DISSERTATIONES
INFORMATICAE
UNIVERSITATIS
TARTUENSIS
79

SIMMO SAAN

Correctness Witnesses for Thread-Modular Program Analysis

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

79

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
79

SIMMO SAAN

Correctness Witnesses for Thread-Modular Program Analysis

UNIVERSITY OF TARTU
Press
1632

Institute of Computer Science, Faculty of Science and Technology, University of Tartu, Estonia.

Dissertation has been accepted for the commencement of the degree of Doctor of Philosophy (PhD) in Computer Science on March 3, 2026 by the Council of the Institute of Computer Science, University of Tartu.

Supervisor

Assoc. Prof. Vesal Vojdani
University of Tartu
Tartu, Estonia

Opponents

Prof. Patrick Cousot
New York University
New York, USA

Prof. Dirk Beyer
Ludwig-Maximilians-Universität München
Munich, Germany

The public defense will take place on April 23, 2026 at 13:00 in Narva Rd. 18-1020.

The publication of this dissertation was financed by the Institute of Computer Science, University of Tartu.

ISSN 2613-5906 (print) ISSN 2806-2345 (pdf)
ISBN 978-9908-57-173-7 (print) ISBN 978-9908-57-174-4 (pdf)

Copyright © 2026 by Simmo Saan

University of Tartu Press
http://www.tyk.ee/

To witnessing soundness

ABSTRACT

Multi-threaded programs can be efficient by executing multiple operations concurrently, but they can be hard to write correctly. They are also difficult to test because thread scheduling is non-deterministic. Thus, formal verification is vital for developing multi-threaded software. This can be automated using static analyzers — other programs which try to verify software automatically. Automated software verifiers are themselves complex programs which may also contain bugs and, thus, produce incorrect results. To make their verdicts more trustworthy, verifiers may produce witnesses that expose their reasoning about the analyzed program. These proof objects allow the analysis results to be inspected by humans or even other programs called witness validators. Unfortunately, there were previously no verification witnesses for explaining the correctness of multi-threaded programs. Therefore, this thesis aims to fill the research gap by using witnesses to make the analysis of concurrent programs trustworthy, but also performant and precise.

Our solution relies on abstract interpretation, which is a successful method for verifying real-world software and multi-threaded programs. First, this thesis investigates the value analysis of concurrent programs using abstract interpretation. We find existing approaches to be incomparable and devise new thread-modular analyses, which are more precise than prior work. Second, this thesis studies how abstract interpretation can benefit from witnesses, both in terms of performance and precision. We achieve both benefits by introducing a novel unassume operation, and use it to develop the first witness validator using abstract interpretation. Third, this thesis joins the previous directions and proposes correctness witnesses for concurrent programs, thereby filling the research gap. We propose two different flavors of witnesses, which respectively correspond to a local and a global view of concurrency, and experimentally evaluate them.

All of the ideas are implemented in the GOBLINT static analysis framework, which is based on abstract interpretation. Thus, this thesis also provides an up-to-date overview of the framework and its progress. Our work with witnesses has revealed pre-existing bugs in GOBLINT and their fixes have improved it overall. Therefore, this thesis advances the state-of-the-art of abstract interpretation and verification witnesses. Witnesses allow us to develop better static analyzers, which in turn help developers write correct multi-threaded programs.

CONTENTS

List of Figures **11**

List of Tables **12**

List of Listings **14**

Glossary **15**

1. Introduction **17**
1.1. Problems and Goals 18
1.2. Methodology 20
1.3. Overview 21
1.4. Contributions 22

2. Background **25**
2.1. Notation 25
2.2. Language 26
2.3. Order Theory 28
2.4. Abstract Interpretation 29
2.4.1. Abstract Domains 29
2.4.2. Abstract Semantics 30
2.4.3. Value Domains 30
2.4.4. Constraint Systems 31
2.4.5. Side-Effecting Constraint Systems 33
2.5. Witnesses 35
2.5.1. Violation. 35
2.5.2. Correctness 35
2.5.3. YAML Format. 37
2.5.4. Applications 39
2.6. SV-COMP 41
2.6.1. Properties 41
2.6.2. Tasks. 42
2.6.3. Categories 45

3. GOBLINT Analyzer **46**
3.1. Introduction 46
3.2. Software Architecture 47
3.3. Verification Approach 49
3.3.1. Values and Reachability 49
3.3.2. Overflows 51
3.3.3. Data Races 51
3.3.4. Memory Safety 52

3.3.5. Memory Leaks . . . 52
3.3.6. Termination . . . 53
3.4. GOBLINT in SV-COMP . . . 53
3.4.1. SV-COMP 2021 . . . 56
3.4.2. SV-COMP 2022 . . . 56
3.4.3. SV-COMP 2023: Autotuning . . . 58
3.4.4. SV-COMP 2024: Memory Safety and Termination . . . 60
3.4.5. SV-COMP 2025 . . . 61
3.4.6. Threats to Validity . . . 62
3.5. Data Race Analysis . . . 64
3.5.1. Pairwise Race Analysis . . . 64
3.5.2. History-Based Thread IDs . . . 66
3.5.3. Excluding Races with Thread IDs . . . 70
3.5.4. Simpler Thread IDs . . . 71
3.6. Related Work . . . 73
3.7. Conclusion and Future Work . . . 75

4. Witness-Guided Abstract Interpretation **77**
4.1. Introduction . . . 77
4.2. Preliminaries . . . 79
4.3. Initialization-Based Approaches . . . 81
4.4. Unassuming . . . 82
4.4.1. Specification . . . 83
4.4.2. Naïve Definition . . . 85
4.4.3. Dual-Narrowing . . . 87
4.4.4. Pointers . . . 90
4.5. Propagating Algorithm . . . 92
4.5.1. Assuming . . . 93
4.5.2. Unassuming . . . 96
4.5.3. Local Iteration . . . 99
4.5.4. Pointers . . . 102
4.6. Implementation . . . 107
4.6.1. Witness-Guided Verification . . . 107
4.6.2. Witness Validation . . . 109
4.6.3. Witness Generation . . . 110
4.7. Evaluation . . . 111
4.7.1. Precision . . . 111
4.7.2. Performance . . . 113
4.7.3. SV-COMP 2024 . . . 114
4.7.4. Threats to Validity . . . 118
4.8. Related Work . . . 120
4.9. Conclusion and Future Work . . . 122

5. Thread-Modular Abstract Interpretation **124**
5.1. Introduction . . . 124
5.2. Preliminaries . . . 125
5.2.1. Language . . . 125
5.2.2. Interleaving Semantics . . . 126
5.2.3. Local Trace Semantics . . . 127
5.2.4. Constraints . . . 130
5.3. Prior Analyses . . . 131
5.3.1. Baseline . . . 131
5.3.2. Miné's Analysis . . . 134
5.3.3. Vojdani's Analysis . . . 136
5.4. Improved Analyses . . . 141
5.4.1. Protection-Based Reading . . . 142
5.4.2. Lock-Centered Reading . . . 149
5.4.3. Write-Centered Reading . . . 153
5.4.4. Combined Reading . . . 155
5.5. Relational Analysis . . . 158
5.5.1. Mutex-Meet . . . 158
5.5.2. Improvements . . . 161
5.6. Implementation . . . 164
5.6.1. Differences from Theory . . . 165
5.6.2. Assuming Globals . . . 166
5.7. Evaluation . . . 167
5.7.1. Non-relational Analyses . . . 167
5.7.2. Relational Analysis . . . 169
5.7.3. Threats to Validity . . . 171
5.8. Related Work . . . 172
5.9. Conclusion and Future Work . . . 174

6. Thread-Modular Correctness Witnesses **176**
6.1. Introduction . . . 176
6.2. The Local View . . . 177
6.3. Implementation of the Local View . . . 184
6.3.1. Witness Generation . . . 184
6.3.2. Witness-Guided Verification . . . 185
6.4. Evaluation of the Local View . . . 186
6.4.1. Results . . . 187
6.4.2. Threats to Validity . . . 189
6.5. The Global View . . . 190
6.6. Ghost Witnesses . . . 192
6.6.1. Validity w.r.t. Interleaving Semantics . . . 193
6.6.2. Validity w.r.t. Local Trace Semantics . . . 194
6.6.3. YAML Format . . . 196

6.7. Implementation of the Global View. 198
6.7.1. Witness Generation 198
6.7.2. Witness Validation 201
6.8. Evaluation of the Global View 202
6.8.1. Results. 203
6.8.2. Bugs 205
6.8.3. Threats to Validity. 206
6.9. Related Work 207
6.10. Conclusion and Future Work 208

7. Conclusion **211**

Bibliography **213**

Acknowledgements **237**

Data Availability Statement **238**

Sisukokkuvõte (Summary in Estonian) **239**

Curriculum Vitae **241**

Elulookirjeldus (Curriculum Vitae in Estonian) **242**

List of Original Publications **243**

LIST OF FIGURES

1. Automated software verification and witness validation workflow . . . 18
2. Chapter dependencies of the thesis . . . 22

3. Scoring schema for the SV-COMP verification track . . . 43
4. Scoring schema for the SV-COMP validation track . . . 44

5. Physical lines of code of OCaml in Goblint over time by author . . . 47
6. SV-COMP 2024 *Overall* category results as a quantile plot . . . 61

7. Plots of relations for unassume specification **S7** . . . 85
   a. $0 = \mathsf{x} \leq \mathsf{y}$ . . . 85
   b. $0 \leq \mathsf{x} \leq \mathsf{y}$ . . . 85
   c. $0 \leq \mathsf{x}, \mathsf{y}$ . . . 85
8. Phases of propagating algorithm for $[\![\mathsf{assume}(\mathsf{x} + 2 = 42)]\!]^\sharp\, d$ where $d = \{\mathsf{x} \mapsto [40, \infty]\}$ . . . 95
9. Phases of propagating algorithm for $[\![\mathsf{unassume}(\mathsf{x}+2 \geq 42)]\!]^\sharp\, d$ where $d = \{\mathsf{x} \mapsto [40, 40]\}$ . . . 98
10. Phases of propagating algorithm for $[\![\mathsf{assume}(\mathsf{p} = \&\mathsf{i} \wedge *\mathsf{p} = 0)]\!]^\sharp\, d$ . . . 104
    a. Iteration 1 with $d = \{\mathsf{p} \mapsto \{\&\mathsf{i}, \&\mathsf{j}\}, \mathsf{i} \mapsto [0, \infty], \mathsf{j} \mapsto [0, \infty]\}$ . . . 104
    b. Iteration 2 with $d = \{\mathsf{p} \mapsto \{\&\mathsf{i}\}, \mathsf{i} \mapsto [0, \infty], \mathsf{j} \mapsto [0, \infty]\}$ . . . 104
11. Phases of propagating algorithm for $[\![\mathsf{unassume}((\mathsf{p} = \&\mathsf{i} \vee \mathsf{p} = \&\mathsf{j}) \wedge *\mathsf{p} \geq 0)]\!]^\sharp\, d$ . . . 106
    a. Iteration 1 with $d = \{\mathsf{p} \mapsto \{\&\mathsf{i}\}, \mathsf{i} \mapsto [0, 0], \mathsf{j} \mapsto [0, 0]\}$ . . . 106
    b. Iteration 2 with $d = \{\mathsf{p} \mapsto \{\&\mathsf{i}, \&\mathsf{j}\}, \mathsf{i} \mapsto [0, \infty], \mathsf{j} \mapsto [0, 0]\}$ . . . 106
12. Cross-framework witness validation effort for literature examples . . . 113
13. YAML correctness witness confirmation rates for all verifier-validator pairs in SV-COMP 2024 . . . 116
14. Witness validation effort for Goblint in SV-COMP 2024 . . . 117

15. Lattice of thread-modular semantics and analyses as a Hasse diagram . . . 143
16. Non-relational analysis times for Goblint benchmarks . . . 168
17. Relational analysis times for Goblint benchmarks . . . 170

18. Witness-guided verification effort for Goblint benchmarks using the local view . . . 188
19. Witness-guided verification effort change for Goblint benchmarks using the local view . . . 188
20. Cross-framework witness validation effort using the global and the local view . . . 205

LIST OF TABLES

1. Existing witnesses in SV-COMP 19

2. SV-COMP participations of GOBLINT, with relevant publications and artifacts 54
3. SV-COMP categories for GOBLINT 54
4. SV-COMP *NoDataRace* category rankings 57
   a. SV-COMP 2021 (demo) 57
   b. SV-COMP 2022 (demo) 57
   c. SV-COMP 2023 57
   d. SV-COMP 2024 57
   e. SV-COMP 2025 57
5. Average change in resource usage due to disabling all concurrency analyses for single-threaded programs 59
6. Abstract interpreters for C programs 74

7. Analysis progress and result for the program in Listing 3 from scratch. 78
8. Analysis progress and result for the program in Listing 3 unassuming invariant $0 \leq x \leq 40$ on line 3. 79
9. Supported specifications for unassume operators 86
10. Supported specifications for propagating unassume operators. 102
11. Analysis progress and result for the program in Listing 3 unassuming invariant $1 \leq x \leq 40$ on line 4. 108
12. Analysis progress and result for the program in Listing 3 unassuming invariant $1 \leq x \leq 40$ on line 4 with a widening token 109
13. Witness validation verdicts for literature examples 112
14. Witness-guided verification effort for select GOBLINT benchmarks . . 114
15. Witness validation consistency for GOBLINT in SV-COMP 2024. . . . 116

16. Pairs of observable and observing concurrency primitives by programming construct. 128
17. Relational analysis precision summary 169
18. Relational analysis precision for GOBLINT benchmarks 171

19. Analysis progress and result for the program in Listing 4 from scratch. 179
20. Analysis progress and result for the program in Listing 4 starting with the flow-insensitive invariant $[-10, 10]$ 180
21. Analysis progress and result for the program in Listing 4 unassuming invariant $-10 \leq g \leq 10$ on lines 10 and 18 with widening tokens . . . 183
22. Witness-guided verification precision for GOBLINT benchmarks using the local view 187
23. Witness-guided verification effort for select GOBLINT benchmarks from Table 14 using the local view with Protection-Based Reading 189

24. Cross-framework witness-only validation using the global and the local view 203
    a. Correct programs 203
    b. Incorrect programs 203
25. Cross-framework witness validation and total effort using the global and the local view 204

LIST OF LISTINGS

1. YAML correctness witness example 38
    a. Program. 38
    b. YAML witness 38

2. Program for illustrating history-based thread IDs 67

3. Program for illustrating witness-guided abstract interpretation 78

4. Program for illustrating the local view 178
5. YAML witness for the program in Listing 4 181
6. Program for illustrating the global view. 191
    a. Program. 191
    b. Program with ghost instrumentation 191
7. Example of encoding of an atomic block as a critical section 195
8. YAML ghost witness for the program in Listing 6 197
9. Program with a multiply-protected global variable 200

GLOSSARY

Symbols

$\mathcal{A}$ Set of CFG arcs/edges

$\mathcal{A}_f$ Set of CFG arcs/edges of function $f \in \mathcal{F}$

$\mathbb{D}$ Abstract domain

$\mathcal{E}$ Set of side-effect–free expressions

$\mathcal{F}$ Set of functions

$\mathcal{G}$ Set of global variables

$\mathcal{G}_{\text{rw}}[m]$ Set of protected variables of $m \in \mathcal{M}$

$\mathcal{G}_{\text{w}}[m]$ Set of write-protected variables of $m \in \mathcal{M}$

$\mathcal{I}$ Set of invariant expressions

$\mathcal{M}$ Set of mutexes

$\mathcal{M}_{\mathcal{G}}$ Set of atomicity mutexes

$\mathcal{M}_{\text{rw}}[g]$ Set of protecting mutexes of $g \in \mathcal{G}$

$\mathcal{M}_{\text{w}}[g]$ Set of write-protecting mutexes of $g \in \mathcal{G}$

$\mathcal{N}$ Set of CFG nodes

$\mathcal{N}_f$ Set of CFG nodes of function $f \in \mathcal{F}$

$\mathcal{O}_n$ Set of n-ary operators ($n \in \mathbb{N}$, $n \geq 1$)

$\mathbb{R}$ Relational abstract domain

ret_f Return node of function $f \in \mathcal{F}$

$\mathcal{S}$ Set of primitive statements (aka primitives)

st_f Start/entry node of function $f \in \mathcal{F}$

$\mathbb{T}$ Abstract domain for thread IDs

$\mathbb{V}$ Non-relational abstract domain for values

$\mathcal{V}$ Set of variables

$\mathcal{V}_{\text{👻}}$ Set of ghost variables

$\mathcal{X}$ Set of constraint system unknowns

$\mathcal{X}$ Set of local variables

$\mathcal{X}_f$ Set of local variables of function $f \in \mathcal{F}$

Acronyms

CFG Control-flow graph

LLoC Logical lines of code (i.e., only lines with executable code; excluding empty lines, comments and declarations)

LoC Physical lines of code (i.e., including empty lines and comments)

MHP May-happen-in-parallel

NNF Negation normal form

OOB Out-of-bounds

SV-COMP Competition on Software Verification

UAF Use-after-free

1. INTRODUCTION

Everyone wants software to *just work* (correctly). Depending on the application, bugs can range from mildly annoying to truly life-critical. Hence, any decent software developer wants to write bug-free programs. The most common method for achieving this goal (or at least trying to) is testing, which comes in many different shapes and sizes. While testing can catch obvious issues and prevent regressions, it often cannot guarantee complete correctness: fully exhaustive testing of non-trivial programs tends to be impractical, if not impossible. For such guarantees we turn to formal *software verification*.

Formal verification requires precise definitions of what is considered a bug and what it means to work correctly. The process of verification aims to find out whether a given program satisfies or violates a given *property*. High-level functional properties describe the intent of the program and must be provided by the developer, e.g., the program correctly sorts data. Low-level non-functional properties describe general issues which cause the program to misbehave or crash, e.g., the program never divides by zero. This thesis focuses on the latter kind.

The verification of software could be done manually, automatically or using some combination of the two. Manual software verification involves writing mathematical proofs, preferably in a proof assistant — another program which checks the proofs. Automated software verification involves running source code through a static analyzer — another program which tries to prove correctness automatically. Since the manual approach may not scale well and the automated approach may not be clever enough, combining them gives the best of both worlds. This thesis focuses on the automatic approach which is better suited for non-functional properties.

Automated software verifiers are themselves complex programs which may also be faulty and, thus, produce incorrect results. Although they should be based on sound mathematical theory, formal verification of their implementations tends to be out of reach for the time being. To increase trust in their verdicts, verifiers may produce *witnesses* that expose their reasoning about the analyzed program. Such proof objects allow humans to inspect and confirm or reject analysis results. Moreover, this may be done automatically using a *witness validator* — another program which checks the witness. The automated software verification and witness validation workflow is depicted in Fig. 1 on the following page.

The use of witnesses as a standardized way to communicate between different automated software verifiers was pioneered by Beyer et al. [Bey+15]. They introduced an analyzer-agnostic format for explaining property violations using counterexamples, which describe how exactly the property can be violated. This witness format was later extended to explain program correctness using invariants [Bey+16], a standard tool in formal methods. Subsequently, witnesses and their validation have become integral parts of the Competition on Software Verification (SV-COMP) — the annual evaluation and comparison of the state of the art in automated software verification [BS25].

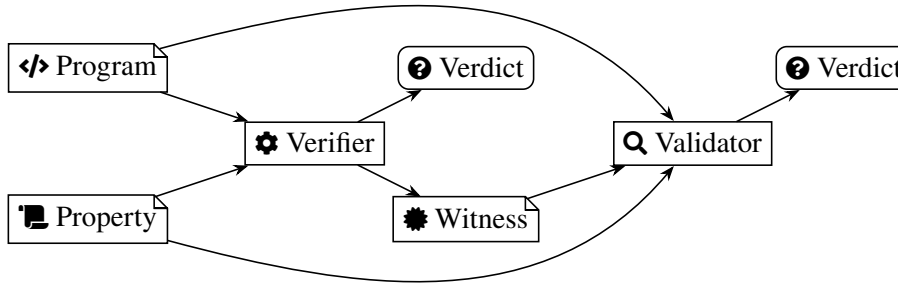


Figure 1. Automated software verification and witness validation workflow

Everyone wants software to be fast as well. A common method for achieving this goal is using *concurrency*, i.e., executing multiple operations at the same time. This also comes in various forms: computational clusters distribute work over many servers, system applications spread work over multiple CPU cores using threads, and web applications invoke server requests asynchronously with the UI. This thesis focuses on the middle case of concurrency by multi-threading.

The execution of multi-threaded programs is essentially non-deterministic: the OS is in charge of choosing which threads execute when. The unpredictability of thread scheduling makes the writing of multi-threaded code particularly error-prone. Moreover, the non-determinism poses a major challenge for testing such programs. Therefore, formal verification becomes especially important for multi-threaded software.

Standing on the foundation of software verification, this thesis delves into witnesses and concurrency, ultimately bringing the two together. Witnesses allow us to develop better static analyzers for multi-threaded programs, which in turn help developers write correct concurrent software, matching the expectations of users.

1.1. Problems and Goals

The overall goal of the thesis is *performant, precise and trustworthy analysis of concurrent programs*. The following describes the choices we make and the problems we solve to achieve this goal. In particular, we focus on using witnesses to this end.

Performance. Concurrent executions of threads may interleave in exponentially many ways. Thus, approaches which exhaustively analyze each individual interleaving do not scale to real-world programs. Instead, we choose a *thread-modular* approach where threads are analyzed separately, and their interferences are abstracted and propagated between threads. More specifically, our method of choice is *abstract interpretation* which is known for its scalability. It has proven successful in the efficient verification of large real-world software [Cou+09; Bau+21] and multi-threaded programs [Min12; MM17; FK12].

It is known how to speed up some approaches, such as model checking, by

providing them pre-existing information like invariants from witnesses. However, it is not known about abstract interpretation. Given all that, the performance subgoal is to speed up abstract interpretation by guiding it with invariants from witnesses.

Precision. Many thread-modular abstract-interpretation–based analyses have been proposed in the literature. The first precision subgoal is to investigate and improve the precision of two prominent thread-modular analysis approaches from literature.

Analogously to performance, it is known how to improve the precision of some approaches, such as model checking, by providing them invariants from witnesses. Again, it is not known about abstract interpretation, so the second precision subgoal is to improve the precision of abstract interpretation by guiding it with invariants from witnesses.

Trustworthiness. Table 1 summarizes which witnesses exist for which programs in SV-COMP, revealing an obvious gap: no agreed-upon witness format exists for the correctness of concurrent programs. This is in no small part due to the divide between the approaches used by analyzers, ranging from interleaving to thread-modular, making it challenging to exchange information. The first trustworthiness subgoal is to fill the gap, i.e., provide a witness format suitable for certifying the correctness of concurrent programs.

Table 1. Existing witnesses in SV-COMP

	Witness type	
Program type	Violation	Correctness
Single-threaded	[Bey+15; Aya+24]	[Bey+16; Aya+24]
Multi-threaded	[BF20]	*This work*

There is little benefit to just defining a new witness format and generating such witnesses with different tools. The usefulness of a witness format comes from using it for greater good. Hence, the second trustworthiness subgoal is to validate the results of one analysis of concurrent programs using another via the proposed witnesses.

Goals. To summarize, the thesis has the following goals:

- **G1.** Speed up abstract interpretation by guiding it with invariants from witnesses.
- **G2.** Investigate and improve the precision of two prominent thread-modular analysis approaches from literature.
- **G3.** Improve the precision of abstract interpretation by guiding it with invariants from witnesses.
- **G4.** Provide a witness format suitable for certifying the correctness of concurrent programs.
- **G5.** Validate the results of one analysis of concurrent programs using another via the proposed witnesses.

1.2. Methodology

Design and Implementation. We begin with theory, using numerous examples to detail the problem and illustrate our proposed solution at a high level. Then, we formalize our solution mathematically and prove its soundness, if necessary.

We continue with practice, implementing our proposed approach in order to apply it to real programs. This allows us to demonstrate the generality and applicability of our approach by evaluating it on more and/or larger programs than a few examples.

It is often advantageous to work on the theory and the implementation in parallel. The latter can quickly reveal unforeseen issues in the theory before trying to prove soundness, yielding a feedback loop in the process.

Evaluation. With the introduction of correctness witnesses, Beyer et al. [Bey+16] also posed three pillars for their evaluation:

Same-framework consistency. Witnesses produced by a verifier should be validated by a validator using the *same framework*. If this is not the case, then the format or its implementation is inconsistent. Thus, this establishes a baseline for the following.

Cross-framework validation. Witnesses produced by a verifier should be validated by a validator using a *different framework*. If this is not the case, then the format or its implementation is too specific to a certain framework and does not generalize.

Content-effort dependence. "[The] effort and feasibility of validation depends on witness contents" [Bey+16]. This can manifest itself in two ways:

Performance. Validation of a witness uses *less resources* than it took to produce the witness. Although this comparison can be done both in the same-framework and the cross-framework case, the former is more insightful as it eliminates the framework as a factor.

