of enhancing the usability of sound static analyzers, specifically focusing on the state-of-the-art data race verifier Goblint. The aim is to soundly post-process the warnings generated by Goblint to make them more understandable for developers, thereby increasing the adoption of sound analyzers in practice. The thesis adapts and extends the warning repositioning algorithm of Muske et al. for data race warnings in multi-threaded C programs. Contributions include identifying and implementing a potential solution within the Goblint analyzer, extending the method for data races, and evaluating and analyzing the adapted algorithm in terms of the reduced distance between possible causes and warnings, as well as the impact on the quality of data race warnings.

Keywords:
Static analysis, static analysis alarms, data-flow analysis, alarms repositioning, Goblint

CERCS:
P170 Computer science, numerical analysis, systems, control

Hoiatuste ümberpaigutamise algoritmi kohandamine andmejooksudele

Lühikokkuvõte:
See magistritöö käsitleb staatiliste analüsaatorite kasutajasõbralikuks muutmise väljakutset, keskendudes paralleelprogrammides andmejooksude tuvastamise tööriistale Goblint. Eesmärk on Goblinti loodud hoiatusi usaldusväärselt järeltöödelda, et hoiatused arendajatele arusaadavamaks muuta ning seeläbi tarkvara verifitseerijate praktikas kasutuselevõttu suurendada. Töös kohandatakse ning täiustatakse Muske jt. poolt välja töötatud hoiatuste ümberpaigutamise algoritmi nii, et see töötaks ka mitmelõimelistes programmides leitud võimalike andmejooksude hoiatuste peal. Töö panusteks on potentsiaalse algoritmi tuvastamine, selle täiustamine ning teostamine Goblinti tööriistas. Saadud algoritmi praktilisust hinnatakse ja analüüsitakse hoiatuse ning hoiatuse võimaliku põhjuse vahel vähenenud koodiridade arvu järgi. Lisaks hinnatakse algoritmi mõju andmejooksude hoiatuste kvaliteedile.

Võtmesõnad:
staatiline analüüs, staatilise analüüsi hoiatused, andmevooanalüüs, hoiatuste ümberpaigutamine, Goblint

CERCS:
P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

Program correctness is a vital aspect of software development, and static analysis tools can effectively detect bugs in code early on, reducing the negative impact of programming errors [21, 24]. Sound analyzers, in particular, can provide strong guarantees of correctness, which is essential for safety-critical and security-sensitive applications where vulnerabilities can be exploited [6].

Although having great potential to help verify programs automatically, sound analyzers are often more challenging to use and produce many false alarms. The significant amount of false alarms [3, 8, 9, 19] and the complexity of the warning messages [32] are some of the main reasons why many static analyzers have not been adapted in practice. The proportion of false alarms produced by static analyzers is reported to range as high as 91%, and moreover, combined with warnings that are difficult to understand, create overhead of alarm inspection and limit their adoption in the industry [18].

Given that verification is undecidable in general, improving analysis precision is a complex and potentially infinite task. If an analyzer fails to verify, it is difficult to know whether there is an actual bug in the code or if the analysis is not precise enough to prove its absence. Therefore, it is crucial to not only improve the precision of sound analyzers but also enhance their usability without sacrificing soundness.

This thesis focuses on the challenge of enhancing the usability of sound analyzers while maintaining soundness. More specifically, the goal is to soundly post-process the warnings of the state-of-the-art data race verifier Goblint [16], a sound static analyzer for multi-threaded C programs that determines the absence of faults using abstract interpretation [11]. Despite being the most successful verifier in the SV-COMP race detection category [4, 5], it generates many false alarms on real-world programs. However by the information in the warnings alone, it is difficult to distinguish if the warning is a true or a false positive.

Multi-threaded programs are notoriously difficult to reason about, as program behaviour depends on thread interactions. Making data race warnings more understandable is an open problem, so in order to lay the groundwork for this challenge, this thesis will explore if state-of-the-art techniques for single-threaded programs can be adapted to the multi-threaded case. After exploring various methods, the warning repositioning algorithm of Muske et al. [26] was chosen for its potential to address both soundness and usability concerns. The original algorithm was proposed for array-out-of-bounds and zero division alarms, but the goal is to analyse if this algorithm could also be used on the data race warnings.

Analyzer warnings are normally generated at the program point where a failure in the program functions, i.e. a run-time error or a data race, would happen. However, this is

generally not the place in the code where the fault causing that failure is. Post-processing warnings can help better understand their causes, as the warnings of the possible failures can be repositioned to and complemented with information about the program points where the possible faults in the code can be. Moreover, in case of a false alarm, adding information about where things could go wrong according to the analyzer can help save time by reducing code traversals during alarm inspection. Overall, the thesis makes the following contributions:

- **Identifying a potential solution.** With the open-ended goal of improving the quality of data race warnings, a method for repositioning analysis warnings is identified and adapted for implementation within the Goblint analyzer.
- **Extending the method for data races.** The repositioning algorithm, which originally handles single-threaded data-flow properties, is extended to reposition data race warnings in multi-threaded programs.
- **Evaluation and analysis**. The adapted algorithm is evaluated in terms of reduced distance between cause and warnings. The impact of the algorithm on the quality of data race warnings is manually analyzed on select benchmarks, yielding insight for future work on producing understandable warnings.

The thesis is structured as follows: Section 2 provides an overview of the Goblint analyzer, its goals, and where this thesis fits in. Section 3 surveys the relevant literature and identifies a potential solution, which is formalized in Section 4. Section 5 is dedicated to formalizing the extension of the chosen method for data races. The implementation of the analyses formalized in Sections 4 and 5 is described in Section 6 and the data race warning repositioning is evaluated in Section 7. The results are discussed in Section 8, and the thesis concludes with Section 9.

2 Background

Several static analysis tools have found their way into developers' workflow, but the analyses that have been integrated are relatively simple [32]. The complex analyses have yet to find a way to achieve widespread usage in the development processes.

Goblint [16] is a sound static analyzer for multi-threaded C programs that determines the absence of faults using abstract interpretation [11]. In addition, it is the best data race verifier, according to the results of the Competition on Software Verification (SV-COMP) 2023 [4, 5], which represent the state-of-the-art in software verification.

For Goblint, the focus is set on truthfulness, with an aim to derive and communicate truthful explanations that developers can comprehend. To achieve that, all of the following must be addressed [35]:

1. **Theoretical Soundness.** By proving the correctness of the analysis on paper, a sound analysis is obtained. This relies on the mathematical theory of abstract interpretation [11].
2. **Transparent Evaluation.** The sound analysis specification then needs to be translated into a trustworthy analyzer that works on real-world programs. The analysis will be implemented and evaluated with complete transparency to ensure applicability. Machine-checkable witnesses are generated to enable assessment and confirmation by other analyzers to establish trust in the implementation.
3. **Tangible Explainability.** While witnesses allow other analyzers to validate the results, they are not suitable for human inspection. Therefore, work on usability and interactivity with the goal of providing accessible, human-readable explanations of analysis results must be done.

The research behind Goblint so far has mainly focused on the first two: proving the correctness of the analyses [36] and transparently implementing and evaluating the analyses [5, 30]. This thesis aims to tackle the third point as the first step in the direction of discovering how to improve the usability and interactivity of Goblint in the sense of warning reduction and explainability.

A high-level illustration of Goblint's architecture is given on Fig. 1. Goblint takes a C program as input and compiles it into a simplified subset of C using CIL (C Intermediate Language) [10]. Then, a control-flow graph (CFG) is constructed from CIL and used for the analysis. The analyzer consists of two main parts: analyses and the framework. Different analyses are implemented in Goblint, some of many being value, mutex, and thread ID analysis. These analyses use different domains ranging from simpler boolean, set, and interval domains to more specific such as congruence domains. The combined analyses and the control-flow graphs of the functions yield a constraint system [2], which

is solved using a local generic solver [31, 33].

The recent efforts to analyze larger programs [14] have led to the discovery of Goblint producing many similar warnings, out of which numerous can possibly be false alarms. Therefore, a warning post-processing algorithm is sought to help reduce the number of warnings. Currently, the only post-processing method implemented is to produce correctness witnesses [29] for the validators to verify the proofs. Still, these are not meant to be human-readable, and are only produced for the verified programs. For the programs that were not verified and thus produced warnings, there are no post-processing methods yet applied in Goblint. The following section will give an overview of the related work on false positive reduction with a focus on post-processing methods. To satisfy the requirement of truthfulness, the post-processing method must be sound to preserve the soundness of the analysis.

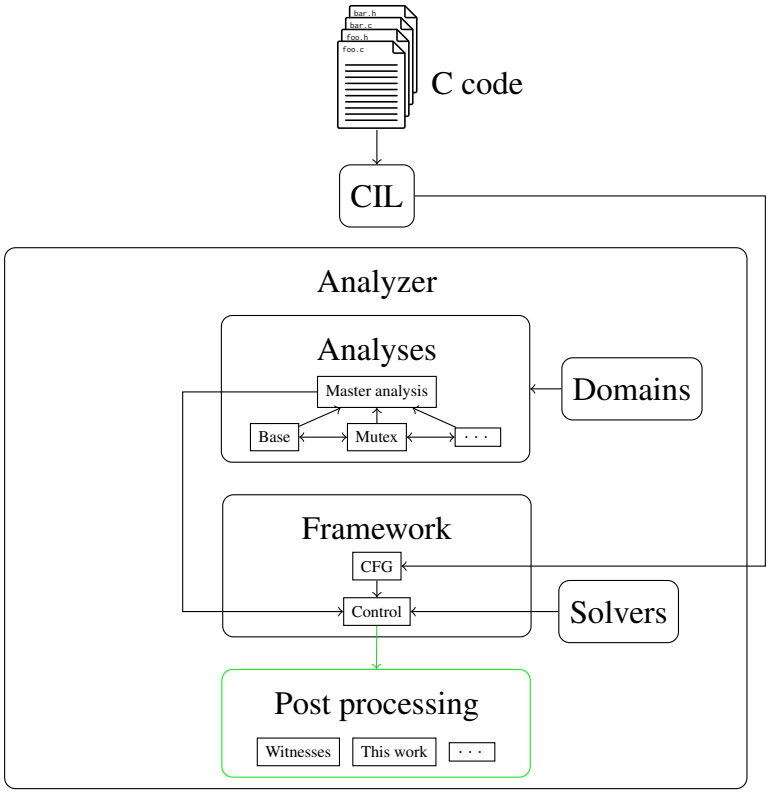


Figure 1. Goblint high level architecture.

3 Literature review

This section aims to give a brief overview of the different methods used to reduce false alarms in sound static analyses, and the methods for post-processing static analysis alarms with the goal of improving their usability, as well as how the method for post-processing warnings in Goblint was chosen based on our goal and the methods available.

3.1 False positives reduction in sound static analysis

The most evident way to reduce false alarms is to improve the precision of the analyzer. One such approach has been to design increasingly case-specific analyses and abstract domains for each programming feature or idiom. But for the vastly case-specific analyses, the balance between the cost of one analysis design, and the number of false alarms it reduces, is unreasonable.

Instead, one may prefer having an abstract domain that is sufficiently expressive to reason about all programs. However, such domains are naturally computationally expensive and will require precision to be adjusted. As the domains are made less expressive and less precise to overcome the performance issues, several manual and automatic precision refinement methods have been proposed to again overcome the imprecision problems.

An example of an automatic precision refinement method is demand-driven value refinement [34], which uses relational information to soundly and independently regain precision in critical locations during the data-flow analysis. The manual approaches, on the contrary, let the user help guide the analysis with their external knowledge, using code annotations and tool's configuration parameters. Such user-guided analysis approach is used by Astrée [12] and Frama-C [7], but also supported by Goblint [15, 31].

The Astrée approach is of interest because, similarly to Goblint, Astrée is an abstract interpretation-based static analyzer for C, and AstréeA [23], an extension of Astrée, can also analyze concurrent C programs. The tool's parameters can be tuned to use specific analyses in specific program parts. For example, it can be configured to run separate analyses for separate values of a variable instead of one for all its values, improving the precision so that a false alarm caused by this imprecision is not raised [12].

Using parameters and annotations gives the user more control over the analyses, leading them to the desired results – a verified program with zero false alarms. However, the usage of directives and parameters expects manual and time-consuming configurations by an expert on the particular analysis, making such approaches unavailable for most software engineers. Although Astrée has shown great success in reducing the false alarms to zero [23], it only performs on specific programs and is not scalable on other software.

As verification problems are undecidable, improving the precision of the analyzers is

possibly an infinite process that will, in general, not result in getting rid of all the false alarms. Furthermore, even for the precision refinement, it would help if the warnings assist the user in understanding why the analysis failed. Therefore, hand-in-hand with improving precision, the presentation of false alarms must be improved to make it as easy as possible for the developers to distinguish between true and false alarms.

3.2 Approaches for post-processing static analysis alarms

Several methods have been developed for post-processing static analysis alarms to sort, prioritize and cluster the warnings to make them more understandable for the analysis users. Muske and Serebrenik [25] have studied the different approaches for post-processing static analysis alarms and have come up with the following categorisation:

1. **Clustering.** Alarms are clustered into several groups based on similarity or correlations among them.
2. **Ranking.** Alarms are ranked using various characteristics of the alarms, the source code, history of bug/alarm fixes, code-commit history, and so on.
3. **Pruning.** Alarms are classified into two classes, actionable and non-actionable, and the non-actionable alarms are pruned.
4. **Automated false positives elimination (AFPE).** Alarms are processed further using more precise techniques like model checking and symbolic execution to identify and eliminate false positives from the alarms automatically.
5. **Combination of static and dynamic analyses.** Alarms are processed using dynamic analysis to generate test cases that validate true errors.
6. **Simplification of manual inspection.** Manual inspection of alarms is simplified by enriching alarms with additional information, providing tool support, etc.

The techniques to implement these post-processing methods differ greatly and do not always match those used in the static analysis tools for generating the alarms. Our goal is to remain sound even after post-processing the alarms, and for an analyzer to stay sound, the warnings cannot be removed using just any heuristics as removing an alarm in the absence of the proof might lead to stating that an incorrect program is actually correct. Therefore, only those post-processing methods that preserve the soundness of the tool can be considered.

3.3 Sound alarms post-processing

The guidelines proposed by Muske and Serebrenik [25] state that if the objective of the analysis is to verify, then the alarms can be subjected to post-processing techniques such as sound clustering, ranking, and simplification of manual inspection because using other approaches can result in false negatives.

We have excluded any ranking methods, which mostly rank the alarms based on some heuristics or rules. That is because statistical methods usually do not guarantee soundness and we do not want to misjudge any of the produced alarms. Furthermore, for the main focus of this thesis, the race alarms, there is already a sound ranking method in place in Goblint [36].

Automated false positives elimination methods largely fall into the same category as the Astrée approach, as they use precision refinement, which requires expert knowledge from the user of the analyzer and thus does not fulfil the goal of this thesis. For the same reason, we have also excluded combining the static analysis with dynamic analyses because the goal is not to validate true errors but to find a more general method to better explain warnings produced by the analyzer, both true and false positives.

The approaches that best fit the goal of this thesis are clustering and simplification of manual inspection. There exist both sound and unsound clustering methods, out of which we will consider only sound ones. Sound clustering techniques can be useful for reducing the number of alarms reported by static analysis tools and is appropriate for both code-proving and bug-finding tools [25], and thus, satisfy our goal.

Out of the seven proposed sound clustering methods, we have chosen the repositioning algorithm proposed by Muske and Tukram [26], as they have taken into account the limitations of several of the other six methods and improved upon them. The algorithm promises to help reduce the number of warnings as well as report the alarms closer to their causes, the latter making it also fall under the simplification of manual inspection category.

4 Repositioning of alarms

To cluster the warnings generated by Goblint, the technique introduced by Muske and Tukram [26] was used. The method's goal is to soundly reduce the number of warnings while moving the alarms closer to their cause. Although Muske et al. have evaluated [27] that the method has limitations for reducing the number of false alarms, it has the potential to help reduce the number of repetitive warnings, which is one of the usability issues of Goblint. In addition, the repositioning of warnings is the first step in the effort to locate and reposition the warnings to their possible causes and thus also improve the explainability of the warnings.

The method is a two-step algorithm consisting of intermediate repositioning and repositioning refinement. On a very high level, the alarms are first hoisted up in the program to find the possible cause of the alarm and then sank back down to find alarms that could be merged to reduce the number of warnings. The algorithm uses an analysis similar to anticipable expressions analysis (also known as very busy expressions analysis) [20] as backward analysis and an analysis similar to available expressions analysis [20] for forward analysis. But instead of expressions, the property tracked in this data-flow analysis is conditions that produce the alarms. Let us denote a warning message generated in program point o as m_o and define anticipable and available alarm conditions.

Definition 4.1 (Anticipable alarm conditions)**.** A condition c is anticipable at a program point p if every path from p to the exit of the program contains the warning message m_o caused by condition c, and the program point o is not preceded by any assignments to any operand in c on any path from p to o.

Definition 4.2 (Available alarm conditions)**.** A condition c is available at a program point p if every path from the program entry to p contains the warning message m_o caused by condition c, the program point o is not followed by any assignments to any operand in c on any path from o to p.

The alarms in the original article are referred to as warnings in Goblint terminology. Therefore, alarm and warning (message) in this context are used interchangably.

Intermediate Repositioning. During the first step, backward data-flow analysis is used to hoist the original warnings to the highest program point where an alarm condition is *anticipable*, but the same condition is no longer anticipable at any program point just before. Meaning that alarms are moved up in the program until an assignment to an operand in c is found, or the analysis reaches the program start. Hoisting the alarms moves the warning closer to its cause, but this step alone can increase the number of warnings, as there might be several branches in the program where the condition is met.

Repositioning Refinement. The second step refines the alarms hoisted at the first step by sinking the alarms to determine if they can be merged. During the forward data-flow analysis, the *available* conditions are computed. This analysis also identifies the highest program points where moving the warnings further down would not yield any more merges before reaching the lowest program points where condition c is still available. These highest program points are computed by finding the lowest meet-program points of the *available* conditions. Finding these ensures that for each alarm condition c available in a program point p, there is only one warning with the same alarm condition on all of the paths from program entry to program point p. This step reduces the number of warnings but moves the warnings further away from the causes.