Precision. A validator *can validate* a witness (produced by a verifier using a different framework) for a program which *could not be verified* by a verifier using the same framework as the validator. This corresponds to the cross-framework case and should not happen in the same-framework case.

Throughout our work on witnesses, we seek to establish these in experimental evaluations. Although the pillars refer to witness validation, they also apply to witness-guided verification (elaborated in Section 2.5.2).

Archival and Reproducibility. The benchmarks, tools, scripts, raw results and witnesses from all our evaluations are archived as *artifacts* (Data Availability Statement on page 238). They allow our evaluations to be reproduced and/or extended by future work.

Most of the artifacts have also been evaluated by artifact evaluation committees of the related conferences. All of the evaluated artifacts have been awarded the *Available* and *Functional* badges, meaning all of them are publically available and have successfully been executed by the committees to reproduce the results. Some of the artifacts have even been awarded the *Reusable* badge.

Engineering and Maintenance. Since we evaluate our proposed approaches via their implementations, quality of the implementations is critical to the validity of our conclusions [Alg+11a; RED16]. As such, we must value software engineering and maintenance efforts which yield mature, well-tested and optimized tools. This minimizes the risk of our evaluations being accidentally undermined by unrelated bugs in the tools.

We evaluate our techniques using actively maintained state-of-the-art tools, as evidenced by their successful participation in SV-COMP. Furthermore, we maintain the tool GOBLINT and participate in the competition, which drives continuous evaluation and improvement. Our top priority is soundness: across 5 years, SV-COMP has revealed countless soundness issues in GOBLINT, all of which we have fixed.

1.3. Overview

Figure 2 on the following page illustrates the structure of the thesis:

Chapter 2 provides the background on the two main themes of the thesis: abstract interpretation and witnesses. Additionally, it defines mathematical and programming language notations used throughout the thesis.

Chapter 3 introduces the GOBLINT static analysis framework, which is based on abstract interpretation and used for implementations of the following chapters. Section 3.4 presents its achievements in SV-COMP. Section 3.5 describes its advances in data race analysis, which is GOBLINT's shtick.

Chapter 4 investigates how abstract interpretation can benefit from witnesses, both in terms of precision (**G3**) and performance (**G1**), by introducing a novel unassume operation. This development yields GOBLINT VALIDATOR, the first witness validator using abstract interpretation.

Chapter 5 takes a deep dive into the value analysis of concurrent programs using abstract interpretation. An incomparability of existing approaches starts a journey which results in new thread-modular analyses, which are more precise than prior work (**G2**).

Chapter 6 builds on the two previous chapters to finally propose correctness witnesses for concurrent programs, thereby filling the research gap (**G4**). It proposes two different flavors which respectively correspond to a local and a global view of concurrency. Both flavors are validated by experimental evaluation (**G5**).

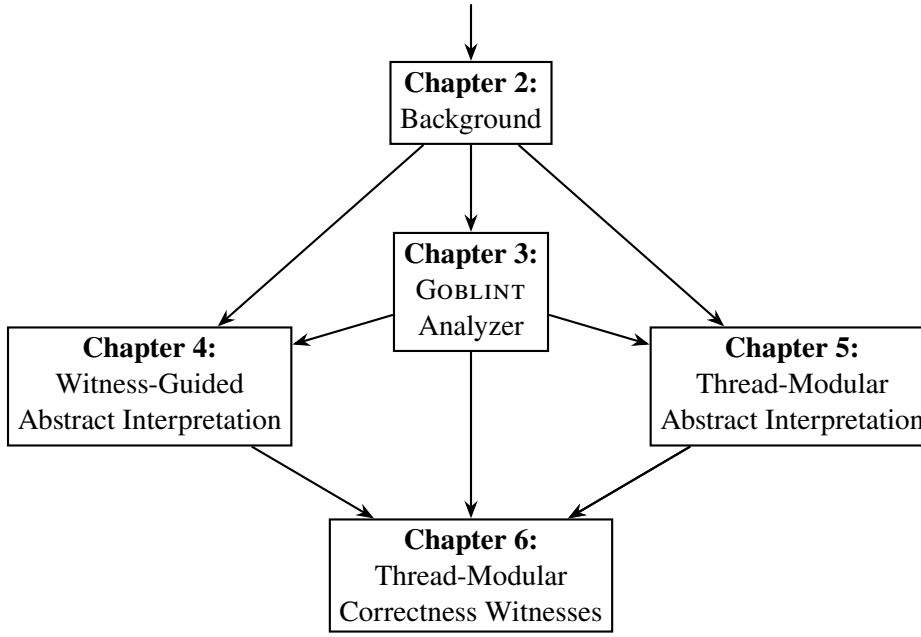


Figure 2. Chapter dependencies of the thesis

1.4. Contributions

Chapter 3 is based on the tool papers

- **Simmo Saan**, Michael Schwarz, Kalmer Apinis, Julian Erhard, Helmut Seidl, Ralf Vogler, and Vesal Vojdani. "GOBLINT: Thread-Modular Abstract Interpretation Using Side-Effecting Constraints (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 12652. Springer, Mar. 2021, pp. 438–442. DOI: 10.1007/978-3-030-72013-1_28.
- **Simmo Saan**, Michael Schwarz, Julian Erhard, Manuel Pietsch, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. "GOBLINT: Autotuning Thread-Modular Abstract Interpretation (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13994. Springer, Apr. 2023, pp. 547–552. DOI: 10.1007/978-3-031-30820-8_34.
- **Simmo Saan**, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl. "GOBLINT: Abstract Interpretation for Memory Safety and Termination (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 14572. Springer, Apr. 2024, pp. 381–386. DOI: 10.1007/978-3-031-57256-2_25.

The author has led the development of GOBLINT and its participation in SV-COMP for 5 years. Not reflected in the papers, as the GOBLINT representative

on the SV-COMP jury, the author reviewed other tools, their tool papers and benchmarks. These papers are a real team effort, both the improvements in GOBLINT and the writing. The last two papers incorporate work of students from Technische Universität München (TUM). Compared to the papers, the chapter includes: extended description of the GOBLINT framework, and its SV-COMP 2022 and 2025 participations which have no corresponding tool papers.

Additionally, Section 3.5 on data race analysis includes an adaptation of history-based thread IDs from a paper corresponding to Chapter 5 [Sch+23b]. Compared to the paper, the section includes: description of data race analysis, improved predicates for history-based thread IDs, and description of existing thread IDs.

Chapter 4 is based on the paper

- **Simmo Saan**, Michael Schwarz, Julian Erhard, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. "Correctness Witness Validation by Abstract Interpretation". In: *Verification, Model Checking, and Abstract Interpretation*. Vol. 1. LNCS 14499. Springer, Jan. 2024, pp. 74–97. DOI: 10.1007/978-3-031-50524-9_4.

which received the Best Paper Award at VMCAI 2024. The author contributed most of the work: concept of the unassume operator with three instances (naïve, dual-narrowing and propagating), implementation, evaluation and most of the writing. The co-authors contributed: one unassume operator instance (exploding) and improvement of the writing. Compared to the paper, the chapter includes: more (detailed) examples, extended description of the propagating algorithm, and more general handling of pointers.

Additionally, the implementation was evaluated in SV-COMP 2024. Thus, Section 4.7.3 originates from the tool paper

- **Simmo Saan**, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl. "GOBLINT VALIDATOR: Correctness Witness Validation by Abstract Interpretation (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 14572. Springer, Apr. 2024, pp. 335–340. DOI: 10.1007/978-3-031-57256-2_17.

The author contributed: validator implementation and the writing. The co-authors contributed: general improvement of GOBLINT.

Chapter 5 is based on the paper

- Michael Schwarz, **Simmo Saan**, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. "Improving Thread-Modular Abstract Interpretation". In: *Static Analysis*. LNCS 12913. Springer, Oct. 2021, pp. 359–383. DOI: 10.1007/978-3-030-88806-0_18.

The author contributed: implementation, evaluation and feedback from these to fix/improve the analyses. The co-authors contributed: theory of local traces, the analyses, their proofs and the writing. Compared to the paper, the chapter includes: descriptions of prior analyses, more examples, and implementation details.

Additionally, the relational analysis originates from the paper

- Michael Schwarz, **Simmo Saan**, Helmut Seidl, Julian Erhard, and Vesal Vojdani. "Clustered Relational Thread-Modular Abstract Interpretation with Local Traces". In: *Programming Languages and Systems*. LNCS 13990. Springer, Apr. 2023, pp. 28–58. DOI: 10.1007/978-3-031-30044-8_2.

The author contributed: implementation of history-based thread IDs and Mutex-Meet analysis. The co-authors contributed: theory, the analyses, their proofs, implementation of Mutex-Meet improvements, evaluation and the writing.

Chapter 6 consists of two halves.

The first half on the local view is based on unpublished work that was initially part of the paper corresponding to Chapter 4 [Saa+24d] but excluded in later revisions. The author contributed all of the work: concept, implementation, evaluation and the writing.

The second half on the global view is based on the paper

- Julian Erhard, Manuel Bentele, Matthias Heizmann, Dominik Klumpp, **Simmo Saan**, Frank Schüssele, Michael Schwarz, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. "Correctness Witnesses for Concurrent Programs: Bridging the Semantic Divide with Ghosts". In: *Verification, Model Checking, and Abstract Interpretation*. Vol. 1. LNCS 15529. Springer, Jan. 2025, pp. 74–100. DOI: 10.1007/978-3-031-82700-6_4.

The author contributed: witness generation (Section 6.7.1) and its implementation. The co-authors contributed: formalization, proofs, validation implementation (in UGemCutter), evaluation and the writing.

2. BACKGROUND

Before jumping into problem-solving, we review the necessary background on the two main themes of the thesis: abstract interpretation and witnesses.

First, we establish general mathematical notations (Section 2.1) and introduce some order theory (Section 2.3), which forms the basis of abstract interpretation. Then, we define a concurrent programming language (Section 2.2) and recall the framework of abstract interpretation (Section 2.4) for statically analyzing programs. Finally, we introduce software verification witnesses (Section 2.5) and give an overview of the software verification competition (Section 2.6), which prominently uses witnesses.

2.1. Notation

The following mathematical notations are used throughout the thesis:

- The relation $\stackrel{\text{def}}{=}$ indicates a definition, as opposed to any equality.
- The relation $\stackrel{\text{def}}{\iff}$ indicates a definition, as opposed to any equivalence.
- The set $\mathbb{B} \stackrel{\text{def}}{=} \{\mathsf{true}, \mathsf{false}\}$ for *booleans*.
- The set $\mathbb{N} \stackrel{\text{def}}{=} \{0, 1, 2, \dots\}$ for *natural numbers*.[1]
- The set 2^S denotes the *power set* of the set S.
- The set S^* denotes the set of (possibly empty) *finite sequences* with elements from the set S.
- The set $X \rightharpoonup Y$ denotes the set of *partial functions* from the set X to the set Y.
- The set dom f denotes the *domain* of the (partial) function f.
- The function $f|_S$ denotes the *restriction* of the function f to the domain S.

The following notations are borrowed from functional programming:

- The formula $f\ x$ means applying the function f to the argument x.
- The formula $\lambda x.\ e$ is an unnamed function with parameter x and body e.
- Functions with multiple arguments are curried:
  - The type $X \to Y \to Z$ is the type of a function effectively taking two arguments of types X and Y, and returning a value of type Z.
  - The formula $f\ x\ y$ means applying the function f to the arguments x and y.
- The formula **if** c **then** t **else** e is equivalent to $\begin{cases} t & \text{if } c, \\ e & \text{otherwise.} \end{cases}$
- The formula **let** $x = e_1$ **in** e_2 is equivalent to e_2 where the variable x is equal to e_1.

[1] Zero is natural in computer science.

- The "name" $(x_1, \ldots, x_n)$ in a function parameter or let binding is a (tuple) *pattern match* where $x_1, \ldots, x_n$ are variables corresponding to components of the tuple.
- The name _ in a function parameter, let binding or pattern match is a *wildcard* used for intentionally unused values.

The following notation is additionally introduced:

- The formula **let** x **such that** c **in** e is equivalent to e where the variable x has a value such that c holds. In other words, it binds the existential witness x from $\exists x : c$ for use in e. It is only used in contexts where the existence is guaranteed by a surrounding conditional formula.

The following notations are used for mappings, which are often viewed as sets of bindings (of the form $x \mapsto v$):

- The formula $\varnothing$ indicates the empty mapping (in addition to the empty set).
- The formula $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ indicates a finite mapping with the given set of n bindings.
- The formula $\{x \mapsto v_x \mid x \in X\}$ is equivalent to $\lambda x.\, v_x$, indicating a mapping with the bindings given using set-builder notation.
- The formula $f[x \mapsto v]$ is equivalent to $\lambda y.\, \textbf{if } y = x \textbf{ then } v \textbf{ else } f\, y$, indicating a *function update*.
- The formula $f[x \mapsto v_x \mid x \in X]$ is equivalent to $\lambda y.\, \textbf{if } y \in X \textbf{ then } v_x \textbf{ else } f\, y$, indicating multiple function updates using set-builder notation.
- The formula $f \cup g$ indicates the disjoint union of the mappings f and g with disjoint domains (in addition to indicating the set union).

2.2. Language

Throughout the thesis, we use the following imperative programming language with concurrency support.

Functions. Let $\mathcal{F}$ be the set of program functions (as simply abstract names). We assume the program execution begins from the function $\mathsf{main} \in \mathcal{F}$. We will only use functions for concurrently-executing threads (as so-called thread templates). Thus, we omit function calls, their arguments and return values for simplicity of presentation. The omitted features are orthogonal to this work and can be handled using existing approaches [ASV12].

Variables. Let $\mathcal{V} \stackrel{\text{def}}{=} \mathcal{X} \cup \mathcal{G}$ be the set of program variables, consisting of disjoint subsets of *local* and *global* variables. Let $\mathcal{X} \stackrel{\text{def}}{=} \bigcup_{f \in \mathcal{F}} \mathcal{X}_f$ be a disjoint union, where $\mathcal{X}_f$ is the set of local variables of the function $f \in \mathcal{F}$. We do not specify the types of variables, which may be integers, booleans, pointers, etc. However, we assume programs to be well-scoped and well-typed. Local variables are uninitialized at function entry, possibly having any value. Global variables are initialized at program start to some initial value based on their types, e.g., 0 for integers (like in the C programming language).

Expressions. Let $\mathcal{E}$ be the set of side-effect–free expressions. We do not fix the language of expressions but assume that it at least includes variables and constants. Since expressions are used for logical conditions, the language should include booleans and boolean expressions, or assign truth values to some expressions of other types (e.g., 0 is false and non-0 is true, like in C).

Mutexes. Let $\mathcal{M}$ be the set of mutexes, distinct from program variables. We assume the mutexes are non-recursive, i.e., one thread cannot lock a mutex multiple times without unlocking it in between.

Statements. Let $\mathcal{S}$ denote the set of primitive statements (aka primitives). This includes the following sequential primitives:

Assignment. $v \leftarrow e$ assigns the value of the expression $e \in \mathcal{E}$ to the variable $v \in \mathcal{V}$.

Non-deterministic assignment. $\mathsf{havoc}(V)$ assigns arbitrary values to the variables $V \subseteq \mathcal{V}$. This can be used to model the environment, e.g., user input, possibly with subsequent assumptions (described below) on the variables for more precise modeling.

Assertion. $\mathsf{assert}(e)$ continues execution only if the expression $e \in \mathcal{E}$ evaluates to true and abnormally terminates (i.e., crashes) the program otherwise.

Assumption. $\mathsf{assume}(e)$ continues execution only if the expression $e \in \mathcal{E}$ evaluates to true and has unspecified behavior otherwise. This is only used to model conditional control flow (as *guards*) or the environment, and does not occur in programs otherwise. Thus, the unspecified behavior does not cause semantic issues.

No-op. skip does nothing. This can be used for empty branches in conditional control flow.

Concurrency Primitives. Moreover, $\mathcal{S}$ includes the following primitives for concurrent programs:

Thread creation. $\mathsf{create}(f)$ creates a new thread which starts executing the function $f \in \mathcal{F}$ concurrently with the creator thread.

Locking. $\mathsf{lock}(m)$ waits for the mutex $m \in \mathcal{M}$ to be unlocked (if it is not already) and then locks it. We assume a thread does not lock a mutex which it already holds.

Unlocking. $\mathsf{unlock}(m)$ unlocks the mutex $m \in \mathcal{M}$. We assume a thread does not unlock a mutex which it does not hold.

Compound Statements. Statements which contain other statements are called compound statements, for example:

Sequencing. $s_1; s_2$ first executes s_1 and then s_2. More generally $s_1; \ldots; s_n$ executes the statements in order.

Branching. **if** e **then** s_1 **else** s_2 executes s_1 if the expression $e \in \mathcal{E}$ evaluates to true and s_2 otherwise.

Looping. **while** e **do** s repeatedly executes s as long as the expression $e \in \mathcal{E}$ evaluates to true.

Although (non-trivial) programs are written as compound statements (per function $f \in \mathcal{F}$), we abstract the latter into corresponding control-flow graphs (CFGs) by using primitive assume statements to model conditional control flow.

Control-Flow Graphs. Each function $f \in \mathcal{F}$ is represented by a CFG which is a directed graph with

1. the set of nodes $\mathcal{N}_f$, representing program locations of the function;
2. the set of arcs/edges $\mathcal{A}_f \subseteq \mathcal{N}_f \times \mathcal{S} \times \mathcal{N}_f$, representing control flow between the locations;
3. the unique start/entry node st_f;
4. the unique return node ret_f;

satisfying the property that every node with out-degree ≥ 2 has assume labels on all outgoing arcs.[2] For the compound statements listed above, this is ensured by the standard CFG construction, which introduces branching nodes with exactly out-degree 2 and assume labels with complementary expressions on the outgoing arcs. However, our weaker requirement also admits other compound statements used in theory (e.g., non-deterministic branching) and practice (e.g., `switch`). Note that we allow nodes with a single outgoing assume-labeled arc, as they can be used to precisely model the environment (after a havoc). Functions are disjoint, therefore, the control flow of the entire program is captured by:

$$\mathcal{N} \stackrel{\text{def}}{=} \bigcup_{f \in \mathcal{F}} \mathcal{N}_f, \qquad \mathcal{A} \stackrel{\text{def}}{=} \bigcup_{f \in \mathcal{F}} \mathcal{A}_f.$$

2.3. Order Theory

Let S be a set.

Definition 1 (Partial order)**.** A relation $\sqsubseteq \subseteq S \times S$ is called a *partial order* if it is:

1. reflexive: $x \sqsubseteq x$ for all $x \in S$;
2. antisymmetric: $x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$ for all $x, y \in S$;
3. transitive: $x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$ for all $x, y, z \in S$. ⌟

Definition 2 (Minimal element)**.** An element $x \in S$ is called a *minimal element w.r.t. the partial order* $\sqsubseteq$ if there does not exist $y \in S$ such that $y \sqsubseteq x \wedge y \neq x$. ⌟

Definition 3 (Maximal element)**.** An element $x \in S$ is called a *maximal element w.r.t. the partial order* $\sqsubseteq$ if there does not exist $y \in S$ such that $x \sqsubseteq y \wedge y \neq x$. ⌟

Note that a partial order may have multiple minimal and maximal elements.

Definition 4 (Least element)**.** An element $x \in S$ is called the *least element w.r.t. the partial order* $\sqsubseteq$ if $x \sqsubseteq y$ for all $y \in S$. ⌟

Definition 5 (Greatest element)**.** An element $x \in S$ is called the *greatest element w.r.t. the partial order* $\sqsubseteq$ if $y \sqsubseteq x$ for all $y \in S$. ⌟

[2] This property is later used in Section 3.5.2 when defining $\mathcal{P}$.

The least element is often denoted $\bot$ and called *bottom*. The greatest element is often denoted $\top$ and called *top*.

Note that a partial order does not have a least/greatest element if it has multiple minimal/maximal elements, respectively. The least/greatest element is unique, if it exists.

Definition 6 (Ascending chain)**.** A sequence $(x_i)_{i \in \mathbb{N}}$ from S is called *an ascending chain w.r.t. the partial order* $\sqsubseteq$ if $i \leq j \implies x_i \sqsubseteq x_j$ for all $i, j \in \mathbb{N}$. ⌟

Definition 7 (ACC)**.** A partial order $\sqsubseteq$ satisfies the *ascending chain condition* if every ascending chain w.r.t. $\sqsubseteq$ is ultimately stable, i.e., $\exists k \in \mathbb{N} : \forall n \in \mathbb{N}.\ k \leq n \implies x_k = x_n$. ⌟

Ascending chains which are not ultimately stable are often called *infinite strictly ascending chains*. *Descending chains* and the *descending chain condition* (DCC) are defined dually. Ascending/descending chains are often also called increasing/decreasing chains.

Definition 8 (Monotonic function)**.** Let $\sqsubseteq$ be a partial order on the set S and $\preceq$ be a partial order on the set T. A function $f : S \to T$ is called *monotonic* if $x \sqsubseteq y \implies f\,x \preceq f\,y$ for all $x, y \in S$. ⌟

Definition 9 (Strict function)**.** A function $f : S \to S$ is called *strict* if $f \bot = \bot$. ⌟

2.4. Abstract Interpretation

In an ideal world, we would like static analysis to be:

Sound. If the program contains a bug, then the analysis will find it. Or contrapositively, if the analysis claims the program to be bug-free, then it indeed is.

Complete. If the analysis finds a bug, then it actually is a bug. Or contrapositively, if the program is bug-free, then the analysis claims it as such.

Terminating. The analysis always terminates.

Unfortunately, Rice's theorem states that static analysis is undecidable, i.e., it is impossible to have all three simultaneously.

To counter the inevitable, we give up on completeness and turn to the framework of *abstract interpretation* as introduced by Cousot and Cousot [CC77; CC92]. Using over-approximation, it yields sound and terminating static analyses. We briefly recall relevant notions of abstract interpretation here. For in-depth exposition, refer to Miné [Min17] and Cousot [Cou21].

2.4.1. Abstract Domains

Let S denote the set of all concrete program states. Its subsets are abstracted by an *abstract domain* $\mathbb{D}$ consisting of the following [Min17]:

- a *partial order* $\sqsubseteq$, modeling the relative precision of abstract states;

- a monotonic *concretization* function $\gamma : \mathbb{D} \to 2^S$, mapping an abstract state to the set of concrete states it represents, i.e., $x \sqsubseteq y \implies \gamma\, x \subseteq \gamma\, y$ for all $x, y \in \mathbb{D}$;
- a least element $\bot$, representing unreachability, i.e., $\gamma\, \bot = \varnothing$;
- a greatest element $\top$, representing triviality, i.e., $\gamma\, \top = S$;
- sound abstractions *join* ($\sqcup$) and *meet* ($\sqcap$) of $\cup$ and $\cap$ on S, respectively, i.e., $\gamma\, x \cup \gamma\, y \subseteq \gamma\, (x \sqcup y)$ and $\gamma\, x \cap \gamma\, y \subseteq \gamma\, (x \sqcap y)$ for all $x, y \in \mathbb{D}$;
- a *widening* (∇) operator, computing upper bounds that ensure termination in abstract domains with infinite strictly ascending chains, i.e.,
  1. $x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$ for all $x, y \in \mathbb{D}$;
  2. for every sequence $(y_i)_{i \in \mathbb{N}}$ from $\mathbb{D}$, the sequence $(x_i)_{i \in \mathbb{N}}$ defined by $x_0 \stackrel{\text{def}}{=} y_0$, $x_{i+1} \stackrel{\text{def}}{=} x_i \nabla y_{i+1}$ stabilizes, i.e., $\exists k \in \mathbb{N} : x_i = x_{i+1}$;
- a *narrowing* (Δ) operator, recovering some precision given up by widening while terminating in abstract domains with infinite strictly descending chains, i.e.,
  1. $x \sqcap y \sqsubseteq x \Delta y \sqsubseteq x$ for all $x, y \in \mathbb{D}$;
  2. for every sequence $(y_i)_{i \in \mathbb{N}}$ from $\mathbb{D}$, the sequence $(x_i)_{i \in \mathbb{N}}$ defined by $x_0 \stackrel{\text{def}}{=} y_0$, $x_{i+1} \stackrel{\text{def}}{=} x_i \Delta y_{i+1}$ stabilizes.

In practice, widening/narrowing is also used in abstract domains where all strictly ascending/descending chains are finite but very long.

2.4.2. Abstract Semantics

Functions $[\![s]\!] : 2^S \to 2^S$ for statements $s \in \mathcal{S}$ model the concrete semantics of a program. It is necessary to provide concrete semantics for primitive statements, while the ones for compound statements can be defined in a generic manner. We omit these definitions for simplicity.