The data-flow analyses traverse a control-flow graph, defined as follows:

Definition 4.3 (Control-flow graph)**.** "A control-flow graph (CFG) [1] of a program is a directed graph $\langle N, E \rangle$, where N is a set of nodes representing the program statements (like assignments and controlling conditions); and E is a set of edges where an edge $(n_1, n_2) \in E$ represents a possible flow of program control from $n_1 \in N$ to $n_2 \in N$ without any intervening node" [26].

In [26], the states right before and after executing a statement correspond to the entry and exit of each node, respectively. These states are denoted as *in* and *out* and placed right before and after each node. It is assumed that an exit from a node does not directly correspond to an entry of another node and vice versa, although in some cases, they might represent the state of the exact same program point.

The repositioning analysis by Muske and Tukram was proposed to be general. The concrete examples discussed and the evaluation of relocating alarms was done on two commonly checked runtime error categories: array index out of bounds (AIOB) and division by zero [26]. In this work, the methodology is extended to work with data race warnings which is the main focus of Goblint.

The analyses are described in detail in Sections 4.1 and 4.2. Both analyses are formalized without the weakest precondition and strongest postconditions calculations and at an intraprocedural level. The extension of the algorithm to data race warnings is described in Section 5.

4.1 Backward data-flow analysis for hoisting alarm conditions

The original warnings to be repositioned are generated by an analysis which checks whether some condition holds in a program point. If the condition does not hold, the analysis outputs a warning message. Using the original warning messages and the CFG, the backward data-flow analysis computes the highest program points where the warning message is anticipable.

Let us denote the set of conditions as C and the set of warning messages as M. The backward data-flow analysis computes subsets of condition and message pairs $S \subseteq L_b = C \times M$ in each program point $p \in P$. The lattice of those subsets is given as $\langle B, \sqcap_B \rangle$, where B is the powerset of L_b. For the subsets, the following functions are defined:

$$\text{conds}(S) = \left\{ c \;\middle|\; \langle c, m \rangle \in S \right\} \tag{1}$$

$$\text{filter}(c, S) = \left\{ \langle c', m \rangle \;\middle|\; \langle c', m \rangle \in S \wedge c = c' \right\} \tag{2}$$

Equation (1) computes the set of conditions from a set of condition and message pairs and Eq. (2) the set that contains only those condition and message pairs that match the condition given as the function argument. The meet $\sqcap_B$ of the lattice can then be defined so that given $X, Y \in B$

$$X \sqcap_B Y = \bigcup_{c \,\in\, (\text{conds}(X) \,\cap\, \text{conds}(Y))} \text{filter}(c, X) \cup \text{filter}(c, Y) \tag{3}$$

The result of the meet is a set of those conditions and message pairs for which the condition is present in both of the given sets. Let there be $n, s \in N; c \in C; m \in M; X \in B$, then the data-flow equations of the backward analysis are:

$$\text{Ant}_n^{out} = \begin{cases} \emptyset & n \text{ is } End \text{ node} \\ \bigsqcap_{B} \limits_{s \in \text{succ}(n)} \text{Ant}_s^{in} & \text{otherwise} \end{cases} \tag{4}$$

$$\text{Ant}_n^{in} = \text{Gen}_n \cup (\text{Ant}_n^{out} \setminus \text{Kill}_n(\text{Ant}_n^{out})) \tag{5}$$

$$\text{Gen}_n = \left\{ \langle c, m \rangle \;\middle|\; m \text{ was generated in } n \text{ by } c \right\} \tag{6}$$

$$\text{Kill}_n(X) = \left\{ \langle c, m \rangle \in X \;\middle|\; \begin{array}{c} n \text{ has an assignment} \\ \text{to a variable in } c \end{array} \right\} \tag{7}$$

Equations (4) and (5) define the subsets of condition and message pairs computed in each program point. Gen_n (Eq. (6)) contains a set of upwards exposed warning messages generated in node n, and Kill_n (Eq. (7)) contains a set of those condition and message pairs where a variable in the condition is changed in node n.

Finally, the program points where the warnings are hoisted are calculated. In each path reaching the original warning location, the warning is hoisted to the highest hoisting point. The highest hoisting point is the highest program point where the conditions are anticipable, i.e. where in any program point before n, the condition is no longer anticipable. There are two cases for finding the highest hoisting points, formalized by the following equations:

$$\mathrm{Hoist}_n^{in} = \left\{ c \;\middle|\; \begin{array}{c} c \in (\mathrm{conds}(\mathrm{Ant}_n^{in}) \setminus \bigcap_{p \in \mathrm{pred}(n)} \mathrm{conds}(\mathrm{Ant}_p^{out})) \\ \wedge\ \mathrm{alwaysTrue}^{in}(c, in_n) \neq true \end{array} \right\} \tag{8}$$

$$\mathrm{Hoist}_n^{out} = \left\{ c \;\middle|\; \begin{array}{c} c \in \mathrm{conds}(\mathrm{Kill}_n(\mathrm{Ant}_n^{out})) \\ \wedge\ \mathrm{alwaysTrue}^{out}(c, out_n) \neq true \end{array} \right\} \tag{9}$$

$$\mathrm{alwaysTrue}^{in}(c, p) = \begin{cases} true & \text{if condition } c \text{ always holds at } p \\ false & \text{otherwise} \end{cases} \tag{10}$$

$$\mathrm{alwaysTrue}^{out}(c, p) = \begin{cases} true & \text{if after evaluating the transfer function} \\ & \text{from } p \text{ the condition } c \text{ always holds} \\ false & \text{otherwise} \end{cases} \tag{11}$$

First, a condition c is anticipable at out_n and not anticipable at in_n, when the anticipability is killed because the value of a variable in c is changed by a transfer function from node n. These highest hoisting points are identified by Eq. (9). The second case occurs when c is anticipable through some of the branches coming out of a branching node p, but not anticipable in another branch coming from the same node. Therefore, if condition c is anticipable at in_n at the beginning of a branch but not anticipable at some predecessor out_p, the highest hoisting point in that path is at in_n. The highest hoisting points for such paths are identified by Eq. (8).

Some highest hoisting points can be redundant. An example of redundancy can be seen in Fig. 2 where the cause of the warning message is in one branch, but in the other branch, the condition would always hold. Meaning that if there is no fault in a branch and on any of the paths reaching that branch, then the warning can safely be removed. Such redundant highest hoisting points are discarded by the helper functions in Eqs. (10) and (11). After all the highest hoisting points are located, the hoisted warnings will be complemented with the related program point location as the possible cause.

```
1  void foo() {
2      int arr[3]={1,2,3}, x=0, r1=rand(), r2=rand();
3      int i = 1;
4
5      if(r1) {
6          i = r2;     // highest hoisting point
7      } else {
8          x = 1;      // highest hoisting point (discarded by alwaysTrue)
9      }
10
11     arr[i] = 1;     // original warning
12 }
```

Figure 2. An example of a redundant highest hoisting point that can be discarded.

```
1  void bar() {
2      int arr[3]={1,2,3}, i=0, r1=rand(), r2=rand(), r3=rand();
3
4      if(r1) {
5          i = r2; // highest hoisting point
6      } else {
7          i = r3; // highest hoisting point
8      }
9
10     arr[i] = 1; // original warning
11 }
```

Figure 3. Simple code example where hoisting warnings increases the number of warnings.

In some cases, such as a similar simple branching, the hoisting can also increase the number of warnings. Let us look at the example in Fig. 3. There is an access to an array index with a variable whose value is unknown on line 10, originally generating only one warning. But there are assignments to that variable in both of the branches of an if statement (lines 5 and 7). While hoisting the original warning up, the anticipability is killed by the assignment in both of the branches and as by Eq. (9), the program points on lines 5 and 7 are identified as the highest hoisting points.

Although successfully locating the two actual causes for the generated warning, in this case, the overall number of warnings increased. Therefore Muske and Tukram [26] have proposed that after hoisting the warnings up, they be sunk down for the purpose of merging.