An abstract interpreter uses an abstract domain $\mathbb{D}$ and abstractions $[\![s]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ of primitives s to model the abstract semantics of a program. An abstract operator $[\![s]\!]^\sharp$ for a statement s is *sound* if $[\![s]\!]\, (\gamma\, d) \subseteq \gamma\, ([\![s]\!]^\sharp\, d)$ for all $d \in \mathbb{D}$. Abstract operators are strict, meaning they do not spuriously introduce reachable states from an unreachable one.

Fixpoint iteration (potentially with widening and narrowing) is used to compute for each program location an abstract state, which represents a superset of all reaching concrete program states. The resulting abstract states may be used to check whether the program satisfies a given property.

2.4.3. Value Domains

The most common and the most intuitive abstract domains are *value domains*, which provide abstractions of values for program variables $\mathcal{V}$. They can be subdivided into *relational* and *non-relational* domains: the former (can) relate values of different variables, while the latter do not.

Non-relational Domains. These domains usually take the form of a mapping $\mathbb{D} = \mathcal{V} \to \mathbb{V}$, where $\mathbb{V}$ is an abstract domain that only abstracts values of an individual variable, not the entire program state. For example, values of numeric variables are often abstracted using the *interval* domain [CC77].

Abstract semantics for assignment in a non-relational domain can be defined in terms of abstract expression evaluation $\mathbb{E}[\![e]\!]^\sharp : \mathbb{D} \to \mathbb{V}$ ($e \in \mathcal{E}$) as

$$[\![x \leftarrow e]\!]^\sharp\, d \stackrel{\text{def}}{=} d[x \mapsto \mathbb{E}[\![e]\!]^\sharp\, d].$$

A definition of $\mathbb{E}[\![e]\!]^\sharp$ is given later in Eq. (9) in Section 4.5. Non-deterministic assignment is defined as

$$[\![\mathsf{havoc}(V)]\!]^\sharp\, d \stackrel{\text{def}}{=} d[v \mapsto \top \mid v \in V].$$

Abstract semantics $[\![\mathsf{assume}(e)]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ in a non-relational domain is also presented and discussed later in Section 4.5.1.

Relational Domains. For example, relations between numeric variables are often abstracted using the following domains, where a_v ($v \in \mathcal{V}$) and b are numeric constants:

Octagon domain [Min01; Min06] infers a conjunction of constraints of the form $\pm v_1 \pm v_2 \leq b$, where $v_1, v_2 \in \mathcal{V}$. Interval bounds can be represented with $v_1 = v_2$. The octagon domain is classified as a *weakly relational domain* due to the limited nature of its constraints. This makes it relatively efficient and widely used [Bla+03; Cou+09].

Affine equalities domain [Kar76] infers a conjunction of constraints of the form $\sum_{v \in \mathcal{V}} a_v v = b$.

Polyhedra domain [CH78] infers a conjunction of constraints of the form $\sum_{v \in \mathcal{V}} a_v v \geq b$. This generalizes both affine equalities and octagons.

Non-relational domains are a subset of relational domains. For example, the interval domain is often also called the *box* domain when viewed relationally.

A *relational domain* $\mathbb{R}$ provides abstract semantics for primitives:

$$\begin{aligned}
[\![v \leftarrow e]\!]^\sharp &: \mathbb{R} \to \mathbb{R} && \text{(assignment)},\\
[\![\mathsf{havoc}(V)]\!]^\sharp &: \mathbb{R} \to \mathbb{R} && \text{(non-deterministic assignment)},\\
[\![\mathsf{assume}(e)]\!]^\sharp &: \mathbb{R} \to \mathbb{R} && \text{(assumption)}.
\end{aligned}$$

2.4.4. Constraint Systems

Homogeneous Constraint Systems. Let $\mathcal{X}$ be a set of constraint system *unknowns*[3] and $\mathbb{D}$ an abstract domain. A *constraint* for an unknown $x \in \mathcal{X}$ is of the form $\eta\, x \sqsupseteq f_x\, \eta$ with a left-hand side x and a right-hand side $f_x : (\mathcal{X} \to \mathbb{D}) \to \mathbb{D}$.

[3] Sometimes also called constraint system variables, but we refrain from that to avoid confusion with program variables.

A set of such constraints is called a *constraint system*. A mapping $\eta : \mathcal{X} \to \mathbb{D}$ is a *solution* of the constraint system if $\eta\, x \sqsupseteq f_x\, \eta$ holds for all constraints. Note that a constraint system may include for one unknown $x \in \mathcal{X}$ multiple constraints (with different right-hand sides), all of which must be satisfied by the solution. If desired, multiple constraints for an unknown can be combined into one by joining ($\sqcup$) their returned values.

Algorithms for solving such constraint systems are called *solvers*. The simplest solvers are Kleene iteration and round-robin (RR) iteration [Api14]. More efficient solvers, which make use of dependencies of right-hand sides, are worklist iteration [Api14] and top-down (TD) [SV21]. Some solvers (or their variants) are *local*, allowing finite portions of infinite constraint systems to be solved [Api14; Ama+16; SV21].

Example 1. In the simplest form, an analysis is realized by solving the following constraints:

$$\begin{aligned} \eta\,[\mathsf{st}_{\mathsf{main}}] &\sqsupseteq \mathsf{init}^\sharp, \\ \eta\,[v] &\sqsupseteq [\![[u], s]\!]^\sharp\, \eta \qquad\qquad \forall (u, s, v) \in \mathcal{A}, \end{aligned}$$

where the unknowns $[u]$ correspond to nodes $u \in \mathcal{N}$ and each analysis defines the corresponding right-hand sides:

- $\mathsf{init}^\sharp : \mathbb{D}$ defines the initial state. Technically, as defined above, a right-hand side should be a function of η. While $\mathsf{init}^\sharp$ itself is not, it can be viewed in the necessary form as
$$\mathsf{init}^\sharp = (\lambda_.\, \mathsf{init}^\sharp)\, \eta.$$
We use the implicit form for simplicity of presentation.
- $[\![[u], s]\!]^\sharp : (\mathcal{X} \to \mathbb{D}) \to \mathbb{D}$ defines the abstract semantics for statement $s \in \mathcal{S}$ at node $u \in \mathcal{N}$. When the analysis is *interprocedural*, meaning, only the main function is used and no other functions are called, then all control flow is described by the CFG, so one can define
$$[\![[u], s]\!]^\sharp\, \eta \stackrel{\text{def}}{=} [\![s]\!]^\sharp\, (\eta\,[u]).$$
Since $[\![s]\!]^\sharp$ is strict, the right-hand side $[\![[u], s]\!]^\sharp$ is also strict w.r.t. $\eta\,[u]$. □

Heterogeneous Constraint Systems. In the constraint systems defined above, every unknown $x \in \mathcal{X}$ takes values from the same abstract domain $\mathbb{D}$, making it *homogeneous*. This can be generalized to allow different unknowns to take values from different abstract domains, yielding a *heterogeneous* constraint system, i.e., unknown $x \in \mathcal{X}$ takes values from some abstract domain $\mathbb{D}_x$. Let $(x : \mathcal{X}) \to \mathbb{D}_x$ denote the set of dependently-typed functions taking an argument $x \in \mathcal{X}$ and returning a value specifically from $\mathbb{D}_x$.[4] A constraint for an unknown $x \in \mathcal{X}$ of the form $\eta\, x \sqsupseteq f_x\, \eta$ then has a right-hand side $f_x : ((y : \mathcal{X}) \to \mathbb{D}_y) \to \mathbb{D}_x$.

[4] In type theory, this is usually written as $\prod_{x:\mathcal{X}} \mathbb{D}_x$ instead.

Note that such a heterogeneous constraint system can be viewed as a homogeneous one by defining $\mathbb{D} \stackrel{\text{def}}{=} \bigcup_{x \in \mathcal{X}} \mathbb{D}_x$. Thus, heterogeneity does not yield additional expressiveness, but offers extra type safety.

2.4.5. Side-Effecting Constraint Systems

Seidl, Vene, and Müller-Olm [SVM03] introduced so-called *side effects* to constraint systems for combining flow-sensitive and flow-insensitive information into one framework. They allow the evaluation of right-hand sides of constraints (for flow-sensitive unknowns) to make additional contributions (called side effects) to other constraint system unknowns (for flow-insensitive invariants). Apinis, Seidl, and Vojdani [ASV12] further developed them for partially context-sensitive interprocedural analysis (see Example 3 below). Erhard et al. [Erh+24e] use side effects to analyze dynamic non-local control flow arising from the `setjmp`/`longjmp` constructs in C.

A *side-effecting constraint* for an unknown $x \in \mathcal{X}$ is of the form $(\eta, \eta\, x) \sqsupseteq f_x\, \eta$ with a right-hand side $f_x : (\mathcal{X} \to \mathbb{D}) \to (\mathcal{X} \rightharpoonup \mathbb{D}) \times \mathbb{D}$. The latter returns a collection of side effects to other unknowns in $\mathcal{X}$ together with a contribution to the left-hand side. A set of such constraints is called a *side-effecting constraint system*. A mapping $\eta : \mathcal{X} \to \mathbb{D}$ is called *solution* of the constraint system if for all constraints $(\eta, \eta\, x) \sqsupseteq f_x\, \eta$ where $(\eta', d) = f_x\, \eta$, both $\eta|_{\operatorname{dom} \eta'} \sqsupseteq \eta'$ (pointwise) and $\eta\, x \sqsupseteq d$ hold. That is, all side effects of the right-hand side and its contribution to the left-hand side are accounted for by η.

Solvers for side-effecting constraint systems can be constructed from solvers for normal constraint systems in a general (but suboptimal) manner [SVM03]. More efficient solvers, which rely on bespoke extensions, have been obtained from worklist iteration [SVM03], SLR [Api14] and TD [SV21]. All of these also happen to be local solvers.

Example 2. The constraint system from Example 1 can be reformulated as side-effecting as follows:

$$\begin{aligned} (\eta, \eta\, [\mathsf{st}_{\mathsf{main}}]) &\sqsupseteq \mathsf{init}^\sharp, \\ (\eta, \eta\, [v]) &\sqsupseteq [\![[u], s]\!]^\sharp\, \eta \qquad\qquad \forall (u, s, v) \in \mathcal{A}. \end{aligned}$$

The right-hand sides are extended to return side effects:

- $\mathsf{init}^\sharp : (\mathcal{X} \rightharpoonup \mathbb{D}) \times \mathbb{D}$ defines the initial state and initial side effects.
- $[\![[u], s]\!]^\sharp : (\mathcal{X} \to \mathbb{D}) \to (\mathcal{X} \rightharpoonup \mathbb{D}) \times \mathbb{D}$ defines the abstract semantics for statement $s \in \mathcal{S}$ at node $u \in \mathcal{N}$ along with side effects. When all control flow is described by the CFG, then there are no side effects, so one can define

  $$[\![[u], s]\!]^\sharp\, \eta \stackrel{\text{def}}{=} (\varnothing, [\![s]\!]^\sharp\, (\eta\, [u])).$$

  Since $[\![s]\!]^\sharp$ is strict, the right-hand side $[\![[u], s]\!]^\sharp$ is also strict w.r.t. $\eta\, [u]$. □

Example 3. Apinis, Seidl, and Vojdani [ASV12] show how side-effecting constraint systems are useful for expressing partially context-sensitive interprocedural analysis. Simplified to be context-insensitive, the idea is to define abstract semantics for function calls as follows:

$$\begin{aligned}[[[u], f()]]^\sharp\, \eta \stackrel{\text{def}}{=} & \textbf{ let } d = \eta\,[u] \textbf{ in} \\ & \textbf{let } \rho = \{[\mathrm{st}_f] \mapsto \text{enter } d\} \textbf{ in} \\ & \textbf{let } d' = \text{combine } d\,(\eta\,[\mathrm{ret}_f]) \textbf{ in} \\ & (\rho, d'),\end{aligned}$$

where:

- enter : $\mathbb{D} \to \mathbb{D}$ defines the initial state for the function from the state of the caller. enter is strict.
- combine : $\mathbb{D} \to \mathbb{D} \to \mathbb{D}$ combines the state of the caller with the final state of the function. combine is strict w.r.t. both arguments.

Note that in this simplified setting the side effects are easily avoidable,[5] but it is less so with (partial) contexts and dynamic function calls (via function pointers) [ASV12]. □

Analyses in later sections will be realized via similar constraint systems, but will further refine the set of unknowns and define constraints involving the refining information, e.g.,

- Section 3.5.2 separates unknowns by thread ID and uses side effects for initial states of created threads.
- Chapter 5 separates unknowns by lockset and uses side effects to auxiliary unknowns for inter-thread data flow.

Similarly to non–side-effecting constraint systems, these can also be made heterogeneous such that a right-hand side for an unknown $x \in \mathcal{X}$ is

$$f_x : ((y : \mathcal{X}) \to \mathbb{D}_y) \to ((z : \mathcal{X}) \rightharpoonup \mathbb{D}_z) \times \mathbb{D}_x.$$

Remark 1 (Narrowing side effects). Traditionally, solvers for side-effecting constraint systems have treated side effects as irreversibly accumulating, using widening for termination but no narrowing to recover precision. For example, if an imprecise side effect from a widening iteration of a flow-sensitive unknown is proceeded by a precise side effect from narrowing iteration, then the value of the side-effected unknown would remain imprecise. Therefore, the computed flow-insensitive invariants could very quickly become very imprecise.

Only recently, Stemmler et al. [Ste+25] have proposed sound narrowing of flow-insensitive unknowns, breaking the long-standing limitation. □

[5] By replacing the side effects with an additional family of constraints:

$$(\eta, \eta\,[\mathrm{st}_f]) \sqsupseteq \text{enter}\,(\eta\,[u]) \qquad \forall (u, f(), _) \in \mathcal{A}.$$

Unlike the side-effecting version, these constraints are very non-local: the entry point of each function is constrained by edges of the entire program.

2.5. Witnesses

We now turn to different kinds of verification witnesses and their formats. After a brief overview of violation witnesses (Section 2.5.1), the rest of the thesis only deals with correctness witnesses (Section 2.5.2). We then describe an existing format for correctness witnesses (Section 2.5.3) and outline numerous applications (Section 2.5.4).

In this thesis, we consider witnesses for *safety* properties, such as assertion safety, memory safety, overflows, data races, etc. Such properties are described in more detail later in Section 2.6.1.

2.5.1. Violation

The canonical witness for a safety property violation is a *counterexample*: a sequence of steps from the beginning of the program up to the point of violation, together with inputs from the environment (e.g., randomly picked numbers, user input from the terminal or a file). In fact, the later are sufficient: by replacing the environment interaction with a corresponding *test harness*, the program can simply be executed to obtain the sequence of steps, as long as the violation is observable at runtime. To account for different levels of detail and offer more flexibility, more general *violation witnesses* have been proposed.

Automaton-Based. Beyer et al. [Bey+15] propose a GraphML format for describing a witness automaton which can be run along with a program. It does not require the violation witness to be a single counterexample sequence (with assumptions as C expressions), but allows describing different paths which could lead to a violation. They may skip irrelevant program steps by stuttering in the witness automaton. Moreover, they are not required to indicate the program location of the violation, but may simply be used to restrict the search space of a validator to certain execution prefixes.

Location-Based. Strejček [Str22] has observed that the control-flow semantics of witness automata are ambiguous, impairing interoperability. This led Ayaziová et al. [Aya+24] to propose a YAML format for describing a violation sequence. Waypoints in the sequence are directly associated with program locations to avoid incompatibilities arising from differing control-flow representations in tools. The sequence may still skip program steps, but it must end at the violating location, not earlier.

2.5.2. Correctness

The notion of a canonical witness for safety property satisfaction is less clear. Beyer and Strejček [BS22b] merely say that "[correctness witnesses] should contain some hints for the proof of program correctness". Nevertheless, the standard approach is to use (loop) invariants a la Hoare [Hoa69].

Automaton-Based. Beyer et al. [Bey+16] extend automaton-based violation witnesses and their GraphML format also for correctness:

1. Correctness witness automata cannot restrict the state space when run along with a program.
2. They can provide invariants at each automaton state.

Conceptually, this allows for a great deal of flexibility because different invariants may be provided for the same program location depending on how it is reached.

Location-Based. As for violation witnesses, the observations of Strejček [Str22] on the ambiguous control-flow semantics of witness automata also apply here. Furthermore, invariants per program location are often sufficient in practice [Bey+16; BSU22]. Thus, Ayaziová et al. [Aya+24] also propose a YAML format for describing correctness through per-location invariants. In this work, we focus on such location-based correctness witnesses. We first give the underlying formal definitions and then look at their textual YAML realization in the following section.

Formally, we consider CFG nodes ($\mathcal{N}$) as program locations. Invariants from some language $\mathcal{I}$, which we do not fix, are used to specify properties of the program executions reaching a particular program location. We assume that there is a trivial invariant $e_{\mathsf{true}} \in \mathcal{I}$ which always holds. Since the goal is to exchange invariants between tools, the choice of an invariant language involves a trade-off:

- Beyer et al. [Bey+16] use boolean-valued side-effect–free C expressions for their invariants. The chief advantage is its conceptual simplicity: the semantics of such assertions is well known, and analyzers already come with the necessary facilities to manipulate these expressions, as they appear in the analyzed program. In C expressions, pointers allow exchanging information also about the heap between verifiers. Nevertheless, the expressivity of such invariants is limited, especially regarding more complex data structures.
- ACSL [Bau+23] has more expressive power than plain C expressions by offering quantification, memory predicates, etc. On the downside, considerably fewer analyzers support it, limiting exchange possibilities.
- Custom invariant languages can be arbitrarily expressive, at the cost of restricting exchange to even fewer tools.

With these notions, we introduce the definitions of a witness, its validation, and witness-guided verification.

Definition 10 (Witness)**.** A *witness* for a safety property Φ of a program P is a tuple (W, P, Φ), where $W : \mathcal{N} \to \mathcal{I}$ is a total mapping from the program locations of P to invariants from $\mathcal{I}$. ⌟

The textual format in which witnesses are exchanged is not required to provide invariants for all program locations — we implicitly assume that if the witness contains no information for some program location, then this location is mapped to e_{true}. If the invariant language contains a contradictory expression $e_{\mathsf{false}} \in \mathcal{I}$ which never holds, then it can be used to convey unreachability of a program location. This

notion of a witness is generic and can be instantiated to different programming and invariant languages. For our examples, we use an invariant language of arithmetic and boolean expressions, which is enriched with pointers in Chapter 4.

Definition 11 (Valid witness)**.** A witness (W, P, Φ) is *valid* if

1. P satisfies the property Φ;
2. whenever the execution of P reaches the location $n \in \mathcal{N}$, the invariant $W\,n$ holds. ⌟

Given this definition, we can distinguish three kinds of tools:

Witness validator attempts to prove that a witness is valid; specifically, it tries to recreate the proof that the program satisfies the property Φ, and checks that the witness makes only true claims about the program.

Witness-guided verifier uses the witness as guidance towards the verification of Φ. A sound verifier can perform this task without assuming the witness invariants to be true; therefore, it qualifies as a sound validator of the first condition. It may additionally verify the invariants in W to perform full witness validation.

Witness-only validator only checks that the witness invariants hold at their corresponding locations, while ignoring whether the program satisfies the property Φ. Hence, it is a sound validator of just the second condition.

2.5.3. YAML Format

We describe here the YAML format for correctness witnesses as proposed by Ayaziová et al. [Aya+24] and illustrate them by a simple example.

Syntax. A YAML correctness witness is a sequence of a single *entry* object with the following fields:

`entry_type`. The constant string `invariant_set`.

`metadata`. An object which includes information about the tool that produced the witness and the verification task. For brevity, we omit here the details which are given by Ayaziová et al. [Aya+24, Table 1].

`content`. A sequence of *invariant* objects with a single field:

`invariant`. An object with the following fields:

`type`. Either the string `location_invariant` or `loop_invariant`.

`location`. An object with the following fields:

`file_name`. A string indicating the file of the invariant.

`line`. An 1-indexed integer indicating the line number of the invariant in the file.

`column`. An optional 1-indexed integer indicating the column number of the invariant in the line. If missing, then it defaults to the first suitable column on the line.

`function`. An optional string indicating the name of the function containing the invariant. It is intended to improve human-readability of the witness.

`format`. The constant string `c_expression`.

`value`. A string of the invariant expression.

Semantics. The meaning of invariants depends on their type:

`location_invariant`. A location invariant must be located at the first character of a statement or a local declaration. It must always hold immediately before evaluating the statement or the declaration.

`loop_invariant`. A loop invariant must be located at the first character of a loop keyword (`while`, `for` or `do`). It must always hold immediately before evaluating the loop condition.

Example 4. Consider the program in Listing 1a and the YAML correctness witness in Listing 1b. The three *valid* invariants in the witness illustrate various uses of

Listing 1. YAML correctness witness example

(a) Program

```
1 void main() {
2   int i = 0;
3   while (i < 100) {
4     i++;
5   }
6 }
```

(b) YAML witness

```
1  - entry_type: invariant_set
2    metadata: # omitted
3    content:
4    - invariant:
5        type: loop_invariant
6        location:
7          file_name: listing-1a.c
8          line: 3
9        value: 0 <= i && i <= 100
10       format: c_expression
11   - invariant:
12       type: location_invariant
13       location:
14         file_name: listing-1a.c
15         line: 3
16       value: i == 0
17       format: c_expression
18   - invariant:
19       type: location_invariant
20       location:
21         file_name: listing-1a.c
22         line: 4
23       value: 0 <= i && i <= 99
24       format: c_expression
```

loop and location invariants:

1. The loop invariant is located at the `while` keyword. This invariant holds (a) before the loop itself, (b) before each loop iteration, (c) after each loop iteration and (d) after the loop itself.
2. The location invariant at the same location, however, only holds before the loop itself. This highlights the difference between loop and location invariants.
3. The other location invariant within the loop body illustrates an invariant at an arbitrary location, not necessarily corresponding to a loop. □

Unlike the GraphML format, the YAML format has been designed with extensibility in mind. As initially proposed (and described above), a correctness witness consists of a single entry, but only because a single `entry_type` is proposed. If new `entry_types` are proposed, along with corresponding `contents` and semantics, then correctness witnesses consisting of multiple entries are naturally possible. This is precisely what we end up doing in Section 6.6.3 for one kind of proposed thread-modular correctness witnesses. Extensibility of the YAML format has also been explored by others who have proposed witnesses for function contracts [Hei+25], termination [BJL25], non-termination [Ádá+25] and weak memory models [BL24; BL25].

2.5.4. Applications

Witnesses focus on the exchange of internal analysis results, which can be used in various ways. For exchanging external results, there are other means, such as the Static Analysis Results Interchange Format (SARIF) [OAS20].

Verification Competitions. Witnesses form a cornerstone of the annual software verification competition SV-COMP [BS25] (described in the following section). To score points in the competition, it is not enough for a competing verifier to just output the expected binary verdict (program violates/satisfies property). The verifier may be faulty and may have coincidentally produced the right verdict but for the wrong reasons.[6]

To make the results more trustworthy, the verifier must additionally output a corresponding witness to explain its reasoning for the verdict. The witness is then validated by a set of validators to confirm that the reasoning in the witness is correct and yields the claimed verdict.

Precision Comparison. Correctness witnesses provide a general method for comparing the precision of different analyses, even in different tools. One analysis generates a witness with as many and as precise invariants it has inferred. Another analysis validates this witness (without witness-guided verification) to determine

[6] As an extreme, consider a verifier with constant or random output. Depending on the balance of programs violating/satisfying the property, such a verifier may appear to be correct in a large amount of cases with no basis in reality (i.e., without even looking at the program!).

how many of those invariants it can confirm. By using invariants as the intermediate format, this allows to compare the precision of analyses with different abstract domains or even approaches where different internal representations cannot be directly compared. This technique is also exploited in this thesis in Section 5.7.2.

Conceptually, it is similar to instrumenting a program with assertions using one analysis and trying to verify the instrumented program using another. However, storing the invariants in a witness is less intrusive than adding assertions into the program itself, e.g., because locations of the original program statements do not change when including the invariants.

Cooperative Verification. Witnesses have played a key role in the emergence of cooperative verification [BK22; HW22], where independent verifiers collaborate by exchanging witnesses [BW20]. As opposed to witness validation, one verifier may assume the witness from another to be valid and use its findings. Thus, witnesses in a common format can be used to transfer knowledge between distinct tools.

One verifier may discharge certain difficult cases to another verifier that is better suited for certain verification problems. The communication via witnesses may even be two-way: two verifiers delegate some tasks to each other, communicate back their results, and jointly refine each other's results. Furthermore, cooperation can be generalized to more than two tools, all talking in a common witness language.

Incremental Verification. Verification of large systems may be time- and resource-intensive. The core idea of incremental verification is to store verification results and reuse them later. In particular, when a small portion of the system under verification is modified, it is desirable to only reanalyze the modified parts (and their dependents), while reusing previous verification results for the unmodified parts. The goal is to reduce time and resources for reverifying the modified system.

As an example, GOBLINT supports incremental verification and provides two different mechanisms for storing previous results. For interactive analysis, it has a long-running server mode, which holds previous results in random-access memory [Erh+24c]. For non-interactive analysis, it stores previous results on-disk, using OCAML's built-in binary marshaling format and library [SEV20]. However, the latter is problematic for multiple reasons:

1. The marshaled data is specific to a version of GOBLINT: if internal data structures of the verifier are changed, then it is unable to unmarshal old data and likely crashes.
2. The verification results in a binary format are difficult to store along the source code: committing (potentially large) binary blobs into a GIT repository is highly discouraged.
3. The stored verification results are not directly inspectable by humans.

Using witnesses in a human-readable format for incremental verification results storage solves all these problems. On the downside, it requires the witness format to be expressive enough to represent the verifier's internal results with sufficient precision to be useful for reuse.

2.6. SV-COMP

The Competition on Software Verification (SV-COMP) is an annual evaluation and comparison of the state of the art in automated software verification [BS25]. So far there have been 14 editions of the competition from SV-COMP 2012 to SV-COMP 2025. It has separate tracks for programs in two different programming languages: C (since 2012) and Java (since 2019). In the following, we only focus on the C track, which is more substantial, both in terms of benchmark programs and competing tools. The latest and greatest edition included 33,353 C benchmarks and 53 C verifiers. Since 2023, the competition also includes a secondary track for witness validators, of which there were 15 for C benchmarks in 2025.

This section is based on information found on the competition website[7] and the competition reports [Bey21; Bey22a; Bey23; Bey24; BS25].

2.6.1. Properties

For C programs, the competition is concerned with the verification of various *properties* (aka specifications). We describe all of them at a high level (somewhat informally):

`unreach-call`. The special function `reach_error` is never called. It usually encodes assertion safety (i.e., no assertion is violated) via the following helper function:

```c
void __VERIFIER_assert(int cond) {
  if (!cond)
    reach_error();
}
```

Assertions using this function always hold if and only if `reach_error` is never called via this definition.

Dually, (un)reachability can be encoded as assertion safety by defining:

```c
void reach_error() {
  assert(0);
}
```

The assertion in this function (vacuously) holds if and only if `reach_error` is never called.

`valid-memsafety`. Consists of three subproperties that are checked simultaneously:

- **`valid-deref`.** All pointer dereferences are valid, i.e., the dereferenced pointer points into a valid block of memory (as opposed to out-of-bounds). Invalid dereferences are undefined behavior in C.
- **`valid-free`.** All memory deallocations are valid, i.e., the deallocated pointer points to dynamically allocated memory and has not already

[7] `https://sv-comp.sosy-lab.org`

been deallocated (as opposed to double-free). Invalid deallocations are undefined behavior in C.

`valid-memtrack`. All still-allocated memory is reachable from roots (global variables and local variables on the call stack) via pointer dereferences. Unreachable allocated memory has definitely leaked.

`valid-memcleanup`. All allocated memory is deallocated before the program terminates, i.e., no memory leaks.

`no-overflow`. No overflow occurs in *signed* integer operations: addition, subtraction, multiplication, negation (of minimum value), division (by -1) and bitwise left shift. Signed integer overflows are undefined behavior before C23.

`no-data-race`. No data race occurs, i.e., there are no multiple concurrent accesses to the same memory location such that at least one of them is a write and at least one of them is non-atomic. Data races are undefined behavior since C11.

`termination`. The program always terminates.

Out of these properties, `termination` is a *liveness* property (good things do happen) and the others are *safety* properties (bad things do not happen) [Lam77]. The `unreach-call` and `termination` properties are general, whereas the other properties are specific to certain programming errors in C, which often constitute undefined behavior under the C standard. The presence of undefined behavior in a program voids *all* guarantees on its compilation and execution.[8] Regehr [Reg10] provides a good introduction and a comprehensive discussion on the topic.

Many properties can be encoded as others. It is particularly common to reduce to assertion safety [Bey+25], asserting either programming language expressions or something more expressive, analogously to correctness witness invariants (Section 2.5.2). Tools are free to do so internally, but it is best to not hard-code such encodings into the programs themselves as this prevents property-specific algorithm improvements.

2.6.2. Tasks

Verification Track. A verification task consists of the following:

1. A program (given to verifier).
2. A data model (given to verifier): ILP32 or LP64.[9]
3. A property (given to verifier).
4. An expected verdict for the property (*not* given to verifier): `true` or `false`.

[8] "Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message)." [ISO11, Section 3.4.3]

[9] Size of pointers and the `long` data type is 4 bytes for ILP32 and 8 bytes for LP64.

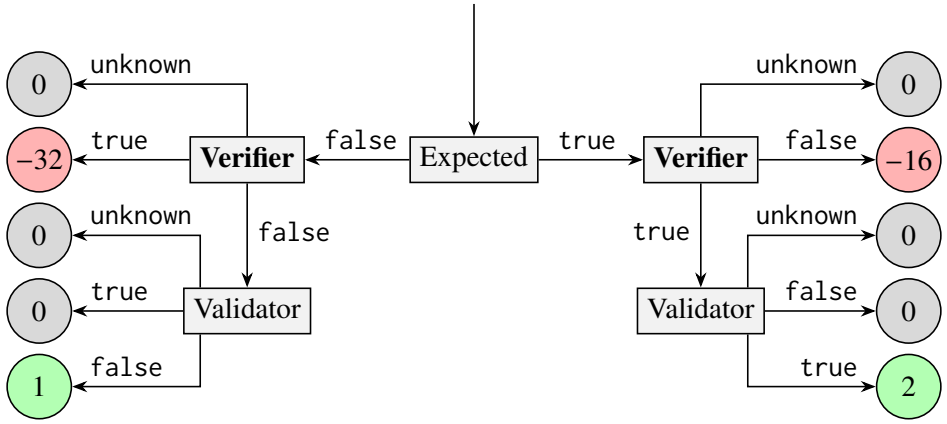


Figure 3. Scoring schema for the SV-COMP verification track as a decision tree: decisions are based on the expected verdict of the task, and the verdicts outputted by the verifier and the validator; score for the verifier is shown in the corresponding circular leaf

For each task, a verifier is executed with a CPU time limit of 900 s and a RAM limit of 16 GB. Within these limits, it should output:

1. A verdict for the property: `true`, `false` or `unknown`. This needs to match the expected verdict to score (positive) points, while `unknown` gives zero points, otherwise negative points are given.
2. A witness corresponding to the verdict (if not `unknown`). This needs to be confirmed by some validator to score (positive) points. However, in some categories, correctness witnesses are not required.

The workflow (with one validator) is depicted in Fig. 1 on page 18. Figure 3 shows the complete scoring schema, exactly as used since SV-COMP 2021. Since verification is generally considered more difficult than bug finding, correct proofs (2 points) are rewarded more than true alarms (1 point). Incorrect verdicts do not disqualify a verifier but come with severe penalties, with wrong proofs (−32 points) being punished more than false alarms (−16 points).

One program may be part of multiple tasks with different properties.[10] Each program contains at most one kind of undefined behavior, which corresponds to the expected property violation.

Validation Track. A validation task consists of the following:

1. A program (given to validator).
2. A data model (given to validator): ILP32 or LP64.

[10] Conceptually, one program could also be part of multiple tasks with different data models. For example, the expected verdict for the `no-overflow` property could differ between ILP32 and LP64, depending on the size of the `long` data type. However, no tasks like this currently exist in the suite.

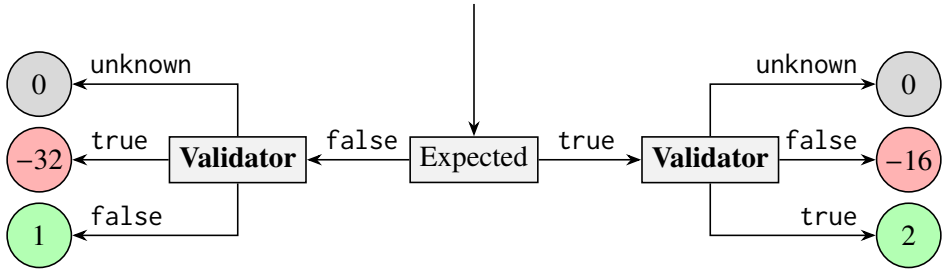


Figure 4. Scoring schema for the SV-COMP validation track as a decision tree: decisions are based on the expected verdict of the task and the verdict outputted by the validator; score for the validator is shown in the corresponding circular leaf

3. A property (given to validator).
4. A witness produced by some verifier (given to the validator).
5. An expected verdict for the witness (*not* given to validator): `true` or `false`.

For each task, a validator is executed with a CPU time limit of 90 s for violation witnesses or 900 s for correctness witnesses, and a RAM limit of 7 GB. The lower limits (compared to verification) are intended to motivate content-effort dependence (Section 1.2). Within these limits, it should output a verdict for the witness: `true`, `false` or `unknown`. This needs to match the expected verdict to score (positive) points, while `unknown` gives zero points, otherwise negative points are given. Figure 4 shows the complete scoring schema, exactly as used since SV-COMP 2024. It is analogous to the verification track, except there is no validation of the validators themselves (yet).[11]

Unlike in the verification track, where expected verdicts are explicit in the benchmark suite, they are not directly available in the validation track, where the witnesses come from verification track tools. In the absence of a better method, expected verdicts for witnesses are determined by a majority vote of the validators: at least 75% of validators must yield the same verdict [BS25]. Witnesses without such a majority are excluded from the validation track scoring.

The validation track in SV-COMP 2023 scored participants as witness-guided verifiers, which only confirm the first condition of Definition 11 (p. 37) [Bey23]. The current scoring schema, adopted in SV-COMP 2024, corresponds to full witness validation [Bey24]. Since SV-COMP 2025, the benchmark suite contains a small set of handcrafted witnesses with explicit expected verdicts that are additionally included in the validation track [BS25].

[11] "Who watches the watchmen?" For the case of violation witnesses, the seminal work of Beyer et al. [Bey+15] proposes *stepwise testification* as the lengthening of the validation chain.

2.6.3. Categories

All tasks are organized into categories based on their property, their origin and some features of the program. For example, all multi-threaded programs are in various *ConcurrencySafety* categories and all large programs (originating from real-world code) are in various *SoftwareSystems* categories. Other distinguishing features include the use of arrays, heap, recursion, etc. in the program.

Groups of categories form metacategories, which consist of subcategories. For example, the *ReachSafety* metacategory, which includes tasks with the `unreach-call` property, contains subcategories such as *ReachSafety-Arrays*, *ReachSafety-Heap*, *ReachSafety-Recursive*, etc. At the top of the three-level hierarchy is the *Overall* category. Scores for metacategories are weighted sums of scores for the subcategories such that each subcategory contributes equally regardless of its task count. See the SV-COMP competition report [BS25] for the full list and hierarchy of categories.

SV-COMP allows tools to *opt out* from some categories: the tool is still executed on the tasks of the category, but its results specifically for that category are not reported. Opt-outs are used when the tool wants to compete in a metacategory but not in its subcategory, for example, because it does not support the tasks in the subcategory.

3. GOBLINT ANALYZER

GOBLINT is a static analysis framework for C programs, based on abstract interpretation (Section 2.4). We dedicate an entire chapter to GOBLINT because it is where all the contributions of this thesis have been implemented.

First, we provide an up-to-date overview of its architecture (Section 3.2) and its abstract-interpretation–based analyses (Section 3.3). Second, we reminisce about GOBLINT participating in SV-COMP (Section 3.4), describing its progress and results over five years. Third, we highlight how its data race analysis has benefited from other contributions of this thesis (Section 3.5).

Parts of each have been described at different times in various papers and other theses. This chapter offers a combined snapshot which is not found elsewhere. Moreover, it documents some valuable institutional knowledge about the framework.

3.1. Introduction

In 2006, Vojdani [Voj06] created GOBLIN (sic!) for analyzing data races in multi-threaded C programs using the PTHREADS library. In 2007, Vojdani and Vene [VV07; VV09] introduced it to the world as GOBLINT.[12] In years to come, the focus was on improving data race analysis of the Linux kernel [Voj10; Voj+16]. Afterward, GOBLINT was adapted to analyze data races in automotive AUTOSAR/OSEK programs [Sch14] and starvation in avionic ARINC 653 programs [SV17]. After these detours, in recent years, the primary target for GOBLINT have again been C programs using PTHREADS.

GOBLINT is a joint project of University of Tartu (Laboratory for Software Science) and Technische Universität München (Chair of Formal Languages, Compiler Construction, Software Construction). It is open source and available under the MIT licence. The development of GOBLINT takes place on GitHub,[13] while related publications are listed on its website.[14]

GOBLINT is implemented in ~83,000 physical lines of code (LoC) of OCAML, with Fig. 5 showing the trend over time. Its primary dependencies are:

- GCC for preprocessing C code,
- an updated fork[15] of CIL [Nec+02] as parser frontend for C code,
- APRON [JM09] for some numeric relational domains.

No other major libraries or external tools are required. The remaining dependencies are standard utilities in the OCAML ecosystem: ZARITH (for arbitrary-precision

[12] The original name and the subsequent rename have been explained by Vojdani [Voj10, Acknowledgments]. The same goes for the discrepancy/redundancy between 2007 and 2009 [Voj10, Section 1.4].

[13] `https://github.com/goblint/analyzer`

[14] `https://goblint.in.tum.de` or `https://goblint.cs.ut.ee`

[15] `https://github.com/goblint/cil`

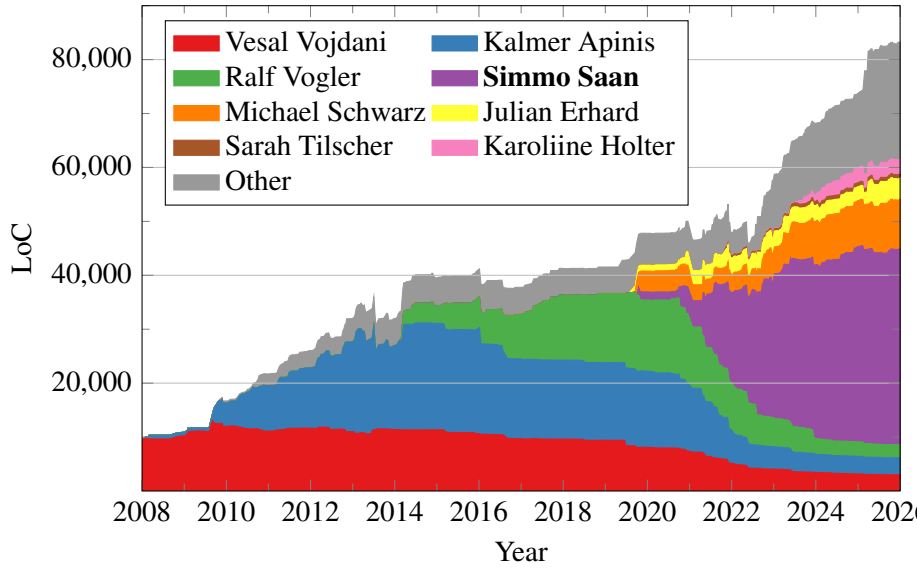


Figure 5. Physical lines of code (LoC) of OCaml in Goblint over time by author

integers), Yojson (for JSON), Batteries (for some standard library extensions), etc.

3.2. Software Architecture

Apinis [Api14] has described the modular architecture of Goblint, which fundamentally has not changed. Nevertheless, there have been additions, justifying the following refreshed overview.

Frontend. Input C code is preprocessed using cpp from GCC to resolve `#include`s, `#define`s, etc. This is necessary because the CIL parser requires preprocessed code. In case of multiple input files, CIL is also used to perform source-level linking. A CFG is constructed for every function defined in the program.

Analyses. Analyses are defined through their abstract domains and transfer functions, which can communicate with other analyses using predefined queries and events:

Queries allow analyses to use information already computed by other analyses (a pull mechanism). For example, there are queries for evaluating an integer expression or getting the current lockset. Multiple analyses may answer a single query, in which case their answers refine each other by meet ($\sqcap$) to obtain maximal precision.

Queries were added to the framework by Apinis [Api14]. Since then, queries have been optimized by caching their answers during a single transfer function evaluation.

Events allow analyses to inform other analyses of certain situations of interest (a push mechanism). For example, there are events for locking a mutex or a local variable escaping its scope. They provide more efficient means for detecting changes in information computed by other analyses without having to perform an excessive amount of queries.

Events were introduced for the work of Chapter 5.

Configuration. The combination of analyses and their domains can be configured at runtime. The configuration is represented as a large JSON object which is now described by a JSON schema. The default configuration can be altered by providing JSON configuration files, describing only modified options, or command-line arguments, which modify individual options. Section 3.4.3 describes how a configuration can be automatically selected for SV-COMP.

Constraint System. The combined analyses together with the CFGs of the functions in the program yield a side-effecting constraint system (Section 2.4.5). The constraint system is heterogeneous, using different abstract domains for the flow-sensitive and flow-insensitive unknowns. The latter are split further, allowing each activated analysis to use its own unknowns with a corresponding domain. However, the implementation is not dependently typed: it offers a type-safe interface to solvers and analyses, but uses runtime conversions at some intermediate layers. The constraint system presented to the solver is lifted to be homogeneous.

Constraint Solving. Such a constraint system is solved using a *local* generic solver, which yields a (post-)solution for just the reachable program locations and contexts (see Section 2.4.4). This is where the majority of time is spent: the solver invokes the right-hand side functions of the constraint system, which in turn call transfer functions for individual analyses. While a number of solvers are available, the improved top-down solver TD3 [SV21] is employed by default. Its solving process is not strictly separated into widening and narrowing phases, but these may be intertwined instead [Ama+16]. GOBLINT itself is single-threaded, despite its strong focus on analyzing multi-threaded code. There is ongoing work to parallelize the top-down solver.

Post-processing. The solution is post-processed, which performs the following in a single pass called *post-solving* [Erh+24c]:

Outputting warnings. Warnings are reported only at the end based on the computed solution, as widening during the fixpoint computation might lead to spurious property violations, which later disappear due to narrowing.

Verifying solution. Every constraint is evaluated once more to check that what the solver returned is indeed a solution to the constraint system. This serves a sanity check for the solver to catch bugs in its implementation. It can also reveal issues with abstract domains or transfer functions.

Pruning solution. Intermediate steps of fixpoint computation may have analyzed functions in contexts which no longer arise at the fixpoint [MNY14].

Post-processing determines the set of still reachable unknowns and prunes the rest from the solution. This is a simple form of abstract garbage collection; more complex versions have been studied by Stemmler et al. [Ste+25].

Marshaling solution. The (pruned) solution is marshaled to disk for incremental reanalysis (if enabled).

Then, additional post-processing may be performed for different output formats (e.g., HTML), witness generation (Section 4.6.3) or validation (Section 4.6.2).

3.3. Verification Approach

The interprocedural analysis of GOBLINT is partially context-sensitive using the tabulating functional approach, as opposed to the call string approach, inlining or summarization. Within functions, the analysis is flow- and path-sensitive, for example, w.r.t. the lockset. The analysis of concurrent programs is thread-modular: each thread is analyzed in isolation and their interferences are communicated to each other abstractly. This scales well to larger programs with many threads, as opposed to analyzing thread interleavings. Both the interprocedural and the thread-modular aspects are encoded using the side-effecting constraint systems described above.

The following subsections describe the various abstract-interpretation–based analyses, organized by SV-COMP property (Section 2.6.1). This provides a self-contained overview, combined and expanded from the descriptions originating from GOBLINT's tool papers [Saa+21; Saa+23b; Saa+24b].

3.3.1. Values and Reachability

The analysis of values of program variables is the foundation of abstract interpretation and static analysis:

1. Assertion safety is directly determined through value analysis by abstractly evaluating the assertions.
2. Pointer analysis is key to sound analysis of C code, especially to be able to handle dynamic function calls through function pointers.
3. Reachability is indirectly determined through value analysis, either as dead conditional branches (where conditions have constant value) or as dead functions (which are never called, even via pointers).
4. Overflow, memory safety and termination analyses require the bounds of integer expressions.
5. Data race, memory safety and memory leak analyses rely on the analysis of pointers to check for aliasing.

We now describe the various different value analyses provided by GOBLINT. How they are lifted to the concurrent case is deferred to Chapter 5.

Base Analysis. This non-relational analysis combines various abstract domains depending on the type of the value:

Integers. A configurable product (optionally reduced) of the following non-relational integer domains:

Constants and exclusion sets. In addition to standard definite integer constants, this domain can also represent any integer except a finite set of integer constants. The latter are motivated by C, where every non-0 integer is considered true. An exclusion set of just the constant 0 represents such values exactly.

Intervals. Standard integer intervals using arbitrary-precision integers for bounds [CC77].

Enumerator sets. Sets of integer constants arising from `enum` values in C. Unbounded growth of sets is prevented using very imprecise (arithmetic, etc.) operators, which usually are not used on enumerators.

Congruences. Standard integer congruences [Gra89].

Interval sets. Sets of non-overlapping non-adjacent intervals with a specific widening operator to prevent unbounded growth of set cardinality. Overlapping or adjacent intervals are joined, maintaining a normal form.

Floats. Standard floating-point intervals using native `float` or `double` for bounds [Min04, Section 4].

Pointers. Flow-sensitive points-to sets where each abstract address consists of a base variable and an abstract offset. The base variable may either be a program variable or a synthetic heap variable corresponding to an allocation-site (possibly together with the allocating thread ID and an allocation counter). A fixed number (by default 0) of the first heap variables at an allocation-site are unique copies, the rest are abstracted by a non-unique copy. The abstract offset is a sequence of struct field accesses and array indexings (using the above integer domains to abstract the index) on the base variable.

Structs. Configured to be one of the following:

Non-relational. Each field is independently abstracted by the corresponding value domain (recursively).

Relational. In some cases relations between struct fields are important, e.g., between an enumerator and a function pointer. Some set-based liftings are available to handle such cases.

Unions. A union is abstracted by a flat domain of its possible fields and a corresponding value domain (recursively).

Arrays. Configured to be one of the following:

Collapsed. A single abstract value (recursively) is used to abstract all elements of the array.

Unrolled. A fixed number of the first array elements are abstracted separately, the rest are collapsed.

Partitioned. An array is partitioned by a symbolic expression into three parts: elements before the symbolic index, the element at the symbolic index and elements after the symbolic index. Each of the three is abstracted separately. This abstraction matches a common pattern where the array is processed left-to-right, distinguishing already processed elements from the unprocessed ones.

Thread IDs. Sets of abstract thread IDs, which are described in Section 3.5.2.

Chapter 5 describes thread-modular liftings for global variables.

Apron Analysis. This relational analysis applies relational domains from the Apron library [JM09] to integers (but not floats). Supported domains are box (i.e., interval), octagon, affine equalities and polyhedra. Section 5.5 describes a thread-modular lifting for global variables.

Variable Equalities Analysis. This relational analysis infers simple symbolic integer and pointer equalities, which are used by the symbolic lockset analysis (Section 3.3.3). Equalities between local lvalues and addresses of lvalues are inferred from assignments only.

3.3.2. Overflows

The analysis of overflows completely piggybacks on the value analysis of integers. If the numeric bounds (as arbitrary-precision integers) of an expression fit the bounds of the resulting integer type, then no overflow can occur in this calculation. The checks are performed in the base analysis using the interval or the interval set domain, however, they can benefit from relational analyses using queries.

3.3.3. Data Races

The main goal of Goblint has been data race detection and it has various specialized analyses for this purpose [Voj+16]:

- For simple (coarse-grained) lock-based idioms, it employs a path-sensitive lockset analysis.
- For more complex (fine-grained) lock-based idioms, it employs a symbolic lockset analysis [SVV09], which correlates access expressions with locking expressions using Herbrand address equalities. Unlike the theoretical development, the implementation can only correlate one symbolic index and that correlation must be a direct equality (i.e., not affine).
- For medium-grained lock-based idioms, it employs a region analysis [SV09], which partitions memory accesses into disjoint heap regions.

Besides locking, race freedom can also be ensured via other means, e.g., temporally. Section 3.5 generalizes the race analysis of Goblint and proposes numerous precision improvements, which did not fit the existing framework [Voj+16].

3.3.4. Memory Safety

The memory safety analysis is concerned with invalid dereferences and invalid deallocations. In addition to identifying possible `NULL` pointer dereferences by simply inspecting the points-to sets, the following analyses are performed.

Out-of-Bounds (OOB). Recall that each abstract address in GOBLINT is a pair of a base variable (including special heap variables) and an abstract offset. The bounds check for its dereferencing first determines the size of the memory block corresponding to the base variable. For heap variables, the allocated size is queried from the base analysis. Then, abstract comparisons using activated integer domains determine whether an access must be in bounds or may be out of them.