4.2 Forward data-flow analysis for merging alarm conditions

The hoisted warning messages will be sunk down in order to merge those that were originally the same but have gone up on different paths to different highest hoisting points. The forward data-flow analysis computes program points where the conditions are available and in every program point p where the available alarm conditions meet, the warning messages that have the same condition are merged. Let there be $n, p \in N; c \in C; m \in M; X \in B$, the data-flow equations of the forward analysis are based on the equations for computing available alarm conditions with related original alarms [26] and are defined as follows:

$$\mathrm{Av}_n^{in} = \begin{cases} \emptyset & n \text{ is } Start \text{ node} \\ \bigsqcap_{\substack{B \\ p \in \mathrm{pred}(n)}} \mathrm{Av}_p^{out} & \text{otherwise} \end{cases} \tag{12}$$

$$\mathrm{Av}_n^{out} = \mathrm{Gen}_n^{out} \cup (\mathrm{Av}_n^{in'} \setminus (\mathrm{Kill}_n(\mathrm{Av}_n^{in'}))) \tag{13}$$

$$\mathrm{Av}_n^{in'} = \mathrm{Av}_n^{in} \cup \mathrm{Gen}_n^{in} \tag{14}$$

$$\mathrm{Gen}_n^{out} = \left\{ \langle c, m \rangle \in \mathrm{filter}(c, \mathrm{Ant}_n^{out}) \;\middle|\; c \in \mathrm{Hoist}_n^{out} \right\} \tag{15}$$

$$\mathrm{Gen}_n^{in} = \left\{ \langle c, m \rangle \in \mathrm{filter}(c, \mathrm{Ant}_n^{in}) \;\middle|\; c \in \mathrm{Hoist}_n^{in} \right\} \tag{16}$$

$$\mathrm{Kill}_n(X) = \left\{ \langle c, m \rangle \in X \;\middle|\; \begin{array}{c} n \text{ has an assignment} \\ \text{to a variable in } c \end{array} \right\} \tag{17}$$

Equations (15) and (16) give a set of downward exposed warning messages that have been hoisted to node n, and for which the available alarm conditions are calculated. Equations (12) to (14) define the subsets of available alarm condition and message pairs computed by the forward data-flow analysis in each program point. The final repositioning with traceability links to the highest hoisting points is acquired with the following equations:

$$\mathrm{Sink}_n^{in} = \mathrm{conds}(\mathrm{Kill}_n(\mathrm{Av}_n^{in'})) \tag{18}$$

$$\mathrm{Sink}_n^{out} = \mathrm{conds}(\mathrm{Av}_n^{out}) \setminus \bigcap_{s \in \mathrm{succ}(n)} \mathrm{conds}(\mathrm{Av}_s^{in}) \tag{19}$$

The algorithm makes use of each condition from the entry of the End node for repositioning as a unique circumstance because some conditions may reach the program end without being processed by any of the Eqs. (18) and (19) for any program point [26]:

$$\mathrm{Sink}_{End}^{in} = \mathrm{conds}(\mathrm{Av}_{End}^{in'}) \tag{20}$$

5 Repositioning data race warnings

Unlike AIOB analysis, where it is sufficient to check whether the value of the expression used to access the array is within the bounds of the array's length, the condition for detecting whether a data race occurs is not as clear. This is because, unlike the bugs that can be detected using assertions, detecting data races requires a more complicated analysis, such as analyzing information about threading, synchronization events like locking or happens-before, or even making the analysis memory-model–sensitive. Furthermore, data race warnings come in pairs, as a data race can only happen between two or more accesses. Therefore it must be ensured that the repositioning preserves the pairs.

One possible condition for data race warnings repositioning is a pair $c = (a, A)$, where a is an access, and A is the set of accesses that could race with the access a. During the backward analysis, the warning corresponding to access a is moved upwards in the CFG. The Kill operation is defined as follows:

$$\mathrm{mayRace}(a', n) = \begin{cases} false & \text{if an access in } n \text{ could not race with } a' \\ & \text{based on the abstract state in } n \\ true & \text{otherwise} \end{cases} \tag{21}$$

$$\mathrm{Kill}_n(X) = \left\{ \langle (a, A), m \rangle \in X \;\middle|\; \begin{array}{c} \forall a' \in A, a' : \\ \mathrm{mayRace}(a', n) = false \\ \vee \text{ a lock is acquired in } n \\ \vee \text{ a lock is released in } n \end{array} \right\} \tag{22}$$

At each node, it is checked if some access $a' \in A$ that could have raced with a in the original location can still race with a in the current program point based on its abstract state. The abstract state, calculated during Goblint's analysis, can include information about threading and locksets, based on which $\mathrm{mayRace}$ calculates its result. If, according to $\mathrm{mayRace}$, a' cannot race with a in node n, hoisting the access a is killed. Otherwise, if a' can race with a, it is still killed if a synchronization event happens in that node.

Instead of having a rather specific condition for the data race warnings, such as a set of held locks, the condition holds the racy accesses themselves. Then, $\mathrm{mayRace}$ can query if, for those accesses, there is any protection used in that program point, which can take into account more information than just the set of held locks, such as any symbolic locks as well. Thus the condition never changes and, unlike the original algorithm, no weakest precondition calculations are needed.

Function $\mathrm{mayRace}$ kills those warnings that are hoisted into a correct path, and removes the redundant hoisting points, thus performing like $\mathrm{alwaysTrue}$ as well. Otherwise, if there is a data race warning that is not killed by $\mathrm{mayRace}$ in some program point, the warning will either reach just the beginning of branching or the thread creation itself, which is not as helpful.

5.1 Examples

In order to illustrate how this approach works, we consider a few positive examples from the Goblint benchmark suite where this approach improves warning positioning.

SMTP Relay Checker. Consider a data race warning from the SMTP Relay Checker[1], a network open mail relay checker, simplified on Fig. 4. The original warning is shown on the access of `cur_threads` on line 29. The actual mistake is that the lock is released in the loop but not acquired back at the end of the loop. During the first iteration of the loop, no data race is possible due to the lock acquired on line 27, but in the next iterations, there might be a data race because of the unlock on line 31. Therefore, the loop iteration path will be incorrect, $\mathrm{mayRace}$ will not kill the hoisting, and the warning will be hoisted to the beginning of the loop on line 30.

To stop hoisting in the incorrect path, an analogue to how variable assignment kills in the AIOB analysis must be implemented for the data race analysis. Such an analogue for data race warnings would be a synchronization event. Then the warning will be hoisted to the highest point where the access would still race but not higher than the closest synchronization event. That is in accordance with the anticipability requirement so that from every highest hoisting point, every path will lead to the warning.

The synchronization events, whenever a lock is acquired or released, consider that if both threads acquire the same lock whenever they may access the same memory location as another thread, there are no data races between those threads. If a lock is acquired and a race is still possible, it is most likely that a wrong lock is acquired, which might be the cause of the data race warning.

Conditional locking. Consider the C code on Fig. 5. The main program creates a thread with a function `t_fun`. In that thread, `mutex1` is acquired before reading from and writing to global variables `myglobal1` and `myglobal2`, after which `mutex1` is released. In the `main` function, depending on the branch, either `mutex2` or `mutex1` is acquired before reading from and writing to the same global variables. Therefore, when the else-branch is selected based on some random value, there is no data race because both reads and writes to both global variables are protected by the same `mutex1`. However, when the if-branch is selected, there might be a data race, as the reads and writes to the global variables are not protected by the same locks.

The analyzer would originally show data race warnings for the accesses on lines 10, 11, 24 and 25 as seen on Fig. 6. But the actual cause of those warnings is not the accesses themselves, but the wrong lock taken on line 20. The original data race warnings are

[1] https://sourceforge.net/projects/smtprc/files/smtprc/smtprc-2.0.3/

```c
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t main_thread_count_mutex;
int cur_threads = 10;
int number_of_threads;

int st(void) {
    int cond;

    while(cond) {
        pthread_mutex_lock(&main_thread_count_mutex);
        cur_threads--;                                  // Original warning
        pthread_mutex_unlock(&main_thread_count_mutex);
    }
}

void main(void) {
    struct timespec tv;
    pthread_t c_tid;
    tv.tv_sec = 0;
    tv.tv_nsec = 500000000;

    pthread_mutex_init(&main_thread_count_mutex, NULL);
    pthread_create(&c_tid, NULL, (void *)st, NULL);

    pthread_mutex_lock(&main_thread_count_mutex);

    while (cur_threads>=number_of_threads) {            // Original warning
        // ...
        pthread_mutex_unlock(&main_thread_count_mutex); // Repositioned warning
        nanosleep(&tv, NULL);
        // missing lock
    }

    pthread_mutex_unlock(&main_thread_count_mutex);
}
```

Figure 4. A simplified data race example from SMTP Relay Checker v2.0.3.

```c
#include <pthread.h>
#include <stdio.h>

int myglobal1, myglobal2;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void *t_fun(void *arg) {
  pthread_mutex_lock(&mutex1);  // Repositioned warning
  myglobal1=myglobal1+1;        // Original warning
  myglobal2=myglobal2+2;        // Original warning
  pthread_mutex_unlock(&mutex1);
  return 0;
}

int main(void) {
  pthread_t id;
  pthread_create(&id, NULL, t_fun, NULL);
  if (...) {
    pthread_mutex_lock(&mutex2); // Repositioned warning
  } else {
    pthread_mutex_lock(&mutex1);
  }
  myglobal1=myglobal1+1;         // Original warning
  myglobal2=myglobal2+1;         // Original warning
  if (...) {
    pthread_mutex_unlock(&mutex2);
  } else {
    pthread_mutex_unlock(&mutex1);
  }

  pthread_join(id, NULL);
  return 0;
}
```