Use-After-Free (UAF). The analysis flow-sensitively maintains a set of possibly already deallocated heap variables and warns when they are accessed. In multi-threaded programs, a heap variable could also be deallocated by a different thread. To account for this, the analysis additionally flow-insensitively maintains for each heap variable the set of threads (by their abstract thread IDs) that may deallocate it. Combined with may-happen-in-parallel (MHP) information, GOBLINT can check whether the access definitely happens before deallocation by another thread or not.

Invalid Deallocations. Double-free bugs are just UAF bugs where the use is another deallocation. Thus, the same UAF analysis also identifies those.

3.3.5. Memory Leaks

The analysis of memory leaks tracks a set of possibly (still) allocated heap variables path- and context-sensitively. Deallocation removes a variable from the set only if the deallocated pointer unambiguously points to a single concrete memory location, i.e., a unique heap variable. Since non-unique heap variables represent an unknown number of concrete memory locations, the allocation-site counters for unique heap variables are key to being able to prove leak freedom. A memory leak is reported when the set of possibly still allocated memory is non-empty at program exit (including early termination by C functions like `exit`).

For multi-threaded programs, the thread-modular analysis checks all of the following conditions to prove leak freedom:

1. all threads have terminated at the end of `main`;
2. functions like `exit`, which cause early termination, are not called;
3. each thread has deallocated all the memory it allocated when it terminates.

Such an analysis is sound, but imprecise: for example, it cannot prove leak freedom of programs where some memory is allocated in one thread but deallocated in another.

3.3.6. Termination

The termination analysis accounts for multiple sources of possible non-termination.

Loops. Each syntactic loop is instrumented with a synthetic counter variable, which is initialized to 0 before the loop and incremented by 1 in every iteration. A loop is terminating if existing value analyses prove that the counter variable is bounded. Relational value analysis is particularly useful here as it can infer relationships between the counter variable and program variables, carrying over bounds from one to the other. This approach was already used by Halbwachs [Hal79, Section 8.3].

`gotos`. The above instrumentation of loops and their analysis only applies to syntactic loops, but not those formed using `goto`s. Although the analysis may be generalizable to such cases, GOBLINT currently considers any backward `goto` as possibly non-terminating.

Recursion. Non-termination by recursion is detected by constructing an abstract call graph. It accounts for dynamic function calls via pointers and abstract calling contexts (which may, e.g., distinguish some arguments). The graph is constructed from analysis results during post-processing and then checked for cycles. If no cycles (including self-loops) exist in the abstract call graph, then there can be no cycles in the concrete call graph. Otherwise, the program is considered possibly non-terminating due to recursion.

`longjmps`. The reachability and value analyses of GOBLINT provides support for `setjmp`/`longjmp` [Erh+24e]. These constructs can lead to non-termination without any loops or recursion. However, the termination analysis does not support `longjmp` calls and considers them possibly non-terminating.

Multi-threaded Programs. GOBLINT does not support the termination analysis of multi-threaded programs: they are always considered possibly non-terminating. That is because multi-threaded programs may be non-terminating for a variety of other complex reasons, e.g., deadlocks and livelocks.

3.4. GOBLINT in SV-COMP

GOBLINT has participated in five editions of SV-COMP, listed in Table 2. The novel developments for the competition have been documented in three tool papers on the verifier (described here) and one on the validator (described in Section 4.7.3). This section offers a complete historical account of GOBLINT in SV-COMP by supplementing the papers with recounts of the tool-paper–less years. Table 3 gives an overview of the SV-COMP categories where GOBLINT has participated each year. The changes are discussed in the following subsections.

Table 2. SV-COMP participations of GOBLINT, with relevant publications and artifacts

SV-COMP		GOBLINT		
Year	Report	Paper	Artifact	Version (`git describe`)
2021	[Bey21]	[Saa+21]	[Saa+20]	`svcomp21-0-g82e03b87`
2022	[Bey22a]	—	[Bey22b][a]	`svcomp22-0-ge852ac510`
2023	[Bey23]	[Saa+23b]	[Saa+22]	`svcomp23-0-g4f5dcf38f`
2024	[Bey24]	[Saa+24b]	[Saa+23a]	`svcomp24-0-gc2e9465a7`
2025	[BS25]	—	[Saa+24c]	`svcomp25-0-g5512d834b`

[a] GOBLINT was not archived on Zenodo individually, but only together with other tools.

Table 3. SV-COMP categories for GOBLINT: participated (✓), opted out (✓✗) or did not participate (✗); subcategories matching metacategory omitted for brevity

Metacategory / Subcategory	Year				
	2021	2022	2023	2024	2025
ReachSafety	✓	✓	✓	✓	✓
MemSafety	✗	✓✗	✓✗	✓	✓
ConcurrencySafety	✓	✓	✓	✓	✓
NoDataRace	✓[a]	✓[a]	✓	✓	✓
NoOverflows	✓	✓	✓	✓	✓
Termination	✗	✓✗	✓✗	✓	✓
SoftwareSystems	✓	✓	✓	✓	✓
SoftwareSystems--MemSafety*	✓✗	✓✗	✓✗	✓	✓
FalsificationOverall	✗	✗	✗	✗	✗
Overall	✗	✓	✓	✓	✓

[a] Demo category

Setup. Both the tool-info module for BENCHEXEC [BLW17] and the benchmark definition for SV-COMP are named `goblint`. They correspond to running the tool as follows:

```
./goblint --conf conf/svcomp25.json \
          --set ana.specification property.prp \
          --set exp.architecture 64bit \
          input.c
```

The SV-COMP configuration does, among other things, the following:

- Disables the C code preprocessing step since SV-COMP benchmarks are already preprocessed.
- Enables support for SV-COMP specific `__VERIFIER_*` functions.
- Enables more precise but also more expensive domains and analyses suited for SV-COMP tasks.
- Enables the syntactic autotuner, described in Section 3.4.3.
- Assumes the lack of some undefined behavior (signed integer overflows and `NULL` pointer dereferences), when analyzing other properties.
- Post-processes the solution to determine the verdict and construct witnesses, both described below.

Verdicts. Due to over-approximation, abstract interpretation, as employed by GOBLINT, can only determine whether the correctness specification *must hold* or *may be violated*, but not whether a concrete violating execution exists. Therefore, to avoid a large number of false alarms due to imprecision in SV-COMP, GOBLINT only reports results `true` and `unknown`, respectively. For this reason, GOBLINT has never competed in *FalsificationOverall* (see Table 3), which only considers `false` verdicts from tools, limiting GOBLINT to a score of zero. On the other hand, our approach aims to be sound by design (up to out-of-scope features of the input program, e.g., inline assembly).

Witnesses. GOBLINT can generate correctness witnesses both in the automaton-based GraphML format [Bey+16] and the location-based YAML format (Section 2.5.3). The former requires augmenting the analysis to record additional information which is post-processed into the witness automaton [Saa20]. The latter format is better suited for our approach and can more directly accommodate invariants from the solution (see Section 4.6.3). GOBLINT independently generates witnesses in both formats: YAML witnesses are generally smaller and easier to validate [Aya+24], but there are more validators for GraphML witnesses.

Remark 2 (`unknown` witnesses). Even when GOBLINT outputs the `unknown` verdict, it has computed sound abstractions and invariants for the program, but they were not sufficient for GOBLINT to prove the correctness w.r.t. the given specification. GOBLINT can be configured to export this information to a correctness witness. In theory, invariants from such a witness could help a validator verify a program which it could not on its own. Moreover, such witnesses could be useful for cooperative verification (Section 2.5.4), since they may prove *some* parts of the program correct. This feature was intermittently enabled for GraphML witnesses in SV-COMP 2021, and for YAML witnesses in SV-COMP 2024 and 2025. □

3.4.1. SV-COMP 2021

GOBLINT did not (and still does not) report `false` results, which is a clear limitation of our approach, as all competing tools in SV-COMP 2021 did report definite violations. The strength of our approach is soundness, as evidenced by the fact that GOBLINT did not produce any incorrect results in the competition. GOBLINT could not compete in *Overall*, because it did not compete in two unsupported categories (*MemSafety* and *Termination*). Therefore, it was excluded from the correct verifiers ranking in the competition report [Bey21, Table 9]. To compete in *SoftwareSystems*, it opted out from *SoftwareSystems-*-MemSafety*, where it always returned an error verdict indicating the lack of support for the corresponding property (see Table 3).

GOBLINT performed best in the *SoftwareSystems* and *ReachSafety-ProductLines* categories that consist of larger real-world programs, for which our approach is well suited. On the downside, our verifier performed poorly in reachability safety categories that contain smaller programs with intricate correctness conditions, which our abstract domains cannot express.

Even though the support for checking overflows was very new in GOBLINT,[16] it had some success in the *NoOverflows* category. Unfortunately, the tool had no success in *SoftwareSystems-*-NoOverflows*.

Although GOBLINT specializes in concurrency, it performed quite poorly in the *ConcurrencySafety* category. We believe this is because most benchmarks in the category require rather precise analysis of thread interleavings, which is not done in our thread-modular approach.

As GOBLINT has been optimized for data race detection, it unsurprisingly performed better in the *NoDataRace* demo category, as seen in Table 4a. It must be noted that the majority of benchmarks in the category were submitted from our own test suite, consisting of racy and race-free programs.

3.4.2. SV-COMP 2022

GOBLINT did not see much SV-COMP–specific development since the previous year. Nevertheless, the work of Chapter 5 on thread-modular value analyses took place during this period and was implemented in GOBLINT. Based on its evaluation (Section 5.7.1), the cheap yet sufficiently precise Protection-Based Reading was chosen as the default in GOBLINT and also used in SV-COMP 2022. This replaced an old analysis which turned out to be unsound, although this unsoundness was not revealed by the existing SV-COMP benchmark suite. Hence, the corresponding regression tests from GOBLINT's regression test suite were submitted to the competition.[17]

GOBLINT reconfirmed its soundness by not producing any incorrect results in the competition, thanks to a number of targeted fixes. To qualify for the *Overall* category ranking, it participated in (but opted out from) *MemSafety* and *Termination* by

[16] It was implemented mere days before the competition's tool resubmission deadline.

[17] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1314`

Table 4. SV-COMP *NoDataRace* category rankings

(a) SV-COMP 2021 (demo)

	Verifier	Score
1.	**GOBLINT**	134
2.	DARTAGNAN	−151

(b) SV-COMP 2022 (demo)

	Verifier	Score
1.	UGEMCUTTER	151
2.	UTAIPAN	139
3.	**GOBLINT**	124
4.	UAUTOMIZER	120
5.	CSEQ	39
6.	LOCKSMITH	34
7.	UKOJAK	0
8.	DARTAGNAN	−299

(c) SV-COMP 2023

	Verifier	Score
1.	**GOBLINT**	1,304
2.	DEAGLE	1,211
3.	DARTAGNAN	768
4.	UAUTOMIZER	756
5.	UGEMCUTTER	732
6.	UTAIPAN	612
7.	CPACHECKER	400
8.	LOCKSMITH	226
9.	THETA	205
	⋮	

(d) SV-COMP 2024

	Verifier	Score
1.	**GOBLINT**	1,338
2.	UAUTOMIZER	1,220
3.	UGEMCUTTER	1,200
4.	DARTAGNAN	1,112
5.	UTAIPAN	700
6.	THETA	664
7.	CPACHECKER	554
8.	LOCKSMITH	162
9.	ESBMC-KIND	141
10.	SV-SANITIZERS	68
	⋮	

(e) SV-COMP 2025

	Verifier	Score
1.	**GOBLINT**	1,424
2.	RACERF	1,352
3.	DEAGLE	1,338
4.	UAUTOMIZER	1,311
5.	UGEMCUTTER	1,282
6.	DARTAGNAN	1,175
7.	THETA	823
8.	UTAIPAN	778
9.	ESBMC-KIND	518
10.	CPACHECKER	442
	⋮	

always returning an error verdict indicating the lack of support for the corresponding properties (see Table 3). Therefore, the competition report [Bey22a] declares GOBLINT both as the *most correct* verifier and the *greenest* verifier. The latter reflects the efficiency and scalability of abstract interpretation. Beyer and Strejček [BS22b] go on to highlight GOBLINT as the only tool without incorrect results in the *Overall* category.

In SV-COMP 2022 *NoDataRace* remained a demo category but attracted many more participants, as seen in Table 4b. Although GOBLINT remained the best at verifying race freedom by a close margin, it did not win the category. Instead, model checkers from the ULTIMATE tool family achieved the best scores by also reporting definite data races with `false` verdicts. However, unlike GOBLINT, they also reported incorrect verdicts. Furthermore, GOBLINT was victorious over another well-known race detection tool LOCKSMITH [PFH11].

3.4.3. SV-COMP 2023: Autotuning

New abstract domains were added to enhance precision. In addition to interval analysis of integer variables, GOBLINT now performs interval analysis of floating-point variables following Miné [Min04, Section 4], and maintains congruence information [Gra89]. Furthermore, the APRON library [JM09] was integrated for relational analysis. GOBLINT includes novel approaches to relational analysis of concurrent programs, inferring relations between jointly-protected global variables (see Section 5.5). Data race detection was made more precise using MHP analysis to filter out spurious races with threads that have already been joined or have not yet been created (see Section 3.5).

In SV-COMP 2023, *NoDataRace* became an official category and existing *ConcurrencySafety* reachability tasks were newly included into it. This is where GOBLINT really shined: it proved 652 out of 783 programs race-free, thereby winning the category, as seen in Table 4c. Importantly, GOBLINT remained sound in the competition.

Autotuning. Although GOBLINT is highly configurable, it previously used a fixed SV-COMP configuration which did not depend on the property nor the input program. We implemented analysis configuration *autotuning* based on cheap *syntactic* heuristics on the program, before the analysis begins. This can increase precision, by activating analyses that are more expensive yet offer crucial precision, and also decrease resource usage, by deactivating redundant analyses. The particular features have been chosen according to how expert users might configure GOBLINT for a given program. Measurements of program size (e.g., number of functions, loops, variables) are taken into account to limit slowdown on larger programs.

GOBLINT provides a multitude of concurrency-related analyses (e.g., races, symbolic locking patterns, thread joins) that have no use in single-threaded programs, which abound in SV-COMP. Hence, all such analyses are now automatically deactivated for programs that never create any threads.

GOBLINT implements a wide variety of numerical abstract domains, but most are not necessary for every program, thus, offering many possibilities for autotuning. Interval information is omitted from the calling contexts of recursive functions to avoid an explosion of contexts in which they are to be analyzed. While the congruence domain is generally active on small programs; for medium-sized programs, it is only enabled for functions involving the modulo operator, either directly or indirectly (up to a fixed depth in the call stack). If the program uses `enum`s, then the integer domain for enumerator sets is activated. Octagon analysis is enabled for those local variables which occur most often in linear expressions and conditions. Interval and octagon widening thresholds are extracted from conditional expressions containing constants. Such thresholds are especially useful for flow-insensitive analysis of global variables in multi-threaded programs, since no narrowing is performed on flow-insensitive invariants (see Remark 1, p. 34).

Loop unrolling is a well-known technique to increase the precision of static analysis. GOBLINT now unrolls loops up to their static bounds or feasible unrolled code size. Loops which contain memory allocation, thread creation, or error function calls, are prioritized, since unique heap variables and thread IDs are key to maintaining analysis precision.

Impact. We now describe the impact of autotuning, based on our own preliminary comparative evaluation before the competition took place. Unlike the official SV-COMP evaluation, we used a 1 GB memory limit, which is sufficient for most tasks GOBLINT can solve, and no witness validators.

As noted above, the majority of SV-COMP programs across all categories are single-threaded; thus, the greatest improvement comes from disabling all concurrency analyses in those cases. This yields a notable reduction in runtime and memory usage, as shown in Table 5, improving overall efficiency without compromising precision. The table distinguishes tasks correctly solved by GOBLINT from all tasks, while separately considering all properties which are supported by GOBLINT (at the time) and are applicable to single-threaded programs (thus, excluding `no-data-race`).

Table 5. Average change in resource usage due to disabling all concurrency analyses for single-threaded programs

		Over solved tasks		Over all tasks	
Property	Tasks	CPU time	Memory	CPU time	Memory
`unreach-call`	9,537	−16%	−4%	−5%	−8%
`no-overflow`	7,502	−5%	0%	−16%	−6%
`no-data-race`	783	—	—	—	—
Total	17,822	−11%	−2%	−9%	−7%

The second-greatest improvement is due to the use of relational analysis with octagons. Although this incurs a runtime penalty, it increases the number of correct verdicts notably. The improvement is especially visible in *NoOverflows*, where it yields 104 additional correct results. We also confirmed that the automatic selection of octagon variables is better than tracking all variables: our selection yields more correct verdicts (due to fewer timeouts), while successfully avoiding an unnecessarily large performance penalty.

Autotuning along the other axes is not as impactful. Nevertheless, each leads to GOBLINT being able to solve tasks it could not otherwise. Hence, a small increase in score is achieved, justifying their use. Although disabling unnecessary concurrency analyses reduces resource usage; overall, this performance improvement is canceled out by the simultaneous use of expensive analyses enabled by autotuning, such as octagons. Thus, GOBLINT can solve more tasks while retaining the same level of overall efficiency observed in previous editions of the competition.

3.4.4. SV-COMP 2024: Memory Safety and Termination

GOBLINT was extended with support for the detection of memory safety bugs and non-termination, as described in Sections 3.3.4 to 3.3.6. Thus, for the first time, GOBLINT properly supported *all* SV-COMP properties (Section 2.6.1) and could compete without opt-outs (see Table 3). Of particular note is the support for verifying the memory safety of multi-threaded programs in a thread-modular way, yielding the second-best score in *ConcurrencySafety-MemSafety*, after DEAGLE [HSF22]. GOBLINT once again maintained its 1st place in the *NoDataRace* category, as seen in Table 4d.

The competition results are commonly visualized using *quantile plots* like the one in Fig. 6 for SV-COMP 2024 [Bey24]. For each tool, the plot shows the total score from the tasks correctly solved under every CPU time limit (up to 900 s), offset to the left by the (negative) total score from all incorrectly solved tasks (regardless of time). Thus, the lower and the further to the right, the better. This plot highlights the efficiency of GOBLINT. Despite targeting only proofs, which are traditionally considered to be more time-consuming than counterexamples, it leads the pack in terms of points obtained in ≤ 10 s in *Overall*. This is most pronounced when considering runtimes ≤ 1 s, with GOBLINT solving thousands of tasks faster than the startup time of many other tools.

GOBLINT redemonstrated its soundness in the competition, i.e., it did not produce any false negatives. Only three other tools (AISE [WC24], BRICK, MOPSA [Mon+24]) achieved this, but in limited categories. GOBLINT was, thus, the only sound tool in SV-COMP 2024 to support all properties, and the only sound tool represented in the *Overall* ranking. In Fig. 6, the graph for every other tool begins at negative score, indicating some incorrect results, which include false positives.

Inspired by the soundiness manifesto [Liv+15], GOBLINT was extended to soundly analyze programs using `setjmp`/`longjmp` and flag their misuse [Sch+23a].

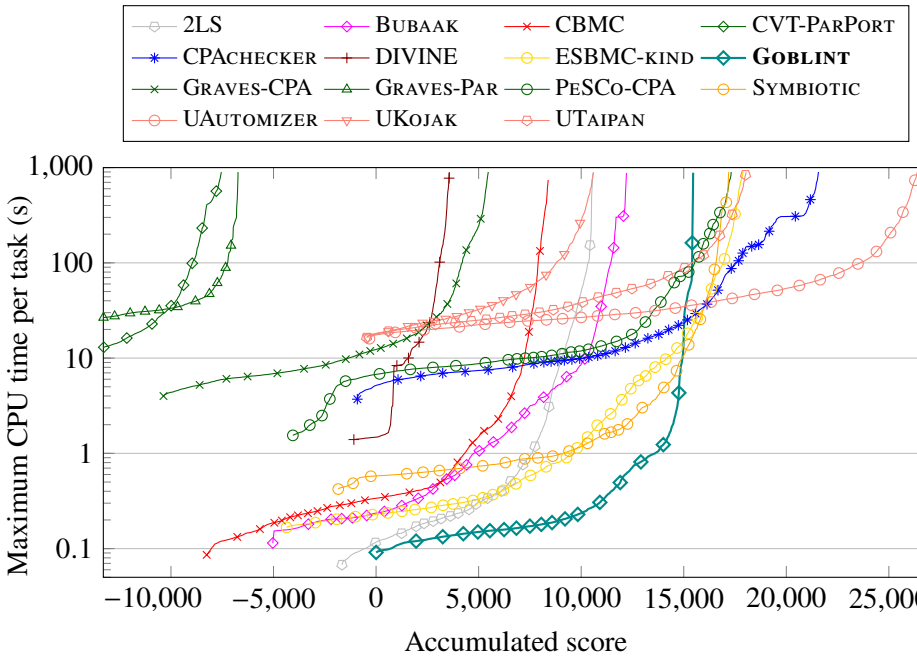


Figure 6. SV-COMP 2024 *Overall* category results as a quantile plot

Programs using this language feature were contributed to the competition.[18] Unsurprisingly, Goblint scored the most points on these tasks, which reveal unsoundness in many other tools.

3.4.5. SV-COMP 2025

Although SV-COMP verification tasks (Section 2.6.2) specify a data model (ILP32/LP64), Goblint essentially ignored it during all of its previous SV-COMP participations.[19] Surprisingly, the SV-COMP benchmark suite failed to expose this unsoundness. In particular, the data model determines the size of the `long` data type which directly matters for the `no-overflow` property, although indirectly it can impact all properties. Support for the two different data models was added to Goblint for SV-COMP 2025 to remedy the soundness issue. Based on our own preliminary comparative evaluation (no witness validators), the data model fix had the following impact:

- 56 previously-`true` verdicts became `unknown`. In these cases, Goblint previously coincidentally reached the correct result unsoundly, while having no incorrect results by pure chance.
- 2 previously-unknown verdicts became `true`. In these cases, the tighter bounds on `long` variables can prove more assertions.

[18] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1465`
[19] It was only used in the metadata for the witnesses it generated, in order for the witnesses to be syntactically correct.

Despite the data-model–related point losses, significant precision improvements were made for the `no-overflow` property. GOBLINT internally uses its non-relational integer domains for more than just computations in the analyzed program: abstract offsets in the pointer domain (Section 3.3.1), abstract sizes of arrays and allocated blocks (Section 3.3.4), results of queries (Section 3.2), and backward operators for refining abstract states at `assume` statements (Section 4.5.1). These computations do not appear in the program and, thus, overflows in them can be soundly ignored (as long as this is still sound for the auxiliary uses of integer domains). Therefore, many false positives of GOBLINT were fixed by ignoring overflows during the tool's internal auxiliary computations.

The work in Section 6.7.1 took place between SV-COMP 2024 and 2025, and as described in Section 6.8.2, revealed and fixed numerous bugs in GOBLINT which are closely related to SV-COMP. For example, to not generate trivial invariants into witnesses for booleans (e.g., `x == 0 || x == 1` when `x` is of type `_Bool`) and for the congruence domain (e.g., `x % 1 == 0`).

In addition to not outputting into witnesses trivial invariants, which unnecessarily increase their size and validation effort, GOBLINT now also simplifies non-relational integer invariants to avoid redundancy. For example:

- If the abstraction is a definite integer according to any integer domain, then output only a single equality.
- If the abstraction is an enumerator set, then output a disjunction of equalities (without interval bounds or exclusion sets).
- Output the tightest interval bounds based on all integer domains. In particular, avoid duplicate bounds from the interval domain and the exclusion set range (in bits).[20]
- Do not output exclusion set inequalities for values outside of above interval bounds. In particular, avoid `x != 0` (abstraction of `true` in C) when `x` is already strictly positive.

Although mutual refinement of integer domains [Min17] can reduce some redundancy, it does not cover all of the above. In particular, even if multiple domains know the definite value or bounds, we do not want duplicate invariant expressions.

Finally, we note that GOBLINT maintained its 1st place in the *NoDataRace* category, as seen in Table 4e. Moreover, GOBLINT was the best sound tool represented in the *Overall* ranking (the only other one being MOPSA [MOM25]).

3.4.6. Threats to Validity

Although SV-COMP is not run by us, the previous sections describe some of the competition results. Validity of these results largely falls to the validity of SV-COMP itself, which is discussed here only in part. In addition to the competitions,

[20] Although the latter is a crude abstraction of the interval domain, it can be more precise than the interval domain itself due to different widening behavior.

the previous sections included two small preliminary evaluations performed by us: for autotuning (in Section 3.4.3) and for the data model fix (in Section 3.4.5).

Construct Validity. SV-COMP is not run to answer specific questions, rather it has high-level objectives which have been last discussed by Beyer [Bey20]. Our preliminary evaluations use an SV-COMP–like setup to predict the impact of particular changes to subsequent competition results.

Internal Validity. Both SV-COMP and our preliminary evaluations were performed using the BENCHEXEC framework [BLW17] to ensure reliable benchmarking. It provides precise and reliable measurement and limitation of resource usage: CPU time, wall time and (peak) memory usage. Furthermore, BENCHEXEC isolates individual benchmark runs using containers and distributes parallel runs fairly across CPU cores.

The SV-COMP benchmark suite is maintained by the community around the competition. Anyone can submit new benchmarks or fixes to existing benchmarks, and these must be reviewed by other community members before they can be incorporated into the suite. This open process ensures that the benchmarks fulfil SV-COMP rules and are correctly labeled, which is crucial for measuring soundness.

Prior to the main evaluation, SV-COMP includes preliminary runs (preruns) of the participating tools in the competition environment. Participating teams can inspect the prerun results to confirm that their tool is working as intended. Moreover, teams can fix and improve their tool based on the prerun results, e.g., fix soundness issues. Finally, witness validation strengthens the validity of verifier verdicts.

External Validity. The SV-COMP benchmark suite for C is the largest suite of its kind [Bey20]. However, only the *SoftwareSystems* category contains large real-world programs. To avoid semantic issues, SV-COMP does not allow multiple kinds of undefined behavior in the same program. This rule inhibits the inclusion of real-world programs for which this is difficult to guarantee. Therefore, good results in SV-COMP may not carry over to the verification of real-world programs.

Although GOBLINT has maintained absolute soundness on the SV-COMP benchmark suite, it is not free of soundness issues. The previous sections describe multiple instances where the suite has not been diverse enough to reveal certain unsoundness. Striving for improvement, we have submitted new benchmarks to SV-COMP. Additionally, the GOBLINT autotuner has (at least in part) been specifically tuned for SV-COMP benchmarks and resource limits. This is by design, but the autotuner is likely less effective on real-world programs.

Our preliminary evaluations were performed in a different context from SV-COMP: on different hardware, using smaller resource limits and without witness validation. Thus, these evaluations do not reflect the precise impact on SV-COMP results but are merely an approximation. This would have been the case even with SV-COMP resource limits and witness validation due to hardware differences. Moreover, this information cannot be extracted from official SV-COMP results after the fact, since only one version of GOBLINT was evaluated in the competition.

3.5. Data Race Analysis

The sound data race analysis of GOBLINT intends to automatically prove the race freedom of a program. This is achieved when all pairs of memory accesses can be excluded from being racy.

This analysis was last described by Vojdani et al. [Voj+16]. Since then, numerous small (but notable) improvements have been made to the analysis while working on thread-modular value analyses (Chapter 5) and performing general maintenance. Although improvements to the race analysis are not part of the goals and contributions of the thesis, their descriptions complete the up-to-date overview of GOBLINT.

Language. For this section, we extend the language from Section 2.2 with:

Thread IDs. There is a type `tid` whose values are uniquely assigned concrete thread IDs.

Thread creation with ID. The primitive $t \leftarrow \mathsf{create}(f)$ creates a new thread which starts executing the function $f \in \mathcal{F}$ concurrently with the creator thread, and assigns its unique ID to the variable $t \in \mathcal{V}$.

Thread joining. The primitive $\mathsf{join}(t)$ joins the thread with the ID stored in the variable $t \in \mathcal{V}$ (of type `tid`), i.e., it waits until the corresponding thread has finished executing.

Improvements. In particular, we describe the following advances:

RI1. Generalize race analysis from protecting mutexes to pairwise checks for the following temporal improvements (Section 3.5.1).

RI2. Exclude races with the current thread itself (i.e., self-races), if the thread is unique (Section 3.5.3).

RI3. Exclude races with threads which have not yet been created (Section 3.5.3).

RI4. Exclude races with ancestor threads before the current thread has been created (Section 3.5.3).

RI5. Exclude races with threads which have already been joined (Section 3.5.3).

Tightly coupled, Sections 3.5.2 and 3.5.4 describe various forms of abstract thread IDs.

3.5.1. Pairwise Race Analysis

Vojdani et al. [Voj+16] describe the race analysis of GOBLINT as follows. For each global variable g it computes the set of *protecting mutexes* $\mathcal{M}_{\mathrm{rw}}[g] \subseteq \mathcal{M}$ definitely held whenever g is accessed (read or written). At the end of the analysis, it reports a race for global g if it has no protecting mutexes, i.e., $\mathcal{M}_{\mathrm{rw}}[g] = \varnothing$, and at least one of its accesses is a write.

Unfortunately, this simple approach does not generalize to achieve the desired (temporal) improvements. Moreover, even with just mutex-based mutual exclusion, this may be imprecise.

Example 5 (Race freedom without protecting mutexes)**.** Consider the following three-threaded program:

```
1  int g = 0;
2  mutex m1, m2, m3;

3  void main() {
4    create(t1);
5    create(t2);
6    lock(m1);
7      lock(m2);
8        g++;
9      unlock(m2);
10   unlock(m1);
11 }

12 void t1() {
13   lock(m2);
14     lock(m3);
15       g++;
16     unlock(m3);
17   unlock(m2);
18 }

19 void t2() {
20   lock(m1);
21     lock(m3);
22       g++;
23     unlock(m3);
24   unlock(m1);
25 }
```

The protecting mutexes for g can be computed as the intersection of all locksets at accesses:

$$\mathcal{M}_{\text{rw}}[\texttt{g}] = \{\texttt{m1}, \texttt{m2}\} \cap \{\texttt{m2}, \texttt{m3}\} \cap \{\texttt{m1}, \texttt{m3}\} = \varnothing,$$

which does not rule out a race.

However, considering the accesses pairwise, it is clear that each pair has one mutex in common, meaning they are mutually exclusive and cannot race. □

Schwarz et al. [Sch+23b] changed GOBLINT to a more conventional race analysis: accesses to a global variable are checked pairwise for mutual exclusion. This approach admits the desired temporal improvements (**RI1**). The pairwise checks are performed using a binary predicate mustNotRace which decides based on some scheduling information associated with each access. For example, if two accesses are made with the locksets S and S', respectively, then the predicate is defined as

$$\text{mustNotRace}_S\ S\ S' \stackrel{\text{def}}{=} S \cap S' \neq \varnothing.$$

Note that in our thread-modular approach, each access must also be checked against racing with itself. An access might be from a non-unique thread, representing a countable set of concrete threads (e.g., created in a loop), which GOBLINT analyzes as one abstract thread. However, one copy of such a concrete thread may race with another copy, which corresponds to a self-race in the abstract.

In the following subsections, we associate additional scheduling information with each access and define the corresponding predicates on them. All of these can be combined such that the overall mustNotRace predicate is a disjunction of the individual components, i.e., a pair of accesses is declared race-free if at least one component of the scheduling information can prove it. Schwarz and Erhard [SE26] have subsequently formalized the new approach using the local trace semantics (Section 5.2.3).

Interference Graph. The pairwise race checking implicitly induces an undirected graph for each global variable. The vertices of such a graph are all accesses to the global variable. The edges are induced by the mustNotRace predicate: there is an edge between two accesses if the predicate on them evaluates to false (i.e., the accesses may race). We call this a *data race interference graph*.

This graph represents the most precise information on data races that GOBLINT has inferred, however, it cannot easily be presented to the user via the usual analysis output mechanisms (command-line output and IDE warnings). The crudest form of output is binary: if the graph is empty (i.e., has no edges), then there are no races; otherwise, there may be races. In the old race analysis based on protecting mutexes, this was the *only* granularity of output possible. The interference graph allows output with intermediate granularity.

Connected Components. For example, GOBLINT groups accesses in the output by the connected components in the race interference graph. Such groups can still be easily presented to the user via the usual outputs, while offering more precise information that GOBLINT already has: accesses between two groups are race-free. A singleton component may still correspond to a race if it has a self-loop, which represents a self-race. However, all non-racing accesses, which are singleton components without self-loops, form an *independent set* and are presented as a separate non-racing group.

3.5.2. History-Based Thread IDs

Schwarz et al. [Sch+23b, Section 6] present an analysis to compute abstract thread IDs and their uniqueness. That analysis enhances precision of race analysis by excluding reads, e.g., from threads that have not yet been started.

We identify threads by their thread creation histories, i.e., by sequences of create arcs. Technically, such sequences can grow infinitely, so we use a finite abstraction which collapses possibly-repeated elements into a set. Let $\mathcal{P} \subseteq \mathcal{N} \times \mathcal{F}$ pair (thread creation) nodes with the functions of the created threads:

$$\mathcal{P} \stackrel{\text{def}}{=} \{\langle u, f\rangle \in \mathcal{N} \times \mathcal{F} \mid \exists v \in \mathcal{N} : (u, \mathsf{create}(f), v) \in \mathcal{A}\}.$$

By the out-degree assumption on CFGs, there is at most one such v, so this simplified definition does not introduce ambiguity.[21] Let $\mathsf{main} \cdot \mathcal{P}^*$ denote the set of finite sequences starting with main followed by (possibly zero) elements from $\mathcal{P}$. History-based thread IDs are the set

$$\mathbb{T} \stackrel{\text{def}}{=} \left\{ (p, s) \in (\mathsf{main} \cdot \mathcal{P}^*) \times 2^{\mathcal{P}} \;\middle|\; \begin{array}{l}\text{each } \langle u, f\rangle \in \mathcal{P} \text{ occurs at most}\\ \text{once in } p \text{ and } s \text{ combined}\end{array} \right\}. \tag{1}$$

Each $(p, s) \in \mathbb{T}$ consists of a *prefix* p and a *set* s.

[21] By the same assumption, f is also uniquely defined, but we keep it explicit for clarity. The explicit function is also useful in practice where the PTHREADS library allows the function of the created thread to be determined dynamically (via a function pointer).

Listing 2. Program for illustrating history-based thread IDs

```
1 void main() {
2   create(t);
3   for (int i = 0; i < 5; i++)
4     create(t);
5 }
6 void t() {
7   create(t);
8 }
```

The ID of the main thread is $\mathsf{main}_{\mathbb{T}} \stackrel{\text{def}}{=} (\mathsf{main}, \varnothing)$. An ID $(p, s) \in \mathbb{T}$ where $s = \varnothing$ is called *unique*, because it represents one concrete thread ID. On the other hand, IDs where $s \neq \varnothing$ are called *non-unique* or *ambiguous*, possibly representing multiple concrete thread IDs.

Example 6. In the program in Listing 2, the `main` thread with the ID $\mathsf{main}_{\mathbb{T}}$ first creates a unique thread with the ID $(\mathsf{main} \cdot \langle \ell_2, \mathtt{t} \rangle, \varnothing)$, where ℓ_2 indicates the node before line 2. That thread creates a thread with the ID $(\mathsf{main} \cdot \langle \ell_2, \mathtt{t} \rangle \cdot \langle \ell_7, \mathtt{t} \rangle, \varnothing)$, which is also unique.

On line 7, the latter thread recursively creates[22] another one starting at `t`. Simply appending $\langle \ell_7, \mathtt{t} \rangle$ to the sequence would violate the invariant (1), therefore, it should receive the ID $(\mathsf{main} \cdot \langle \ell_2, \mathtt{t} \rangle, \{\langle \ell_7, \mathtt{t} \rangle\})$, where the repeated create arc has been moved to the set. That thread recursively creates yet another one at the same arc; however, the created thread will receive the same non-unique ID. This is also the case for all subsequent recursively created threads at this point, avoiding unbounded explosion of abstract thread IDs.

A more common source of non-unique threads are loops. In the first iteration, the `main` thread creates a unique thread with the ID $(\mathsf{main} \cdot \langle \ell_4, \mathtt{t} \rangle, \varnothing)$ on line 4. In subsequent iterations, $\mathsf{main}_{\mathbb{T}}$ creates another thread at the same node. These should not get unique abstract thread IDs because the thread ID would no longer represent a single concrete thread. Thus, all subsequent iterations create non-unique threads with the ID $(\mathsf{main}, \{\langle \ell_4, \mathtt{t} \rangle\})$. □

These history-based thread IDs can be viewed as an abstraction of stacks (see also Remark 3, p. 70). The operator $\circ$ for appending a create arc to a thread ID (called *push* for stacks) is defined as follows:

$$
\begin{aligned}
(p, s) \circ \langle u, f \rangle \stackrel{\text{def}}{=} \; & \textbf{if } \langle u, f \rangle \in p \textbf{ then} \\
& \quad \textbf{let } (p_0, p_1) \textbf{ such that } p_0 \cdot \langle u, f \rangle \cdot p_1 = p \textbf{ in} \\
& \quad (p_0, \{\langle u, f \rangle\} \cup \overline{p_1} \cup s) \\
& \textbf{else if } s = \varnothing \textbf{ then} \\
& \quad (p \cdot \langle u, f \rangle, \varnothing) \\
& \textbf{else} \\
& \quad (p, s \cup \{\langle u, f \rangle\}),
\end{aligned}
\tag{2}
$$

where $\bar{p}$ denotes the set of pairs occurring in the sequence p.

[22] Recursive thread creation is unusual in practice.

It is easy but useful to observe the following:

Proposition 1. *These thread IDs and their append $\circ$ have the following properties:*

1. *Append cannot turn an ambiguous thread ID into a unique one.*
2. *Append cannot remove pairs $\langle u, f\rangle$ from a thread ID.*
3. *Append cannot lengthen the prefix of an ambiguous thread ID.*
4. *Append cannot change pairs (in place) in the prefix of a thread ID.*

Analysis. To automatically compute the thread IDs and use them for other analyses, we define a side-effecting constraint system with unknowns $[u, t]$ that correspond to nodes $u \in \mathcal{N}$ in a specific abstract thread $t \in \mathbb{T}$. The constraints are the following:

$$\begin{aligned}(\eta, \eta\,[\mathsf{st}_{\mathsf{main}}, \mathsf{main}_{\mathbb{T}}]) &\sqsupseteq \mathsf{init}^\sharp,\\ (\eta, \eta\,[v, t]) &\sqsupseteq [\![[u, t], s]\!]^\sharp\,\eta \qquad\qquad \forall (u, s, v) \in \mathcal{A}, t \in \mathbb{T}.\end{aligned}$$

Each right-hand side $[\![[u, t], s]\!]^\sharp$ is strict w.r.t. $\eta\,[u, t]$. Thus, for simplicity, we only present their definitions for the cases where $\eta\,[u, t] \neq \bot$.

The non-trivial part of the analysis of history-based thread IDs is the identification of threads created in a loop. This can be achieved by collecting a set C of create arcs encountered by the current thread. These sets belong to the abstract domain $2^{\mathcal{P}}_\bot \stackrel{\text{def}}{=} 2^{\mathcal{P}} \cup \{\bot\}$, which is the set abstract domain $2^{\mathcal{P}}$ (ordered by subset) extended with a distinct least element $\bot$. Technically, the latter is necessary to distinguish unreachable threads ($\bot$) from reachable threads which have not (yet) created any threads themselves ($\varnothing$).

The right-hand sides for this analysis are the following:

$$\begin{aligned}\mathsf{init}^\sharp &\stackrel{\text{def}}{=} (\varnothing, \varnothing),\\ [\![[u, (p, s)], \mathsf{create}(f)]\!]^\sharp\,\eta &\stackrel{\text{def}}{=} \textbf{let } C = \eta\,[u, (p, s)] \textbf{ in}\\ &\quad \textbf{let } (p', s') = (p, s) \circ \langle u, f\rangle \textbf{ in}\\ &\quad \textbf{let } (p'', s'') =\\ &\quad\quad \textbf{if } s' = \varnothing \wedge \langle u, f\rangle \in C \textbf{ then } (p, \{\langle u, f\rangle\})\\ &\quad\quad \textbf{else } (p', s')\\ &\quad \textbf{in}\\ &\quad (\{[\mathrm{st}_f, (p'', s'')] \mapsto \varnothing\}, C \cup \{\langle u, f\rangle\}),\\ [\![[u, t], s]\!]^\sharp\,\eta &\stackrel{\text{def}}{=} (\varnothing, \eta\,[u, t]) \qquad\qquad \forall s \neq \mathsf{create}(f).\end{aligned}$$

The right-hand side for $\mathsf{create}(f)$ mainly relies on the operator $\circ$ defined above, but is also responsible for using and updating the set C. It side-effects to the entry node st_f of the function f with the newly-determined thread ID the empty set of created threads (by the new thread).

Example 7. When applied to the program in Listing 2, this analysis computes the thread IDs presented in Example 6. While most of them simply follow from the definition of $\circ$, we discuss two interesting cases here.

After creating the thread on line 2, the state of the `main` thread is $\{\langle \ell_2, \mathtt{t}\rangle\}$. In the first loop iteration on line 4, a unique thread is created and the state of the `main` thread becomes $\{\langle \ell_2, \mathtt{t}\rangle, \langle \ell_4, \mathtt{t}\rangle\}$. In subsequent iterations, $\langle \ell_4, \mathtt{t}\rangle \in C$, thus, a non-unique thread with the ID $(\mathsf{main}, \{\langle \ell_4, \mathtt{t}\rangle\})$ is created.

When that created thread reaches line 7, it also creates a non-unique thread with the ID $(\mathsf{main}, \{\langle \ell_4, \mathtt{t}\rangle, \langle \ell_7, \mathtt{t}\rangle\})$, even though $\langle \ell_7, \mathtt{t}\rangle$ did not appear in the creator's ID nor in its state, which is $\varnothing$. This exemplifies the last case in Eq. (2). □

Predicates. In general, abstract thread IDs should provide the following functions:

- unique : $\mathbb{T} \to \mathbb{B}$ tells whether a thread ID is unique.
- mayBeAncestor : $\mathbb{T} \to \mathbb{T} \to \mathbb{B}$ checks whether a thread *may* (transitively) create another.
- mustBeAncestor : $\mathbb{T} \to \mathbb{T} \to \mathbb{B}$ checks whether a thread *must* (transitively) create another.

For the domain $\mathbb{T}$, Schwarz et al. [Sch+23b] define these as

$$\begin{aligned}
\text{unique}\,(p, s) &\stackrel{\text{def}}{=} s = \varnothing,\\
\text{mayBeAncestor}\,(p, s)\,(p', s') &\stackrel{\text{def}}{=} \overline{p} \cup s \subseteq \overline{p'} \cup s',\\
\text{mustBeAncestor}\,(p, s)\,(p', s') &\stackrel{\text{def}}{=} \text{unique}\,(p, s) \wedge p \preceq p',
\end{aligned}$$

where $p \preceq p'$ is true if p is a prefix of p' (or equal to it). However, by further exploiting the properties from Proposition 1, more precise definitions can be given:

$$\begin{aligned}
\text{mustBeAncestor}\,(p, s)\,(p', s') \stackrel{\text{def}}{=}\ & \textbf{if } \neg(\text{unique}\,(p, s)) \textbf{ then } \mathsf{false}\\
& \textbf{else if } \text{unique}\,(p', s') \wedge p = p' \textbf{ then } \mathsf{false}\\
& \textbf{else } p \preceq p',\\
\text{mayBeAncestor}\,(p, s)\,(p', s') \stackrel{\text{def}}{=}\ & \textbf{if } \text{unique}\,(p', s') \textbf{ then}\\
& \quad \text{mustBeAncestor}\,(p, s)\,(p', s')\\
& \textbf{else if } \text{unique}\,(p, s) \wedge p \preceq p' \textbf{ then } \mathsf{true}\\
& \textbf{else if } p' \preceq p \textbf{ then}\\
& \quad \textbf{let } d \textbf{ such that } p' \cdot d = p \textbf{ in}\\
& \quad \overline{d} \subseteq s' \wedge s \subseteq s'\\
& \textbf{else } \mathsf{false}.
\end{aligned}$$

As a sanity check, it is straightforward to verify the following:

Proposition 2. *The refined predicates satisfy the following properties:*

1. *The* main *thread is unique:* $\text{unique}\,\mathsf{main}_{\mathbb{T}}$.
2. *The* main *thread has no ancestor:* $\nexists t \in \mathbb{T}.\ \text{mayBeAncestor}\,t\,\mathsf{main}_{\mathbb{T}}$.
3. *The* main *thread is an ancestor of all other threads:*

$$\forall t' \in \mathbb{T} \setminus \{\mathsf{main}_{\mathbb{T}}\}. \quad \text{mustBeAncestor}\,\mathsf{main}_{\mathbb{T}}\,t'.$$

4. *Definite ancestors are possible ancestors:*

$$\forall t, t' \in \mathbb{T}. \quad \text{mustBeAncestor}\, t\, t' \implies \text{mayBeAncestor}\, t\, t'.$$

This analysis of thread IDs and their uniqueness can be considered as a MHP (or, more precisely, must-not-happen-in-parallel) analysis. MHP information is useful in a variety of scenarios, e.g., a thread-modular analysis of data races or deadlocks, which does not consider thread IDs and joining, can be refined with this analysis to exclude more data races or deadlocks.

Remark 3 (Loop-free call strings). Erhard et al. [Erh+24d] use similar ideas for abstracting call strings/stacks to deal with conventional recursion. However, their *loop-free call strings* take it a step further by allowing unique elements after non-unique sets and longer interleavings. Ported back to our context, their domain is the following:

$$\mathbb{T}' \stackrel{\text{def}}{=} \left\{ t \in \mathsf{main} \cdot \left(\mathcal{P} \cup 2^{\mathcal{P}}\right)^* \;\middle|\; \text{each } \langle u, f\rangle \in \mathcal{P} \text{ occurs at most once in } t \right\}.$$

We do not explore this domain for thread IDs further. □

3.5.3. Excluding Races with Thread IDs

Self-Races. The thread IDs from the previous section can be used to exclude races with the current thread itself, if the thread is unique (**RI2**). By associating thread IDs with accesses, we can define

$$\text{mustNotRace}_t\, t\, t' \stackrel{\text{def}}{=} \text{unique}\, t \wedge t = t'. \tag{3}$$

Not-Yet-Created Threads. By associating the sets C from the previous section also with accesses (in addition to thread IDs), we can exclude races with threads which have not yet been created (**RI3**). This is the case for a thread t' that

1. must be a descendant of the current thread t;
2. according to the set C of threads created by the current thread, it has not yet created t' directly, nor any thread which may be an ancestor of t'.

Formally, the predicate $\text{hasNotYetCreated}\, (t, C)\, t'$, defined as

$$\begin{aligned}&\text{mustBeAncestor}\, t\, t'\\ &\quad \wedge \forall \langle u, f\rangle \in C.\ t \circ \langle u, f\rangle \neq t' \wedge \neg(\text{mayBeAncestor}\, (t \circ \langle u, f\rangle)\, t'),\end{aligned}$$

is true whenever thread t' has definitely not yet been created. Thus, we can define

$$\text{mustNotRace}_C\, (t, C)\, (t', C') \stackrel{\text{def}}{=} \text{hasNotYetCreated}\, (t, C)\, t'. \tag{4}$$

Note that this definition is not symmetric. Viewed from the second thread's perspective, its accesses cannot race with accesses by its ancestor threads which definitely happened before the thread was created (**RI4**). To include only those which happen after the creation of the current thread, we can also define

$$\text{mustNotRace}_{C'}\, (t, C)\, (t', C') \stackrel{\text{def}}{=} \text{hasNotYetCreated}\, (t', C')\, t. \tag{5}$$

Thus, improvement **RI4** is merely the inverse of improvement **RI3**.

Joined Threads. To exclude races with threads which have already been joined (**RI5**), the analysis must be extended to compute the set J of thread IDs (domain ordered by superset) which have *definitely* been joined, either directly by the current thread or transitively through other joined threads.[23] By associating the sets J from local states also with accesses (in addition to thread IDs), we can define

$$\text{mustNotRace}_J\ (t, J)\ (t', J') \stackrel{\text{def}}{=} t' \in J. \tag{6}$$

Since this definition is not symmetric, we should also use the inverse

$$\text{mustNotRace}_{J'}\ (t, J)\ (t', J') \stackrel{\text{def}}{=} t \in J'. \tag{7}$$

3.5.4. Simpler Thread IDs

Notice that the mustNotRace predicates defined in the previous section are actually not specific to the history-based thread IDs. Thus, we briefly describe two alternative thread ID domains, based on existing ad hoc techniques in GOBLINT. This demonstrates the generality of our new approach by seamlessly integrating the existing constructions.

Plain. Prior to the history-based thread IDs, GOBLINT used a simpler form of thread IDs (now called *plain*) which only indicate the last create arc in the thread creation history. Since the main thread has no creator, its ID $\mathsf{main}_{\mathbb{T}}$ must be additionally included. Thus, the plain thread IDs are the set $\mathbb{T}_{\text{plain}} \stackrel{\text{def}}{=} \{\mathsf{main}_{\mathbb{T}}\} \cup \mathcal{P}$.