Figure 5. A code example of data races.

```
[Warning][Race] Memory location myglobal1@4:5-4:14 (race with conf. 110):
  write with [mhp:{tid=[main, t_fun@18:3-18:40]}, lock:{mutex1}, thread:[main, t_fun@18:3-18:40]] (10:3-10:24)
  write with [mhp:{tid=[main]; created={[main, t_fun@18:3-18:40]}}, lock:{mutex2}, thread:[main]] (24:3-24:24)
  read with [mhp:{tid=[main, t_fun@18:3-18:40]}, lock:{mutex1}, thread:[main, t_fun@18:3-18:40]] (10:3-10:24)
  read with [mhp:{tid=[main]; created={[main, t_fun@18:3-18:40]}}, lock:{mutex2}, thread:[main]] (24:3-24:24)
[Warning][Race] Memory location myglobal2@4:16-4:25 (race with conf. 110):
  write with [mhp:{tid=[main, t_fun@18:3-18:40]}, lock:{mutex1}, thread:[main, t_fun@18:3-18:40]] (11:3-11:24)
  write with [mhp:{tid=[main]; created={[main, t_fun@18:3-18:40]}}, lock:{mutex2}, thread:[main]] (25:3-25:24)
  read with [mhp:{tid=[main, t_fun@18:3-18:40]}, lock:{mutex1}, thread:[main, t_fun@18:3-18:40]] (11:3-11:24)
  read with [mhp:{tid=[main]; created={[main, t_fun@18:3-18:40]}}, lock:{mutex2}, thread:[main]] (25:3-25:24)
```

Figure 6. Goblint's original warnings for the program on Fig. 5

grouped by the access variables so that for the program on Fig. 5, the accesses on lines 10 and 24, 11 and 25, respectively, are grouped together.

The repositioning algorithm will first decouple the accesses from the groups and then proceed to hoist them on lines 9, 20 and 22. On lines 9 and 20, the warnings' hoistings are killed because a lock is acquired, but on line 22, they are killed because none of the accesses could race due to both threads having the same locks. Therefore, mayRace will remove all the warnings on line 22, and we are left with the warnings of possibly racy accesses on lines 9 and 20. Then, the accesses are regrouped according to the original grouping, and we have successfully repositioned the data race warnings on their actual cause statements. The repositioned warning messages can be seen on Fig. 7. The aforementioned heuristics of killing whenever a lock is acquired or released, however, only work when at least one of the accesses is protected by some mutex.

```
[Warning][Race] Memory location myglobal1@4:5-4:14 (race with conf. 110): (9:3-9:30)
  Possible cause (9:3-9:30)
  Possible cause (20:5-20:32)
  write with [mhp:{tid=[main, t_fun@18:3-18:40]}, lock:{mutex1}, thread:[main, t_fun@18:3-18:40]] (10:3-10:24)
  write with [mhp:{tid=[main]; created={[main, t_fun@18:3-18:40]}}, lock:{mutex2}, thread:[main]] (24:3-24:24)
  read with [mhp:{tid=[main, t_fun@18:3-18:40]}, lock:{mutex1}, thread:[main, t_fun@18:3-18:40]] (10:3-10:24)
  read with [mhp:{tid=[main]; created={[main, t_fun@18:3-18:40]}}, lock:{mutex2}, thread:[main]] (24:3-24:24)
[Warning][Race] Memory location myglobal2@4:16-4:25 (race with conf. 110): (9:3-9:30)
  Possible cause (9:3-9:30)
  Possible cause (20:5-20:32)
  write with [mhp:{tid=[main, t_fun@18:3-18:40]}, lock:{mutex1}, thread:[main, t_fun@18:3-18:40]] (11:3-11:24)
  write with [mhp:{tid=[main]; created={[main, t_fun@18:3-18:40]}}, lock:{mutex2}, thread:[main]] (25:3-25:24)
  read with [mhp:{tid=[main, t_fun@18:3-18:40]}, lock:{mutex1}, thread:[main, t_fun@18:3-18:40]] (11:3-11:24)
  read with [mhp:{tid=[main]; created={[main, t_fun@18:3-18:40]}}, lock:{mutex2}, thread:[main]] (25:3-25:24)
```

Figure 7. Repositioned Goblint's warnings for the program on Fig. 5

6 Implementation

The hoisting (Section 4.1) and sinking (Section 4.2) algorithms were implemented in Goblint with the modifications described in Sections 6.1 and 6.2. In Goblint terminology (see Fig. 1) these are not normal analyses, which form a product of forward data-flow analyses as *the* main constraint system, but are defined as standalone constraint systems, which are solved separately. This allows the backward analysis as defined by Eqs. (4) and (5) to be implemented in Goblint which normally only supports forward analyses. Goblint's default solver TD3 [33] is also used to solve the added constraint systems.

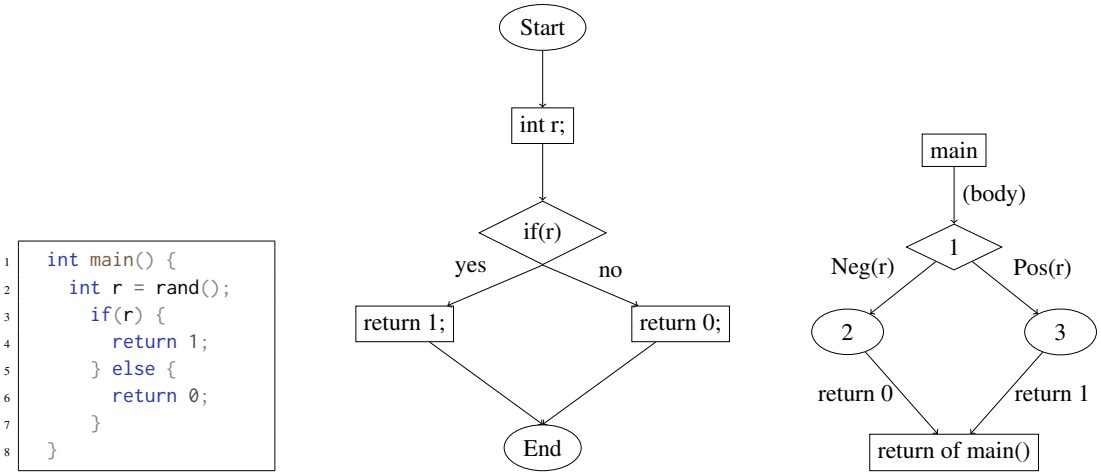
The implementation performs repositioning for array index out-of-bounds and data race warnings as described in Section 5, and is easily extendable to be used for other analyses warnings. To utilize the algorithm for repositioning warnings of another analysis, the condition, killing conditions and a function for discarding redundant hoisting points, like $\mathrm{alwaysTrue}$, have to be defined. The functions $\mathrm{alwaysTrue}$ (Eqs. (10) and (11)) and $\mathrm{mayRace}$ (Eq. (21)) are defined by querying the context-insensitive results (i.e. constraint system solution) of the main analysis. The implementation of repositioning AIOB warnings performed exactly as described in the article by Tukram and Muske [26].

6.1 Adapting the control-flow graph

The proposed two-step alarm repositioning technique [26] works on a control-flow graph (CFG), in which each node denotes a statement in the program, and each edge a possible flow of the program from one node to another. Each such CFG, corresponding to one function, has a *Start* and an *End* node, and all other nodes have a one-to-one correspondence with the statements of the function. An example of such CFG for a simple program with one function (Fig. 8a) is given on Fig. 8b. In Goblint, the CFG representation is different, with the nodes denoting program points and the edges representing statements, illustrated on Fig. 8c.

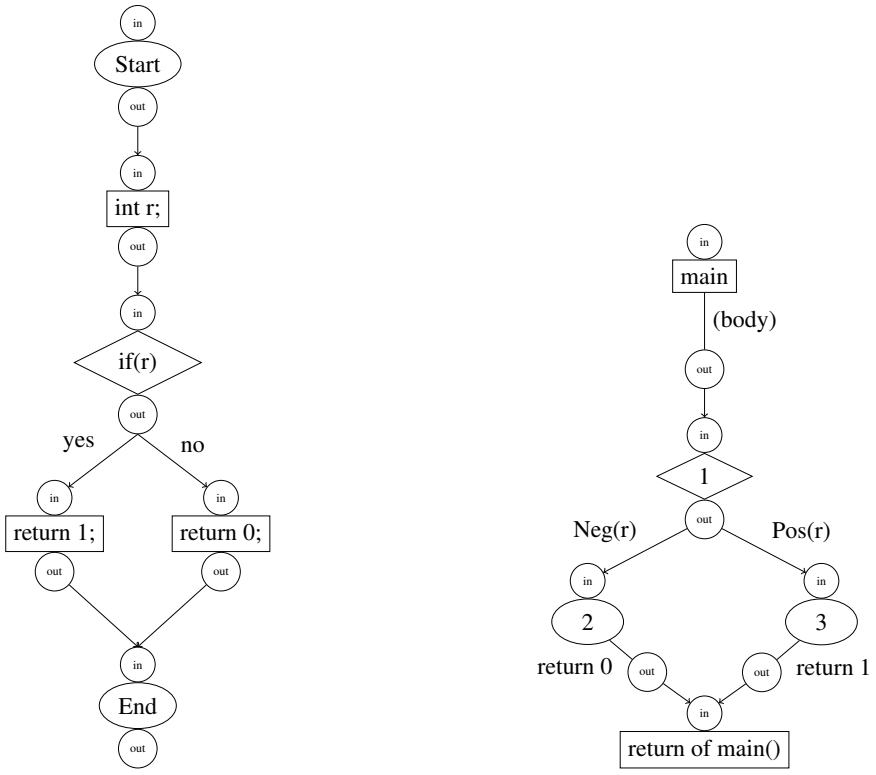
The main difference between these two CFGs is that the states in their corresponding nodes differ. That is because in one (Fig. 8b), the node holds the state *after* executing the corresponding statement, but in the other (Fig. 8c), the corresponding node holds the state *before* executing the corresponding statement – the one on the outgoing edge.