They can also be viewed as an abstraction of stacks where the entire stack is represented by the topmost element. The corresponding operator for appending (i.e., pushing) a create arc to a thread ID is simply $t \circ \langle u, f\rangle \stackrel{\text{def}}{=} \langle u, f\rangle$. The constraints and right-hand sides from Section 3.5.2 work for plain thread IDs with one difference; thread creation is defined by the following:

$$[\![[u, t], \mathsf{create}(f)]\!]^\sharp\, \eta \stackrel{\text{def}}{=} \textbf{let } C = \eta\,[u, t] \textbf{ in } (\{[\mathrm{st}_f, \langle u, f\rangle] \mapsto \varnothing\}, C \cup \{\langle u, f\rangle\}),$$

where the simpler definition of $\circ$ has been inlined.

The only unique plain thread ID is $\mathsf{main}_{\mathbb{T}}$, which has no ancestors and is an ancestor to every other ID. Accordingly, we can define the predicates:

$$\begin{aligned} \text{unique}\, t &\stackrel{\text{def}}{=} t = \mathsf{main}_{\mathbb{T}}, \\ \text{mayBeAncestor}\, t\, t' &\stackrel{\text{def}}{=} t' \neq \mathsf{main}_{\mathbb{T}}, \\ \text{mustBeAncestor}\, t\, t' &\stackrel{\text{def}}{=} t = \mathsf{main}_{\mathbb{T}} \wedge t' \neq \mathsf{main}_{\mathbb{T}}, \end{aligned}$$

which also satisfy the properties from Proposition 2.

[23] Our analyses can only add unique thread IDs to the must-joined sets, but this is not required by the approach. It is future work to develop analyses which could determine non-unique threads as definitely joined.

Given these predicates, Eq. (3) simplifies to

$$\text{mustNotRace}_t\ t\ t' \stackrel{\text{def}}{=} t = t' = \mathsf{main}_{\mathbb{T}}.$$

Moreover, Eqs. (4) and (5) simplify, respectively, to

$$\text{mustNotRace}_C\ (t, C)\ (t', C') \stackrel{\text{def}}{=} t = \mathsf{main}_{\mathbb{T}} \wedge t' \neq \mathsf{main}_{\mathbb{T}} \wedge C = \varnothing,$$
$$\text{mustNotRace}_{C'}\ (t, C)\ (t', C') \stackrel{\text{def}}{=} t' = \mathsf{main}_{\mathbb{T}} \wedge t \neq \mathsf{main}_{\mathbb{T}} \wedge C' = \varnothing.$$

Equations (6) and (7) technically remain unchanged, but see Remark 4 below for implications.

Flag. While the plain thread ID $\langle u, f\rangle \in \mathcal{P} \subset \mathbb{T}_{\text{plain}}$ may provide helpful diagnostics to GOBLINT users, the mustNotRace definitions do not distinguish it from another $\langle u', f'\rangle \in \mathcal{P}$. Thus, $\mathcal{P}$ can be abstracted to a singleton $\mathsf{other}_{\mathbb{T}}$, yielding even simpler thread IDs $\mathbb{T}_{\text{flag}} \stackrel{\text{def}}{=} \{\mathsf{main}_{\mathbb{T}}, \mathsf{other}_{\mathbb{T}}\}$ with $t \circ \langle u, f\rangle \stackrel{\text{def}}{=} \mathsf{other}_{\mathbb{T}}$. Similarly, the contents of the set C do not really matter, other than whether it is empty or non-empty. Hence, the local states can also be abstracted to a boolean.

Combining these abstractions, only three out of four combinations are relevant:

1. Single-threaded: $t = \mathsf{main}_{\mathbb{T}} \wedge C = \varnothing$.
2. Multi-threaded ($\mathsf{main}_{\mathbb{T}}$): $t = \mathsf{main}_{\mathbb{T}} \wedge C \neq \varnothing$.
3. Multi-threaded ($\mathsf{other}_{\mathbb{T}}$): $t = \mathsf{other}_{\mathbb{T}}$.

In fact, GOBLINT has included since 2008 a *thread flag* domain which has precisely these three values and uses them for primitive race analysis (to exclude single-threaded accesses and $\mathsf{main}_{\mathbb{T}}$ from racing with itself).

Remark 4 (Joining $\mathsf{main}_{\mathbb{T}}$). In these simplified domains, the only unique thread is $\mathsf{main}_{\mathbb{T}}$. Thus, if in Eqs. (6) and (7) only unique threads can be considered must-joined, then the set J with only two possible values ($J = \varnothing$ or $J = \{\mathsf{main}_{\mathbb{T}}\}$) can also be abstracted to a boolean, indicating whether the main thread has been joined or not.

However, GOBLINT's thread flag domain does not include this information because it is of questionable value. While it is technically possible to join the main thread in a secondary thread, it involves an obscure coding pattern:

```
1  int g = 0;
2  tid maintid;

3  void main() {
4    maintid = self();
5    create(t);
6    g++;
7  }

8  void t() {
9    join(maintid);
10   g++;
11 }
```

where the special function `self` returns the ID of the current thread. □

3.6. Related Work

Abstract Interpretation. Instead of an extensive overview on abstract interpretation, we mention other noteworthy general-purpose abstract interpreters for C programs:

ASTRÉE is a proprietary industrial abstract interpreter for proving the absence of runtime errors [Bla+03; Cou+05; Cou+09]. It also analyzes concurrent programs and reports data races. It can additionally analyze C++ programs, including a mixture of C and C++.

FRAMA-C is an industrial-strength, extensible and collaborative framework consisting of many plugins [KPS24]. In particular, the EVA plugin performs abstract interpretation of sequential programs to prove the absence of runtime errors. The MTHREAD plugin builds an analysis of concurrent programs on top of EVA. It can infer protecting mutexes of global variables and report data races, albeit with a bounded number of threads and mutexes [Yak+25, p. 23]. FRAMA-C is largely open source (including EVA and recently MTHREAD), although some plugins are closed source. It unofficially participates in SV-COMP using a third-party wrapper [BS22a].

IKOS is an abstract interpreter developed by NASA [Bra+14]. It does not currently analyze concurrent programs. It can also analyze C++ programs.

INFER is a static analyzer developed by Meta [Dis+19]. It includes a data race analysis for C, C++, C# and JAVA programs. It unofficially participates in SV-COMP using a third-party wrapper [KL22]. Unlike others, INFER is not sound, nor does it aim to be.

MOPSA is a "Modular Open Platform for Static Analysis", which is research-oriented and open source [MOM23]. It does not currently analyze concurrent programs. MOPSA officially participates in SV-COMP. It can also analyze PYTHON programs, including a mixture of C and PYTHON.

POLYSPACE (CODE PROVER) is a proprietary industrial abstract interpreter for proving the absence of runtime errors [Deu03]. It also analyzes concurrent programs and reports data races. It can additionally analyze C++ and ADA programs.

Out of these, ASTRÉE, FRAMA-C, INFER and MOPSA are also written in OCAML, like GOBLINT. Table 6 provides a simple comparison of the tools.

Data Race Analysis. Instead of an extensive overview on data race analysis, we mention the techniques used by other tools in SV-COMP (as seen in Table 4):

CPACHECKER uses explicit-state model checking for data races, but has a broad portfolio overall [Bai+24].

CSEQ reduces data races to assertion safety and performs bounded model checking via lazy sequentialization [Cot+22].

DARTAGNAN performs bounded model checking under a fully configurable (weak) memory model [Pon+20].

Table 6. Abstract interpreters for C programs

Tool	Sound	Concurrency	Open source	SV-COMP
ASTRÉE	✓	✓	✗	✗
FRAMA-C	✓	✓	✓	✓✗
GOBLINT	✓	✓	✓	✓
IKOS	✓	✗	✓	✗
INFER	✗	✓	✓	✓✗
MOPSA	✓	✗	✓	✓
POLYSPACE	✓	✓	✗	✗

DEAGLE encodes the program into ordering formulas which are checked using a custom SMT solver [HSF22].

ESBMC-KIND reduces data races to assertion safety and performs SMT-based context-bounded model checking with partial order reduction [Wu+25].

LOCKSMITH uses a type-based label-flow to infer access-mutex correlation constraints [PFH11].

RACERF uses a heuristic combination of under- and over-approximating abstract interpretation [DV25a; DV25b]. It is built using FRAMA-C.

SV-SANITIZERS compiles the program with THREADSANITIZER (TSAN) instrumentation and executes it (repeatedly with different non-deterministic inputs).[24] Since most data race benchmarks in SV-COMP are fairly small, it serves as a basic sanity check for maintaining the benchmarks. This tool is by the thesis author.

THETA uses a portfolio of model checking approaches [Tel+25].

ULTIMATE family tools reduce data races to assertion safety [Die+23] and perform automata-based model checking with the following twists:

- **UAUTOMIZER** uses SMT solving on traces [Hei+23].
- **UGEMCUTTER** additionally uses commutativity to handle Mazurkiewicz-equivalent traces (elaborated in Section 6.7.2).
- **UTAIPAN** uses symbolic abstract interpretation on naïvely sequentialized path programs [Die+23].

Thread IDs. The thread ID analysis perhaps most closely related to GOBLINT's is by Feret [Fer05], who computes IDs for agents in the π-calculus as abstractions of sequences of encountered create arcs. Another line of analysis of concurrent programs deals with determining which critical events may happen in parallel (MHP) [NAC99; Bar06; Aga+07; Di+15; Zho+18; AFG12; AGG15; AGG17] to detect programming errors (e.g., data races) or identify opportunities for optimization.

[24] `https://github.com/sim642/sv-sanitizers`

GOBLINT in the Wild. Participating in SV-COMP has made it easy to include GOBLINT in independent comparative evaluations [Hei+24; DV25b], even on non–SV-COMP programs [DV25b]. Its integration with BENCHEXEC for the competition provides a standardized interface for executing the tool. This has allowed GOBLINT to be incorporated, without assistance from us, into metaverifiers based on portfolios [BKR22] and cooperation [BSU22; Arm+25]. Moreover, Hong and Ryu [HR23] have used GOBLINT within their CONCRAT tool, and Török [Tör23] has built new analyses using the GOBLINT framework.

3.7. Conclusion and Future Work

GOBLINT has established itself as a general abstract interpretation framework for C programs. Despite specializing in data race analysis, its extensibility and flexibility allow it to support every property considered in SV-COMP. It has been successful in the competition, highlighting its soundness, efficiency and data race analysis.

Future Work. GOBLINT will never be complete (pun intended): there is an ever-growing number of ways to improve it, as evidenced by its issue tracker.[25] The following describes some which particularly relate to SV-COMP and data race analysis.

Autotuning. GOBLINT offers a lot more precision configuration than what is currently exploited by the autotuning described in Section 3.4.3. For example:

- Chapter 5 describes many concurrent value analyses which are also implemented in GOBLINT, but only the fastest and least precise of these are used in SV-COMP: Protection-Based Reading for non-relational analysis and Mutex-Meet for relational analysis.
- Many expressive (but also expensive) relational domains are available (both from APRON and directly in GOBLINT), but the current choice is relatively fixed: polyhedra are used for the termination analysis, octagons for everything else.

Designing appropriate heuristics for additional features would allow using GOBLINT's existing abilities to a fuller extent. While the current autotuning in GOBLINT is handcrafted, machine learning may provide additional improvements.

Portfolio. Autotuning could be extended to supply a sequence of configurations, increasing in precision, for a portfolio of analyses, instead of relying on the autotuning to immediately pick the most appropriate configuration. If the analysis with one configuration is inconclusive, then GOBLINT could move on to the next one. This approach has been very successful in SV-COMP [Wen13; Bai+24; MOM23; MOM25].

[25] `https://github.com/goblint/analyzer/issues`

Violations. The major weakness of GOBLINT is its inability to report definite specification violations. In conjunction with a sequential portfolio, integrating counterexample-guided abstraction refinement (CEGAR) into our framework might allow GOBLINT to also violations, while avoiding false alarms and gaining more precision. Milanese and Miné [MM24] have explored under-approximating abstract interpretation for generating violation witnesses. While sound and terminating abstract interpretation cannot be made complete in general, just covering some simple cases could be impactful in SV-COMP. Dacík and Vojnar [DV25a; DV25b] have successfully used a heuristic combination of under- and over-approximating abstract interpretation for data race analysis (see RACERF in Table 4e).

Data Race Interference Graph. A connected component of accesses in the data race interference graph (Section 3.5.1) may be sparse; in the extreme case a *star* graph: one access (e.g., without mutexes) which may race with all others (e.g., with mutexes), the latter of which being pairwise race-free. In such a case, it is desirable to identify the large independent set of race-free accesses and present them to the user as such. Aforementioned temporal race exclusions offer many more similar sparse situations (including in their combination). Although semantic techniques could be devised to identify them, a purely algorithmic one would be to find maximal independent sets by vertex coloring. This is somewhat analogous to the classic application of inference graph vertex coloring to the register allocation problem. However, self-loops require special attention because standard vertex coloring cannot deal with them: a vertex should have a different color from itself. It is future work to implement this approach in GOBLINT and study its efficiency and usability.

Thread IDs. Our history-based thread IDs (Section 3.5.2) are a significant improvement over plain thread IDs (Section 3.5.4), but further improvements may be considered. In particular, the more general loop-free call strings from Remark 3 (p. 70) could be adapted for use as thread IDs as well. This requires defining suitable unique, mustBeAncestor and mayBeAncestor predicates.

Joined Threads. The exclusion of races with joined threads (Section 3.5.3) crucially depends on the analysis which determines the set J of definitely joined threads. As mentioned in Footnote 23 (p. 71), GOBLINT can currently only prove unique threads to be must-joined. This is a severe limitation because real-world programs use thread pools (non-unique threads created in a loop), which are much more difficult to show to be definitely entirely joined [Hol+25]. Thus, more advanced thread-joinedness analyses are needed to exclude such races in practice.

4. WITNESS-GUIDED ABSTRACT INTERPRETATION

This chapter turns to the first branch of the thesis: *witnesses*. Specifically, it addresses goals **G1** and **G3** to improve the precision and the performance of abstract interpretation by guiding it with invariants from witnesses.

Existing witness validators are based on model checking [Bey+22], test execution [Bey+18], interpretation [ŠBK20; ACS22], or SMT-based verification [PHM22]; whereas validators for correctness witnesses in SV-COMP 2023 were all based on model checking [Bey+16; Bey23]. In that year's competition report, the long-time organizer Dirk Beyer highlighted the scarcity of validators — which leaves many verification outcomes without independent confirmation — as a "remarkable gap in software-verification research" [Bey23]. To complement existing validators, we propose enhancing the framework of abstract interpretation to incorporate invariants from witnesses and, thus, to *meaningfully* validate them. It brings witness-based trust from model checking to abstract interpretation and allows leveraging the scalability of the latter for witness validation, reducing technology bias.

4.1. Introduction

For communication across technological boundaries, correctness witnesses must be restricted to invariants that do not expose internal abstractions of tools. For example, as each tool may abstract dynamically allocated memory differently, invariants about the content of such memory may only be expressed indirectly, e.g., via C invariants such as $*\mathtt{p} \geq 0$. The challenge is how to incorporate such tool-independent invariants into an abstract interpreter. A key technical contribution of this chapter are techniques to incorporate witness invariants, given as expressions, into abstract domains, without relying on those invariants to actually hold. This differs from existing work on witnesses for abstract interpretation (see Section 4.8), which does not allow for or aim at the exchange of witnesses across tool boundaries.

Our solution is to introduce a new abstract operation unassume. This operator can be used to selectively inject imprecision (hence the name) to speed up fixpoint computation. Suitably increasing abstract values during fixpoint computation can also improve the precision of an existing analysis, most notably due to the non-monotonicity of widening operators. The following example illustrates both the speedup and the increase in precision.

Example 8. Consider the program in Listing 3 from Miné [Min17]. An abstract interpreter using interval abstraction first reaches the loop head on line 3 with the interval $[40, 40]$ for x. After one iteration, the loop head is reached with $[39, 39]$, so the abstract value at that point is $[39, 40]$. To accelerate fixpoint iteration for termination, standard interval widening is applied, which abandons the unstable lower bound, resulting in $[-\infty, 40]$. Another iteration with this interval reaches

Listing 3. Program for illustrating witness-guided abstract interpretation [Min17, Example 4.6]

```
1 void main() {
2   int x = 40;
3   while (x != 0) {
4     x--;
5   }
6 }
```

Table 7. Analysis progress and result (highlighted) for the program in Listing 3 from scratch

Line	x interval	Comment
2	$\top$	
3	$[40, 40]$	
4	$[40, 40]$	
5	$[39, 39]$	
3	$[-\infty, 40]$	$[40, 40] \nabla ([40, 40] \sqcup [39, 39])$
4	$[-\infty, 40]$	
5	$[-\infty, 39]$	
3	$[-\infty, 40]$	$[-\infty, 40] \nabla ([40, 40] \sqcup [-\infty, 39])$
3	$[-\infty, 40]$	$[-\infty, 40] \Delta ([40, 40] \sqcup [-\infty, 39])$
6	$[0, 0]$	

the loop head with $[-\infty, 39]$, which is subsumed by the previous abstract value. Subsequent standard interval narrowing cannot improve the inferred invariant $[-\infty, 40]$ at the loop head; therefore, the analysis fails to establish a lower bound for x. Table 7 lists the abstract evaluation steps of this analysis in order. □

Example 9. Now, suppose that a witness provides the invariant $0 \leq x \leq 40$ for the loop head on line 3. When guiding the fixpoint iteration with this witness, the loop head is again first reached with the singleton interval $[40, 40]$. Using the provided invariant, the unassume operator relaxes the lower bound of the interval:

$$[\![\mathsf{unassume}(0 \leq x \leq 40)]\!]^\sharp \{x \mapsto [40, 40]\} = \{x \mapsto [0, 40]\}.$$

After one iteration, the loop head is now reached with $[0, 39]$, which makes the abstract value at that point $[0, 40]$. Thus, a fixpoint is reached without the need for widening or narrowing, and the stronger invariant $[0, 40]$ is confirmed. Table 8 lists the analysis steps when using the unassume operator. □

This demonstrates how the same analysis, when guided, can validate an invariant that it could not infer on its own. A well-chosen witness invariant can prevent precision loss during widening that cannot be recovered by narrowing, serving as a

Table 8. Analysis progress and result (highlighted) for the program in Listing 3 unassuming invariant $0 \leq \mathsf{x} \leq 40$ on line 3

Line	x interval	Comment
2	$\top$	
3	$[0, 40]$	$[\![\mathsf{unassume}(0 \leq \mathsf{x} \leq 40)]\!]^\sharp \{\mathsf{x} \mapsto [40, 40]\}$
4	$[1, 40]$	
5	$[0, 39]$	
3	$[0, 40]$	$[0, 40] \nabla ([40, 40] \sqcup [0, 39])$
3	$[0, 40]$	$[0, 40] \Delta ([40, 40] \sqcup [0, 39])$
6	$[0, 0]$	

proxy for providing known widening thresholds. Additionally, the witness-guided analysis required fewer steps (transfer function evaluations and fixpoint iterations). Using the same invariant as a widening threshold does not yield such a speedup.

After introducing relevant terms (Section 4.2) and discussing the shortcomings of intuitive approaches to validate witnesses with abstract interpretation (Section 4.3), the chapter presents the following main contributions:

- a specification of the unassume operator, and a general realization for relational abstract domains using dual-narrowing (Section 4.4);
- an efficient algorithm for unassuming in non-relational abstract domains (Section 4.5), with generalization to pointer variables (Section 4.5.4);
- an implementation of an abstract-interpretation–based witness validator (Section 4.6), which is evaluated using handcrafted invariants for multi-threaded programs and invariants produced by state-of-the-art model checkers for intricate literature examples (Section 4.7).

Our evaluation results provide practical evidence of the unassumed witness invariants making the analysis faster and more precise.

4.2. Preliminaries

For the following, recall the notion of location-based correctness witnesses from Section 2.5.2, subsequently referred to simply as *witnesses*, In particular, a witness (W, P, Φ) for a safety property Φ of a program P is valid (Definition 11) if

1. P satisfies the property Φ;
2. whenever the execution of P reaches the location $n \in \mathcal{N}$, the invariant $W\,n$ holds.

For the examples in this chapter, we enrich the invariant language of arithmetic and boolean expressions with basic pointers, address-taking (&) and dereferencing ($\star$). The pointer constructs pose practical challenges, as will become apparent in the subsequent sections.

For *validating* a witness by abstract interpretation (Section 2.4), we assume that the analyzer provides us with a mapping $\sigma : \mathcal{N} \to \mathbb{D}$ from locations to abstract values. A witness (W, P, Φ) is validated by the abstract interpreter, if

1. σ is sufficient to verify that Φ holds for program P;
2. for each $n \in \mathcal{N}$, the invariant $W\,n$ is true in every state of $\gamma\,(\sigma\,n)$.

In practice, the second condition may not be easy to check since computing γ is not always feasible. Thus, abstract expression evaluation is used instead to perform the validity check, although this is possibly less precise, as the following example shows.

Example 10. Assume the non-relational abstract domain $\mathbb{D} = \mathcal{V} \to \mathbb{V}$ of environments, where $\mathbb{V}$ is the abstract domain of individual values. Using intervals for the latter, let $d = \{\mathsf{x} \mapsto [1, 2]\}$ be the computed abstract state at some program location where $\gamma\,d = \{\{\mathsf{x} \mapsto 1\}, \{\mathsf{x} \mapsto 2\}\}$ is *exact*. Consider the validation of the following two logically equivalent invariants at this location:

1. The invariant $1 \leq \mathsf{x} \wedge \mathsf{x} \leq 2$ holds for each concrete state in $\gamma\,d$. It also evaluates abstractly to true on d using standard syntax-driven evaluation (Section 4.5), because both conjuncts are true for the interval $[1, 2]$.
2. The invariant $\mathsf{x} = 1 \vee \mathsf{x} = 2$ also holds for each concrete state in $\gamma\,d$. However, when evaluated abstractly and syntax-driven on d, it evaluates to an unknown boolean, because both disjuncts evaluate to an unknown boolean for the interval $[1, 2]$.[26] □

Hence, the abstract interpreter is not complete, i.e., it may fail to validate witnesses which are indeed valid, due to imprecision arising from abstraction or fixpoint acceleration. Nevertheless, validating witnesses using abstract expression evaluation is sound, i.e., all witnesses claimed to be validated by the abstract interpreter are indeed valid.

A witness which maps all locations to the invariant e_{true} is called *trivial*. The validation of such a witness trivially passes the second validity condition, and checking of the first condition falls entirely to the analyzer itself. To be useful, a witness has to be non-trivial and aid the analyzer in proving that the program P satisfies the property Φ, either by improving the precision or the performance of the verification process. For *witness-guided verification*, it suffices if the analyzer can show that Φ holds — even if the invariants of the witness cannot be validated. Given a witness (W, P, Φ), the challenge of witness-guided abstract interpretation is to simultaneously achieve the following:

1. to use the invariants in W to reach a fixpoint $\sigma_W : \mathcal{N} \to \mathbb{D}$ in fewer iterations;
2. to avoid overshooting the required property, i.e., σ_W suffices to prove Φ;

[26] Actually, it is possible to define more precise evaluation to handle this case (see Remark 8, p. 95), but we do not consider it in this thesis.

3. to *not trust* the witness, i.e., σ_W should remain sound even in presence of *wrong* invariants.

Subsequently, we first consider some intuitive approaches to motivate our unassume operator for soundly speeding up abstract interpretation with the help of untrusted witnesses.

4.3. Initialization-Based Approaches

Given a witness (W, P, Φ), one natural idea is to extract from the mapping $W : \mathcal{N} \to \mathcal{I}$ a mapping $w : \mathcal{N} \to \mathbb{D}$ of initial abstract values as non-$\bot$ starting points for constructing inductive invariants for the program. We discuss two approaches for realizing this idea, along with their shortcomings.

Total Initial Values. In the first approach, the initial value $w\,n$ for a program location n is chosen such that $\gamma\,(w\,n)$ includes every concrete state where $W\,n$ holds. For example, by directly choosing $w\,\ell_3 = \{\mathsf{x} \mapsto [0, 40]\}$ in Example 9, instead of using the unassume operator. Such a value, however, is only suitable if *all* relevant information for program location n is formalized in the invariant $W\,n$ and expressible by the abstract domain. These requirements limit the applicability, scalability and practicality of this approach.

For example, consider the invariant $*\mathsf{p} \geq 0$ involving a pointer p dereference for some program location n. It provides no information about which variables p may point to, thus, nothing can be concluded about any integer variables it intends to describe. Therefore, $w\,n = \top$ which leads to a complete loss of precision at location n during the analysis.

This approach also makes silent assumptions about the way in which the analyzer computes values, namely how such initial abstract values are incorporated into analysis, if at all. For example, TD fixpoint solvers [SV21] only use the initial values for starting fixpoint iteration at *dynamically* identified widening points. Additionally, context-sensitive interprocedural analysis is known to give rise to *infinite* constraint systems [ASV12], requiring dedicated changes to the analyzer to ensure that all accessed constraint variables associated with a given location are appropriately initialized.