As mentioned in Section 4, the states right before and after executing a statement correspond to the entry and exit of each node, respectively. These states are denoted as *in* and *out*, and placed right before and after each node as seen in the CFG on Fig. 9a. To replicate such node splitting in the CFG representation used in Goblint, the differences must be taken into account. The concept of an entry of a node is the same as in [26], denoted by *in*, which represents the state right before executing a statement. However, to get the corresponding *out*, the transfer function on the outgoing edge must be eval-



(a) A simple program (b) CFG in the algorithm (c) CFG in Goblint

Figure 8. An example of the differences between control flow graphs.



(a) CFG in the algorithm (b) CFG in Goblint

Figure 9. An example of the differences between control flow graphs with added *in* and *out* states.

uated as an extra step for *out* to represent the state right after executing the statement. The described modifications, compared to an example of a CFG corresponding to the description in [26] (Fig. 9a), are illustrated on Fig. 9b, which is the representation used for the implementation. This concerns the equations in the analysis that involve using the state for its calculations.

6.2 Other modifications to the original algorithm

The equations of the algorithm in the Tukram and Muske article [26] assume the existence of functions for calculating the weakest precondition and strongest postcondition. In Goblint, there are no such functions and implementing them is out of the scope of this thesis. This also ensures that the conditions do not change during hoisting or sinking.

Similarly to the unique circumstance described in the end of Section 4.2, where each condition that reaches the program end will be considered for repositioning in the forward analysis, the same unique case can happen during the backward analysis. Therefore, every condition from the exit of the $Start$ node will be used for repositioning during the hoisting to preserve the soundness of the forward analysis:

$$\mathrm{Hoist}^{out}_{Start} = \mathrm{conds}(\mathrm{Ant}^{out}_{Start}) \tag{23}$$

As a design choice from the results of implementation and Goblint's specifics on showing the warnings, the warnings that reach the program start are relocated to their original location, discarding the repositioning.

For the final repositioning in the original algorithm, the sunk alarms will be shown in the sunk position, where the corresponding alarms have been merged. In our algorithm, we will use the sinking for merging, but instead of repositioning the merged alarms in the sunk position, we will reposition them back to the hoisting points. Therefore, the final merged warning messages will be repositioned to the highest hoisting point of all the merged warning messages. Although only one hoisting point out of several different hoisting points is selected, the warning will have traceability links to all the other hoisting points, thus not increasing the overall number of warnings. With this approach, the warnings will be shown closer to their possible causes rather than in a somewhat random position, where the two faulty paths merge.

7 Evaluation

An empirical comparison of the SV-COMP data race benchmark suite assesses the effectiveness and practicality of the implemented data race warning repositioning analysis in Goblint. The evaluation addresses the following quantitative and qualitative research questions:

- **RQ1.** How much does the adapted warning repositioning technique reduce the code traversal between the cause and location of data race warning?
- **RQ2.** How well do the repositioned warnings assist in identifying the cause of the alarm?

The SV-COMP NoDataRace benchmark suite consists of 783 files. Goblint is able to verify 652 of these programs to have no data races and produces data race warnings for 128 programs. In the remaining 3, Goblint either threw an exception or did not complete the analysis within the given time constraint of 60 seconds. With the post-processing algorithm enabled, Goblint was unable to complete the analysis for an additional 15 files within the time constraint of 60 seconds.

The post-processing algorithm for data races is quite expensive, as different from the AIOB warnings post-processing, where the results of the main analysis are queried only when there is an assignment to a variable in condition c, the $\mathrm{mayRace}$ query is run in every program point for several access pairs. Therefore, for files with a higher number of warnings, the algorithm is unable to complete the calculation within a minute, leaving us with 113 programs with repositioned data race warnings.

Answer to RQ1. As seen on Fig. 10, for more than half (98) of the total number (175) of data race warnings, the implemented repositioning algorithm was unable to move the warnings from their original location. This is expected, as due to the design choice described in Section 6.2, the algorithm positions the warnings back to their original locations if they reach the function start during hoisting. That is because the algorithm is intraprocedural, and reaching function start means that the possible cause is outside the current function.

The warning was moved up in 77 cases. The answer to RQ1, based on the results on Fig. 10, is that the warning repositioning reduced the code traversals needed to reach the cause of alarm by an average of 1 line, including the cases where the warning did not move, and by an average of 2 lines if the cases where the warning did not move are not included.

Instead of moving warnings back to their original locations when hoisting reaches the function start, informing the developer of the possible cause being outside the function

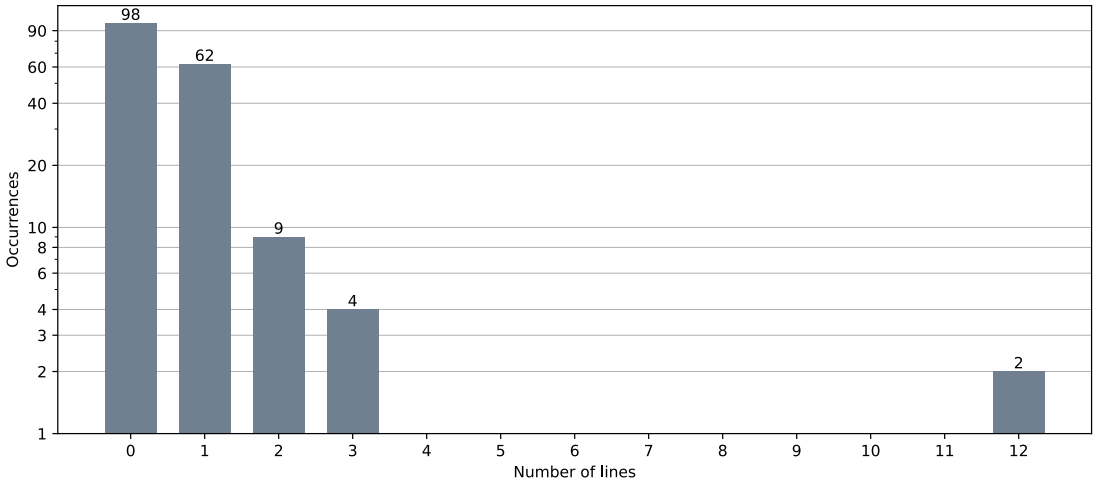


Figure 10. Number of lines between original warning location and repositioned warning location *with* positioning the warnings back to their original locations in case they reach the function start during hoisting.

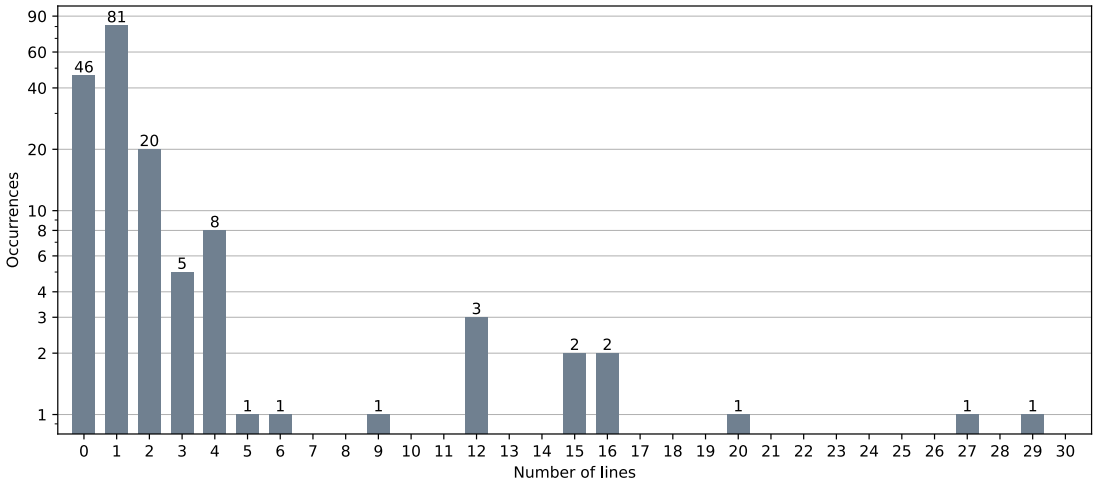


Figure 11. Number of lines between original warning location and repositioned warning location *without* positioning the warnings back to their original locations in case they reach the function start during hoisting.

where the original warning is shown yielded the results in Fig. 11. The analysis was not completed within 60 seconds for another two files, producing warnings for 111 files. The number of warnings not moved from the original location is now reduced to 46, indicating that in 52 cases, the warning reaches the function start. This suggests that an interprocedural analysis instead of an intraprocedural could possibly produce better results. In the remaining cases with the number of lines being 0, manual inspection revealed that most cases where the warning was not relocated were due to if-statements or different loops, meaning that the warning reposition was killed by branching.

The number of lines required to reach the cause of the alarm was reduced by a maximum of 29 lines and, on average, two (including 0) and three lines (not including 0), respectively, which is slightly better than the design choice used in the evaluation run summarized on Fig. 10. However, due to the algorithm being intraprocedural, the number of lines is dependent on the function lengths, and the warnings that moved give a good indication that in the case of longer functions and files, the code traversals could indeed be reduced.

This concludes that the answer to RQ1 (*How much does the adapted warning repositioning technique reduce the code traversal between the cause and location of data race warning?*) is 1 to 3 lines on average. The limitations with the biggest impact on the number of lines reduced are the algorithm being intraprocedural and the way the algorithm handles branching.

Answer to RQ2. We looked into the files where the number of lines between the original warning location and its repositioned location was highest, seen in Table 1. In the Linux drivers, in all cases, the warnings were repositioned to the beginning of if branches, not reaching any actual possible causes. However, even if the warnings had

Table 1. Files with the largest number of lines between the original warning location and its repositioned location from Fig. 10.

	Warning location (line)		
File	Original	Repositioned	Difference
`linux-3.14-drivers-media-platform-marvell-ccic-cafe_ccic.ko.cil-1.i`	5706	5694	12
	5717	5714	3
`linux-3.14-drivers-media-platform-marvell-ccic-cafe_ccic.ko.cil-2.i`	5706	5694	12
	5717	5714	3
	6160	6157	3
`time_var_mutex.i`	701	698	3

been hoisted upwards from the if conditions, they would have reached the function start, as the causes were not in those functions.

In `time_var_mutex.c`[2], seen on Fig. 12, the warnings in one function were hoisted to the beginning of the if branch; and in the other function, the hoisting was killed by an unlocking statement (Eq. (21)) in one branch, and by the branches joining (Eq. (8)), in the other branch. The highest hoisting point on line 13 indeed gives some insight into what might be the cause. However, the locking event on line 23 would be better for understanding the cause because removing or relocating the unlocking statement from line 13 would not fix the race, but the write and read on lines 16 and 17, respectively, must be protected by the same lock that is taken on line 23. Yet, that is not reached as the highest hoisting point again due to the join on the if statement (Eq. (8)) on line 24 killing the hoisting.

While in the code examples on Figs. 4 and 5, the repositioned warnings indeed assisted in identifying the cause of the alarm, we were unable to find any good examples from the SV-COMP benchmark suite. With that, the answer to RQ2 (*How well do the repositioned warnings assist in identifying the cause of the alarm?*) is that the repositioned warnings assist in identifying the cause of the alarm only in simple, specific cases, given as examples in Section 5.1, whereas, for the cases in the SV-COMP benchmark suite, the resulting warning position was not helpful for understanding the warnings.

The evaluation of the method on the SV-COMP benchmark suite confirmed the doubts that arose while formalizing the analyses. The details of the limitation of how the algorithm handles branching and the possible improvements to overcome the issue are discussed in the following section.

[2]The code of `time_var_mutex.i` before C preprocessing.

```
...
int block;
int busy; // flag indicating whether the block has been allocated to an inode
int inode;
pthread_mutex_t m_inode; // protects the inode
pthread_mutex_t m_busy; // protects the busy flag

void *allocator(void *_){
  pthread_mutex_lock(&m_inode);
  if(inode == 0){
    pthread_mutex_lock(&m_busy);
    busy = 1;
    pthread_mutex_unlock(&m_busy);  // Highest hoisting point
    inode = 1;
  }
  block = 1;              // Original warning (write) and highest hoisting point
  assert(block == 1);     // Original warning (read)
  pthread_mutex_unlock(&m_inode);
  return NULL;
}

void *de_allocator(void *_){
  pthread_mutex_lock(&m_busy);
  if(busy == 0){
    block = 0;            // Original warning (write) and highest hoisting point
    assert(block == 0);   // Original warning (read)
  }
  pthread_mutex_unlock(&m_busy);
  return ((void *)0);
}

int main() {
  // ...
  pthread_create(&t1, 0, allocator, 0);
  pthread_create(&t2, 0, de_allocator, 0);
  // ...
}
```

Figure 12. `time_var_mutex.c` from the SV-COMP benchmark suite.

8 Discussion

This section gives a thorough review of the benefits and limitations of the proposed method, as well as proposals for future work.

8.1 Identified benefits

It was shown that post-processing methods for single-threaded flow properties could be adapted in multi-threaded programs. More specifically, the anticipable and available alarm condition analyses can be adapted to accommodate repositioning data race warnings.

As the algorithm is proposed to be general, it was possible to make the implementation so that the repositioning algorithm can be easily extended to work with warnings produced by other analyses. This reduces the implementation effort significantly, as for the many different analyses implemented in Goblint, extending them with this post-processing method only requires some parts of the repositioning analysis to be implemented instead of a separate new analysis for each.

The original article does not specify explicitly where the states in the Control Flow Graph (CFG) come from. However, in Goblint, the pre-existing query system proved extremely useful for obtaining results that the post-processing analysis can utilize. This approach offers an advantage over methods that only have limited information about warnings, such as an analysis that only has information from the warnings themselves, "namely their type (e.g., NULL dereference) and their location in the program's source code" [22].

We have identified the main weaknesses of the chosen method and gained a deeper understanding of the domain of post-processing methods overall. The thesis gives way for future work in the domain, with some directions proposed later in this section.

8.2 Identifying all possible causes

As mentioned in Section 4.1, the number of warnings can increase when hoisting the warnings. Another example is given on Fig. 13. In addition to illustrating how the algorithm increases the number of warnings, it shows how a fault can remain undetected because of the anticipability restriction.

In this example, the original warning is hoisted to program points F and C. There is a fault in program point F, where the anticipability is killed, and the highest hoisting point is identified by Eq. (9). However, there is no fault in the program point C, where the other hoisting will get stuck according to Eq. (8). But there is another fault in the program that reaches both of the highest hoisting points along paths $A \to B \to C$ and $A \to B \to E \to F$ accordingly.

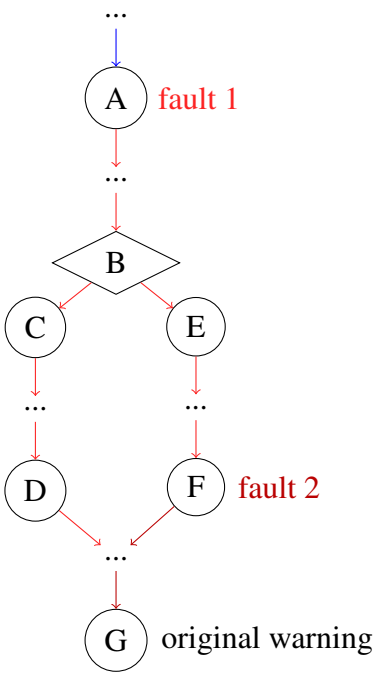


Figure 13. Example of an undetected possible cause due to the anticipability restriction.

To overcome this limitation, one could complement the algorithm such that the backward data-flow analysis is used to hoist the original warnings to the highest program point where the condition causing the alarm evaluates to true, in addition to where the conditions are anticipable.

In a subsequent paper, Muske et al. [27] propose a refined method based on whether alarm conditions are *live* rather than anticipable. They propose hoisting warnings to the highest point where the condition is live, i.e., there exists a non-killing path to a warning. This could be a solution for the proposed algorithm not finding every possible cause for the warning.

They also make clear what sound warning repositioning means: a repositioned alarm is a false positive if and only iff all its original alarms are false positives. This is why they insisted on the anticipable alarm condition because it is conservative w.r.t. control dependence between the if statement and an error. For this example, if B prevents $C \rightarrow D$ (whenever the error condition holds), moving G to A would turn a false positive into a definite error. It's more obvious for examples like:

```
A:  x = 0;
B:  if (x != 0)
F:    1/x;
```

So if the analysis does not understand the condition and the algorithm repositions the warning out of the if-statement, then the false positive has been turned into a definite

warning.

In this case, displaying the warning in program point A would only escalate the false positive. But by displaying the warning with traceability links from the anticipability analysis, giving the developer the full information, it can be more easily determined that the warning was indeed a false alarm. However, a thorough analysis, implementation and evaluation of the fit for using live alarm condition analysis and its combination with anticipable alarm condition analysis for improving the explainability of data race warnings are out of the scope of this thesis and left for future work.

8.3 Causality analysis vs warning reduction

For repositioning data race warnings, the algorithm turned out to be more suited for detecting the cause of the data race rather than reducing the number of alarms. The data race warnings in Goblint are grouped by the variable with possibly racing accesses. Therefore, we have leveraged the algorithm to move the already grouped data race warnings up in the CFG to their possible causes, for which we found and proposed a method. Furthermore, it would be interesting to compare this method with other causality analysis methods, which is another large domain [17] of methods that could be used to improve the explainability of warnings.

Using the repositioning algorithm, an additional level of grouping could have been proposed to further group the warnings by other metrics, such as held locks or threading information. This is another possible direction for future work on the topic.

8.4 Trace-based analysis for explaining data race warnings

It is not clear if the data-flow analysis is the best approach for finding explanations for data race warnings, or if some trace analysis could be a better fit for solving this problem. There are several sound trace analyses proposed for finding the causes of warnings to understand where the warning comes from [13, 22, 28] and it is worth investigating how would these work for explaining data race warning causes compared to the data-flow analysis approach. This thesis can be used for comparisons in future works.

8.5 Context-sensitivity

The aforementioned backward and forward data-flow analyses in Goblint are implemented context-insensitively, meaning that the transfer functions are evaluated on joined contexts. However, joining different contexts reasonably is not always possible, for example, in cases of a function that is called both in single-threaded and multi-threaded contexts. Therefore, to achieve the truthfulness of the method, the analyses must be reimplemented to also consider the contexts of the warning messages.

9 Conclusion

Program verification is vital for reducing the negative impact of software failures, and static analysis tools help detect programming errors in code early in the development process. Although the developers already use several static analyzers, the complex analyses have yet to find a way to be integrated into the development workflow. The main obstacles preventing the use of sound analyzers in the development processes are the significant amount of false alarms and the complexity of the produced warning messages. Since verification problems are undecidable, it is necessary to complement the exploration of new analyses and domains with other methods, such as warning post-processing. Therefore a method was sought for post-processing the warnings of Goblint, a static analyzer for multi-threaded C programs, that is the state-of-the-art data-race freedom verifier for C.

To achieve this, we first conducted a literature review of methods for reducing false alarms in sound static analyses in general, with a focus on post-processing methods. Based on the review and the thesis goals, an alarm repositioning method was chosen. Several modifications were necessary to adapt the algorithm for the implementation within the Goblint analyzer. Taking into account the differences between the CFG proposed in the article and the CFG structure in Goblint, the data-flow equations for both backward and forward analyses were formalized, and the required modifications were described. The analyses were implemented in the Goblint analyzer.

The core contribution of this thesis was extending the method for repositioning, originally designed for single-threaded data-flow properties, to work with data race warnings in multi-threaded programs. This was a challenging task since the conditions for detecting data race freedom are not as clear as they are for assertion-based analyses. Additionally, data race warnings come in pairs, and it was necessary to ensure that the access pairs were preserved even after repositioning both warnings. The study showed that post-processing methods for single-threaded flow properties could be adapted in multi-threaded programs.

The study evaluates the effectiveness and practicality of the implemented data race warnings repositioning analysis in Goblint in addressing two research questions, namely, the reduction of code traversals and the identification of the cause of the alarm in data race warnings. The SV-COMP NoDataRace benchmark suite was used to assess the performance of the method. The study found that the warning repositioning technique reduced the code traversals; however, the number of lines the warnings moved was, on average, one to three lines. Additionally, the repositioned warnings only assisted in identifying the cause of the alarm in simple, specific cases, whereas for the studied examples in the benchmark suite, the resulting warning position was not helpful for understanding the warnings. The study highlights the limitations of the algorithm in handling branching and suggests possible improvements to overcome the issue.

The proposed method for data race warning repositioning was thoroughly reviewed, highlighting its benefits and limitations. The limitations having the biggest impact on the results were the algorithm being intraprocedural and the restrictions the anticipability requirement imposes on how the algorithm handles branching. The thesis gives way for future work in the domains of causality analysis and warning reduction, with proposals to improve the presented method, as well as investigating other techniques in comparison to this work.

References

[1] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 1–19, New York, NY, USA, 1970. Association for Computing Machinery.

[2] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. Side-effecting constraint systems: A swiss army knife for program analysis. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, pages 157–172, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[3] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481, 2016.

[4] Dirk Beyer. 12th competition on software verification (sv-comp 2023) results of the competition, 2023. https://sv-comp.sosy-lab.org/2023/results/results-verified/, accessed: 2023-04-20.

[5] Dirk Beyer. Competition on software verification and witness validation: Sv-comp 2023. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 495–522, Cham, 2023. Springer Nature Switzerland.

[6] Paul Black, Mark Badger, Barbara Guttman, and Elizabeth Fong. Dramatically reducing software vulnerabilities: Report to the white house office of science and technology policy, 2016-12-01 2016.

[7] David Bühler, Pascal Cuoq, and Boris Yakobowski. *Eva - The Evolved Value Analysis plug-in*.

[8] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 332–343, 2016.

[9] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 332–343, New York, NY, USA, 2016. Association for Computing Machinery.

[10] CIL. Source code. https://github.com/goblint/cil.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[12] David Delmas and Jean Souyris. Astrée: From research to industry. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, pages 437–451, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[13] Chaoqiang Deng and Patrick Cousot. Responsibility analysis by abstract interpretation, 2019.

[14] Julian Erhard, Simmo Saan, Sarah Tilscher, Michael Schwarz, Karoliine Holter, Vesal Vojdani, and Helmut Seidl. Interactive abstract interpretation: Reanalyzing whole programs for cheap, 2022.

[15] Goblint. Goblint documentation. Annotating code. https://goblint.readthedocs.io/en/latest/user-guide/annotating/.

[16] Goblint. Source code. https://github.com/goblint/analyzer.

[17] Joseph Y. Halpern and Judea Pearl. Causes and explanations: A structural-model approach. part i: Causes. *The British Journal for the Philosophy of Science*, 56(4):843–887, 2005.

[18] Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363–387, 2011. Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing.

[19] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 672–681. IEEE Press, 2013.

[20] U. Khedker, A. Sanyal, and B. Sathe. *Data Flow Analysis: Theory and Practice*. CRC Press, 2017.

[21] Herb Krasner Krasner. *The Cost of Poor Software Quality in the US: A 2022 Report*. Consortium for Information & Software Quality (CISQ), Dec 2022.

[22] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. Pse: Explaining program failures via postmortem static analysis. *SIGSOFT Softw. Eng. Notes*, 29(6):63–72, oct 2004.

[23] Antoine Miné and David Delmas. Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In *2015 International Conference on Embedded Software (EMSOFT)*, pages 65–74, 2015.

[24] Anders Møller and Michael I. Schwartzbach. Static program analysis, October 2022. Department of Computer Science, Aarhus University, https://cs.au.dk/~amoeller/spa/.

[25] Tukaram Muske and Alexander Serebrenik. Survey of approaches for postprocessing of static analysis alarms. *ACM Comput. Surv.*, 55(3), feb 2022.

[26] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. Repositioning of static analysis alarms. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 187–197, New York, NY, USA, 2018. Association for Computing Machinery.

[27] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. Reducing Static Analysis Alarms Based on Non-impacting Control Dependencies. In Anthony Widjaja Lin, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 115–135, Cham, 2019. Springer.

[28] Xavier Rival. Understanding the origin of alarms in astrée. In Chris Hankin and Igor Siveroni, editors, *Static Analysis*, pages 303–319, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[29] Simmo Saan. Witness generation for data-flow analysis. Master's thesis, University of Tartu, 2020.

[30] Simmo Saan, Michael Schwarz, Kalmer Apinis, Julian Erhard, Helmut Seidl, Ralf Vogler, and Vesal Vojdani. Goblint: Thread-modular abstract interpretation using side-effecting constraints. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 438–442, Cham, 2021. Springer International Publishing.

[31] Simmo Saan, Michael Schwarz, Julian Erhard, Manuel Pietsch, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. Goblint: Autotuning thread-modular abstract interpretation. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 547–552, Cham, 2023. Springer Nature Switzerland.

[32] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, mar 2018.

[33] Helmut Seidl and Ralf Vogler. Three improvements to the top-down solver. *Mathematical Structures in Computer Science*, 31(9):1090–1134, 2021.

[34] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. Static analysis with demand-driven value refinement. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[35] Vesal Vojdani. Laboratory for software science: Research: Truthful explainable program analysis, 2023. https://sws.cs.ut.ee/Main/Research, accessed: 2023-05-08.

[36] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: The goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 391–402, New York, NY, USA, 2016. Association for Computing Machinery.

Appendix

I Goblint implementation

The OCaml implementatation of the repositoning algorithm in Goblint can be found from the following Goblint fork repository on GitHub, under the tag "`msc-thesis`": https://github.com/karoliineh/goblint/releases/tag/msc-thesis.

For the usage of Goblint, please refer to Goblint documentation: https://goblint.readthedocs.io/en/latest/. The reposition analysis can be turned on with the configuration option **`--enable ana.warn-postprocess.enabled`**.

II Evaluation files and script

The SV-COMP benchmark suite results for the evaluation and the evaluation script are included in a zip file. There are two folders, `evaluation1` and `evaluation2`, with the warnings produced by Goblint from the two SV-COMP benchmark runs. In both folders, there is one JSON file with the results for each analyzed file and a Jupyter Notebook script for extracting the results from the JSON files.

III Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Karoliine Holter**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   "Adapting an Alarm Repositioning Algorithm to Data Races",

   supervised by Vesal Vojdani and Simmo Saan.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe on other persons' intellectual property rights or rights arising from the personal data protection legislation.

Karoliine Holter
09.05.2023