Partial Initial Values. In order to remedy the problem that all relevant information must be provided in the invariants for program locations, one may instead rely on *partial* initialization. For that to work, we require here a non-relational abstract domain $\mathbb{D} = \mathcal{V} \to \mathbb{V}$, which does not reduce to $\bot$ if the value of any variable is $\bot$. Assuming that the invariant $W\,n$ only speaks of variables from $V \subseteq \mathcal{V}$, the partial initial value is the same as the total initial value $w\,n$, except all unmentioned variables $x \in \mathcal{V} \setminus V$ are assigned $\bot$. In order to exploit the witness, the complete state reaching the location n is *joined* with the partial state constructed from the witness in the corresponding transfer function.

Example 11. Consider for a particular program location n, two integer variables $\mathtt{i}$ and $\mathtt{j}$ and a pointer variable $\mathtt{p}$. Let the abstract domain $\mathbb{V}$ of values consist of intervals for abstracting integers and points-to sets for abstracting pointers. Consider two invariants:

1. The witness invariant $\mathtt{i} \geq 0 \wedge \mathtt{j} \geq 0$ can be represented by the partial state $d_1 = \{\mathtt{p} \mapsto \bot, \mathtt{i} \mapsto [0, \infty], \mathtt{j} \mapsto [0, \infty]\}$.
2. The witness invariant $*\mathtt{p} \geq 0$, on the other hand, results in the partial state $d_2 = \{\mathtt{p} \mapsto \top, \mathtt{i} \mapsto \bot, \mathtt{j} \mapsto \bot\}$.

Now, assume that during the analysis of the program, the complete abstract state $d = \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto [0, 0], \mathtt{j} \mapsto [0, 0]\}$, where $\mathtt{p}$ may point to either $\mathtt{i}$ or $\mathtt{j}$, reaches the program location n. When this value is joined with the partial states corresponding to the two invariants above, we respectively obtain:

1. $d \sqcup d_1 = \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto [0, \infty], \mathtt{j} \mapsto [0, \infty]\}$,
2. $d \sqcup d_2 = \{\mathtt{p} \mapsto \top, \mathtt{i} \mapsto [0, 0], \mathtt{j} \mapsto [0, 0]\}$.

The first may be useful to guide the analysis since the information for $\mathtt{i}$ and $\mathtt{j}$ is maximally relaxed such that the witness invariant can still be validated, while the information for the pointer variable $\mathtt{p}$ is retained. On the other hand, the second state loses all information about $\mathtt{p}$, which is problematic if memory is accessed through $\mathtt{p}$ later in the program. At the same time, here, the values for the variables $\mathtt{i}$ and $\mathtt{j}$ remain overly precise. Instead, one would have liked to obtain the former abstract state also when using the invariant $*\mathtt{p} \geq 0$. □

By joining the initial values within transfer functions, this partial approach is more general than the total one: it works for all program locations regardless of the analysis engine and can be seamlessly applied to infinite constraint systems. Nevertheless, the incorporation of witnesses via partial initial values is only applicable to non-relational domains and cannot depend on analysis state. Therefore, in the next section, we propose a more general solution that overcomes these issues.

4.4. Unassuming

We introduce new primitive statements $\mathsf{unassume}(e)$ to the programming language for all invariants $e \in \mathcal{I}$. Given a witness (W, P, Φ), we insert at every location n, the statement $\mathsf{unassume}(W\, n)$ if it is different from e_{true}. The insertion need not be done at source code level; instead, we may insert the statement at the location into the internal representation used by the analyzer (e.g., the CFG). In the concrete semantics, $\mathsf{unassume}$ statements have no effect, i.e., their arguments are not evaluated and, thus, do not cause runtime errors or undefined behavior.

During the abstract interpretation of the program, the abstract state transformer for the statement $\mathsf{unassume}(e)$ for location n is meant to inject the desired imprecision into the abstract state for n. Intuitively, the abstract semantics of

unassume is dual to the assume operation, i.e., it relaxes a state instead of refining it. Thus, e.g.,

$$\{\mathsf{x} \mapsto [0,\infty]\} \underset{\text{unassume}(\mathsf{x}\geq 0)}{\overset{\text{assume}(\mathsf{x}=0)}{\rightleftarrows}} \{\mathsf{x} \mapsto [0,0]\}. \tag{8}$$

Note that unassume is not the inverse of assume because the used expressions are different. By integrating unassume operations as statements, they can be treated path- and context-sensitively — just like all other statements — if the abstract interpreter supports such sensitivity, yielding a general approach.

4.4.1. Specification

Subsequently, we provide abstract operators $[\![\mathsf{unassume}_V(e)]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ which we use to abstractly interpret the corresponding unassume statement. The abstract operators are parameterized by the set of variables $V \subseteq \mathcal{V}$ whose values are relaxed up to the constraining invariant e. The abstract unassume operator $[\![\mathsf{unassume}_V(e)]\!]^\sharp$ is *sound* if it abstracts the concrete no-op operator, i.e.,

$$\gamma\, d \subseteq \gamma\, ([\![\mathsf{unassume}_V(e)]\!]^\sharp\, d)$$

for all $d \in \mathbb{D}$. In particular, this is the case if the operator is *extensive*, i.e.,

$$d \sqsubseteq [\![\mathsf{unassume}_V(e)]\!]^\sharp\, d.$$

Given that the abstract interpreter is sound w.r.t. the original program, and sound unassume operations are inserted, we conclude that the resulting abstract interpreter is sound w.r.t. the modified program. Since the newly inserted statements have no effect in the concrete, the result remains sound also w.r.t. the original program. This implies the soundness of our validation approach.

Theorem 1 (Sound witness validation)**.** *Assume a witness* (W, P, Φ) *is used to insert* unassume *statements and* $\sigma_W : \mathcal{N} \to \mathbb{D}$ *is the result of analyzing the instrumented program. If the sound analyzer confirms* Φ *and all invariants of* $W : \mathcal{N} \to \mathcal{I}$ *abstractly evaluate to* true *in* σ_W*, then the witness is valid.*

The desired behavior of unassume operators is specified by the following examples.

S1. The intuition from Eq. (8) should be realized:

$$[\![\mathsf{unassume}_{\{\mathsf{x}\}}(\mathsf{x} \geq 0)]\!]^\sharp\, \{\mathsf{x} \mapsto [0,0]\} = \{\mathsf{x} \mapsto [0,\infty]\}.$$

S2. Unmentioned parts of the abstract state should be retained:

$$[\![\mathsf{unassume}_{\{\mathsf{x}\}}(\mathsf{x} \geq 0)]\!]^\sharp\, \{\mathsf{x} \mapsto [0,0], \mathsf{y} \mapsto [0,0]\} = \{\mathsf{x} \mapsto [0,\infty], \mathsf{y} \mapsto [0,0]\}.$$

S3. Information on variables used in the invariant but not contained in V should also be retained:

$$[\![\mathsf{unassume}_{\{\mathtt{i}\}}(\mathtt{i} \le \mathtt{n})]\!]^\sharp \{\mathtt{i} \mapsto [0,0], \mathtt{n} \mapsto [10,10]\} \\ = \{\mathtt{i} \mapsto [-\infty, 10], \mathtt{n} \mapsto [10,10]\}.$$

S4. Conjunctions on the same variable apply jointly:

$$[\![\mathsf{unassume}_{\{\mathtt{x}\}}(0 \le \mathtt{x} \wedge \mathtt{x} \le 40)]\!]^\sharp \{\mathtt{x} \mapsto [40,40]\} = \{\mathtt{x} \mapsto [0,40]\}.$$

This is how the invariant from Example 9 (p. 78) with $V = \{\mathtt{x}\}$ would likely be represented in practice.

S5. Conjunctions on different variables apply independently:

$$[\![\mathsf{unassume}_{\{\mathtt{x},\mathtt{y}\}}(\mathtt{x} \ge 0 \wedge \mathtt{y} \ge 0)]\!]^\sharp \{\mathtt{x} \mapsto [0,0], \mathtt{y} \mapsto [0,0]\} \\ = \{\mathtt{x} \mapsto [0,\infty], \mathtt{y} \mapsto [0,\infty]\}.$$

S6. Disjunctions on the same variable account for all cases:

$$[\![\mathsf{unassume}_{\{\mathtt{x}\}}(\mathtt{x} = 1 \vee \mathtt{x} = 3)]\!]^\sharp \{\mathtt{x} \mapsto [1,1]\} = \{\mathtt{x} \mapsto [1,3]\}.$$

S7. Relational invariants between the relaxed and not relaxed variables should be preserved whenever possible without restricting the unassumed invariant; e.g., relaxing the state $0 = \mathtt{x} \le \mathtt{y}$ (visualized in Fig. 7a) with $0 \le \mathtt{x}$ should result in $0 \le \mathtt{x} \le \mathtt{y}$ (visualized in Fig. 7b):

$$[\![\mathsf{unassume}_{\{\mathtt{x}\}}(\mathtt{x} \ge 0)]\!]^\sharp \{\mathtt{x} \le 0, -\mathtt{x} \le 0, -\mathtt{y} \le 0, -\mathtt{x} - \mathtt{y} \le 0, \mathtt{x} - \mathtt{y} \le 0\} \\ = \{-\mathtt{x} \le 0, -\mathtt{y} \le 0, -\mathtt{x} - \mathtt{y} \le 0, \mathtt{x} - \mathtt{y} \le 0\}$$

when using the octagon domain [Min06].[27] More specifically, this is the most precise result which, when restricted to V, still includes (here equals) the abstract state $\{-\mathtt{x} \le 0\}$, defined only by the unassumed invariant on V.

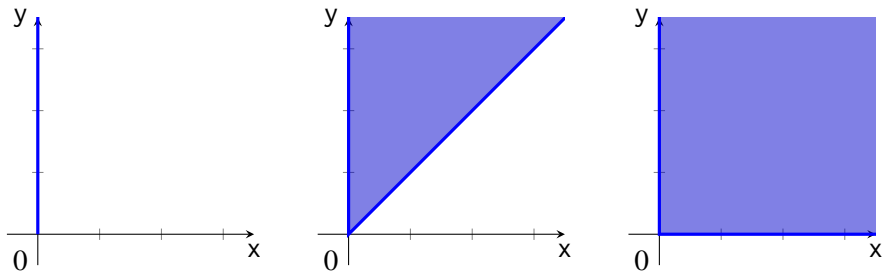
S8. Information provided by the input abstract state should be leveraged to propagate imprecision to further variables not mentioned in the invariant, e.g., via pointers:

$$[\![\mathsf{unassume}_{\{\mathtt{i}\}}(*\mathtt{p} \ge 0)]\!]^\sharp \{\mathtt{p} \mapsto \{\&\mathtt{i}\}, \mathtt{i} \mapsto [0,0]\} \\ = \{\mathtt{p} \mapsto \{\&\mathtt{i}\}, \mathtt{i} \mapsto [0,\infty]\}.$$

S9. Imprecision should be propagated via ambiguous pointers (like in Example 11):

$$[\![\mathsf{unassume}_{\{\mathtt{i},\mathtt{j}\}}(*\mathtt{p} \ge 0)]\!]^\sharp \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto [0,0], \mathtt{j} \mapsto [0,0]\} \\ = \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto [0,\infty], \mathtt{j} \mapsto [0,\infty]\}.$$

[27] Redundant constraints are grayed out. They can be derived from non-redundant (non–grayed-out) constraints using the octagon closure algorithm.



(a) $0 = \mathsf{x} \leq \mathsf{y}$ **(b)** $0 \leq \mathsf{x} \leq \mathsf{y}$ **(c)** $0 \leq \mathsf{x}, \mathsf{y}$

Figure 7. Plots of relations for unassume specification **S7**

S10. Relaxation of one variable (`i` or `j`) may depend on the relaxation of another (`p`):

$$[\![\mathsf{unassume}_{\{\mathsf{p},\mathsf{i},\mathsf{j}\}}((\mathsf{p} = \&\mathsf{i} \vee \mathsf{p} = \&\mathsf{j}) \wedge *\mathsf{p} \geq 0)]\!]^\sharp \{\mathsf{p} \mapsto \{\&\mathsf{i}\}, \mathsf{i} \mapsto [0,0], \mathsf{j} \mapsto [0,0]\} = \{\mathsf{p} \mapsto \{\&\mathsf{i}, \&\mathsf{j}\}, \mathsf{i} \mapsto [0,\infty], \mathsf{j} \mapsto [0,\infty]\}.$$

This combines specifications **S6** and **S9**.

S11. Unrepresentable invariants should not cause excessive loss of precision:

$$[\![\mathsf{unassume}_{\{\mathsf{x}\}}(\mathsf{x} \bmod 2 = 0)]\!]^\sharp \{\mathsf{x} \mapsto [0,0]\} = \{\mathsf{x} \mapsto [0,0]\}$$

instead of $\{\mathsf{x} \mapsto \top\}$, because intervals cannot represent evenness.[28]

We remark that specifications **S3**, **S8** and **S9** illustrate cases where V differs from the set of variables syntactically occurring in e. Since the behavior with relational invariants (**S7**) depends on the expressivity and representation of the *abstract* domain, we do not characterize the operator on sets of concrete states.

We now turn to designing unassume operators to fulfil these specifications. For reference, Table 9 presents a summary of which unassume operators support which specifications.

4.4.2. Naïve Definition

We present the first unassume operator in terms of the abstract operators for non-deterministic assignments ($\mathsf{havoc}(V)$ where $V \subseteq \mathcal{V}$) and guards ($\mathsf{assume}(e)$ where $e \in \mathcal{E}$). In this section, we assume the invariant language $\mathcal{I}$ is a subset of the side-effect–free expressions $\mathcal{E}$ used for conditional branching in the programming language. Let $[\![\mathsf{havoc}(V)]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ and $[\![\mathsf{assume}(e)]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ be the respective sound abstract operators.

[28] This would be exactly representable using the congruence domain [Gra89]:

$$[\![\mathsf{unassume}_{\{\mathsf{x}\}}(\mathsf{x} \bmod 2 = 0)]\!]^\sharp \{\mathsf{x} \mapsto 0\} = \{\mathsf{x} \mapsto 2\mathbb{Z}\}.$$

Table 9. Supported specifications for unassume operators: supported (✓) or not supported (✗)

Specification	Naïve (Definition 12)	Dual-narrowing (Example 12)	Exploding (Definition 14)	Propagating (Section 4.5.2)
S1	✓	✓	✓	✓
S2	✓	✓	✓	✓
S3	✓	✓	✓	✓[a]
S4	✓	✓	✓	✓
S5	✓	✓	✓	✓
S6	✓	✓	✓	✓
S7	✗	✓	✗	✗
S8	✓	✓	✓	✓
S9	✗	✗	✓	✓
S10	✗	✗	✗	✓[b]
S11	✗	✗	✗	✗

[a] Only with a minor modification described in Remark 10 (p. 98).
[b] Depending on variant, see Table 10.

Definition 12 (Naïve unassume)**.** Let $V \subseteq \mathcal{V}$, $e \in \mathcal{I}$ and $d \in \mathbb{D}$. Then the *naïve* unassume is defined as

$$[\![\mathsf{unassume}_V(e)]\!]^\sharp_1\, d \stackrel{\text{def}}{=} d \sqcup ([\![\mathsf{assume}(e)]\!]^\sharp \circ [\![\mathsf{havoc}(V)]\!]^\sharp)\, d. \qquad \lrcorner$$

Intuitively, the argument state is relaxed by joining with an additional value. This value is obtained by first forgetting all information about the variables from V and then assuming the information provided by e. Due to the join, this unassume operator is *sound by construction*. The naïve unassume operator is already sufficient to gain the improvements illustrated by Example 9 (p. 78) when choosing $V = \{\mathsf{x}\}$ with $d = \{\mathsf{x} \mapsto [40, 40]\}$:

$$\begin{aligned}
&[\![\mathsf{unassume}_{\{\mathsf{x}\}}(0 \le \mathsf{x} \le 40)]\!]^\sharp_1\, d \\
&\quad = d \sqcup ([\![\mathsf{assume}(0 \le \mathsf{x} \le 40)]\!]^\sharp \circ [\![\mathsf{havoc}(\{\mathsf{x}\})]\!]^\sharp)\, d \\
&\quad = d \sqcup [\![\mathsf{assume}(0 \le \mathsf{x} \le 40)]\!]^\sharp \{\mathsf{x} \mapsto \top\} \\
&\quad = d \sqcup \{\mathsf{x} \mapsto [0, 40]\} \\
&\quad = \{\mathsf{x} \mapsto [0, 40]\}.
\end{aligned}$$

In addition to specification **S4**, this operator already succeeds for many specifications, as shown in Table 9. But it fails when there are relations between the elements of V and $\mathcal{V} \setminus V$, e.g., for specification **S7** with $d = \{\mathsf{x} \le 0, -\mathsf{x} \le 0, -\mathsf{y} \le 0, -\mathsf{x} - \mathsf{y} \le 0,$

$\mathsf{x} - \mathsf{y} \leq 0\}$ (visualized in Fig. 7a):

$$\begin{aligned}
[\![\mathsf{unassume}_{\{\mathsf{x}\}}(\mathsf{x} \geq 0)]\!]_1^\sharp\, d &= d \sqcup ([\![\mathsf{assume}(\mathsf{x} \geq 0)]\!]^\sharp \circ [\![\mathsf{havoc}(\{\mathsf{x}\})]\!]^\sharp)\, d \\
&= d \sqcup [\![\mathsf{assume}(\mathsf{x} \geq 0)]\!]^\sharp \{-\mathsf{y} \leq 0\} \\
&= d \sqcup \{-\mathsf{x} \leq 0, -\mathsf{y} \leq 0, -\mathsf{x} - \mathsf{y} \leq 0\} \\
&= \{-\mathsf{x} \leq 0, -\mathsf{y} \leq 0, -\mathsf{x} - \mathsf{y} \leq 0\}
\end{aligned}$$

(visualized in Fig. 7c). In this case, the octagon constraint $\mathsf{x} - \mathsf{y} \leq 0$ is lost by havocing and cannot be recovered by assuming.

4.4.3. Dual-Narrowing

We will address the above challenge by relying on additional insights from abstract interpretation. Let us recall the term *dual-narrowing*, which is the lattice analogue of Craig interpolation [Cou15]. A dual-narrowing operator $\tilde{\Delta} : \mathbb{D} \to \mathbb{D} \to \mathbb{D}$ returns for every $d_1, d_2 \in \mathbb{D}$ with $d_1 \sqsubseteq d_2$, a value between both of them, i.e., $d_1 \sqsubseteq d_1 \,\tilde{\Delta}\, d_2 \sqsubseteq d_2$.

Using such an operator, we can define an abstract $\mathsf{unassume}$ that, given d, may return an abstract value in the range from d to $[\![\mathsf{unassume}_V(e)]\!]_1^\sharp\, d$:

Definition 13 (Dual-narrowing $\mathsf{unassume}$)**.** Let $\tilde{\Delta} : \mathbb{D} \to \mathbb{D} \to \mathbb{D}$ be a dual-narrowing. Let $V \subseteq \mathcal{V}$, $e \in \mathcal{I}$ and $d \in \mathbb{D}$. Then the *dual-narrowing* $\mathsf{unassume}$ is defined as a wrapper around the naïve $\mathsf{unassume}$:

$$[\![\mathsf{unassume}_V(e)]\!]_2^\sharp\, d \stackrel{\text{def}}{=} d \,\tilde{\Delta}\, [\![\mathsf{unassume}_V(e)]\!]_1^\sharp\, d. \qquad \lrcorner$$

Example 12. A dual-narrowing for relational domains can be defined using *heterogeneous* environments and *strengthening* [JMO19]. In this chapter, let $\operatorname{dom} d \subseteq \mathcal{V}$ denote the environment of the abstract value $d \in \mathbb{D}$. Moreover, let $d|_V$ denote the general environment change (not just restriction) of the abstract value $d \in \mathbb{D}$ to the program variables $V \subseteq \mathcal{V}$, i.e., $\operatorname{dom}(d|_V) = V$. This is achieved by

- existentially quantifying and removing the variables $\operatorname{dom} d \setminus V$,[29]
- adding the unconstrained variables $V \setminus \operatorname{dom} d$.

An environment-aware order $\boxleft$ is defined for $d_1, d_2 \in \mathbb{D}$ by

$$d_1 \boxleft d_2 \quad \stackrel{\text{def}}{\iff} \quad \operatorname{dom} d_1 \subseteq \operatorname{dom} d_2 \wedge d_1 \sqsubseteq d_2|_{\operatorname{dom} d_1}.$$

Let $\boxplus : \mathbb{D} \to \mathbb{D} \to \mathbb{D}$ be an upper bound operator w.r.t. $\boxleft$, such that the resulting environment is minimal, i.e., $\operatorname{dom}(d_1 \boxplus d_2) = \operatorname{dom} d_1 \cup \operatorname{dom} d_2$. Specifically, Journault, Miné, and Ouadjaout [JMO19] define $\boxplus$ as follows. The result of joining d_1 and d_2 in their common environment $\operatorname{dom} d_1 \cap \operatorname{dom} d_2$ is extended to $\operatorname{dom} d_1 \cup \operatorname{dom} d_2$ by adding unconstrained dimensions. A strengthening operator

[29] This half is called projection [Min17; Jou19; JM09] or quantifier elimination [Min17].

refines this result by iteratively adding back constraints from both arguments which would not cause the upper-boundedness w.r.t. $\trianglelefteq$ to be violated. Formally:

$$
\begin{aligned}
d_1 \bowtie d_2 \stackrel{\text{def}}{=} & \textbf{ let } c = \operatorname{dom} d_1 \cap \operatorname{dom} d_2 \textbf{ in} \\
& \textbf{let } d = (d_1|_c \sqcup d_2|_c)|_{\operatorname{dom} d_1 \cup \operatorname{dom} d_2} \textbf{ in} \\
& \textbf{let } C = \text{constraints } d_1 \cup \text{constraints } d_2 \textbf{ in} \\
& \text{strengthening } d\, C\, d_1\, d_2,
\end{aligned}
$$

where

$$
\begin{aligned}
\text{strengthening } d\, C\, d_1\, d_2 \stackrel{\text{def}}{=} & \textbf{ if } C \neq \varnothing \textbf{ then} \\
& \quad \textbf{let } c \textbf{ such that } c \in C \textbf{ in} \\
& \quad \textbf{let } d' = [\![\mathsf{assume}(c)]\!]^\sharp\, d \textbf{ in} \\
& \quad \textbf{let } d'' = \\
& \quad\quad \textbf{if } d_1 \trianglelefteq d' \wedge d_2 \trianglelefteq d' \textbf{ then } d' \\
& \quad\quad \textbf{else } d \\
& \quad \textbf{in} \\
& \quad \text{strengthening } d'' \, (C \setminus \{c\})\, d_1\, d_2 \\
& \textbf{else } d.
\end{aligned}
$$

Note that this definition is not semantic, i.e., the result depends on the constraints representing the arguments and the order they are processed by strengthening. Journault [Jou19, Definition 6.12] proposes an improved $\bowtie_2$ operator that handles constraints whose variables occur in only one of the argument environments more efficiently.

By defining $d_1 \mathbin{\tilde{\Delta}} d_2 \stackrel{\text{def}}{=} d_1 \bowtie d_2|_V$, which is *parameterized* by V, the dual-narrowing $\mathsf{unassume}$ yields the following desired result for specification **S7** with $d = \{\mathsf{x} \leq 0, -\mathsf{x} \leq 0, -\mathsf{y} \leq 0, -\mathsf{x} - \mathsf{y} \leq 0, \mathsf{x} - \mathsf{y} \leq 0\}$ (visualized in Fig. 7a):

$$
\begin{aligned}
[\![\mathsf{unassume}_{\{\mathsf{x}\}}(\mathsf{x} \geq 0)]\!]^\sharp_2\, d &= d \mathbin{\tilde{\Delta}} [\![\mathsf{unassume}_{\{\mathsf{x}\}}(\mathsf{x} \geq 0)]\!]^\sharp_1\, d \\
&= d \mathbin{\tilde{\Delta}} \{-\mathsf{x} \leq 0, -\mathsf{y} \leq 0, -\mathsf{x} - \mathsf{y} \leq 0\} \\
&= d \bowtie \{-\mathsf{x} \leq 0, -\mathsf{y} \leq 0, -\mathsf{x} - \mathsf{y} \leq 0\}|_{\{\mathsf{x}\}} \\
&= d \bowtie \{-\mathsf{x} \leq 0\} \\
&= \{-\mathsf{x} \leq 0, -\mathsf{y} \leq 0, -\mathsf{x} - \mathsf{y} \leq 0, \mathsf{x} - \mathsf{y} \leq 0\}
\end{aligned}
$$

(visualized in Fig. 7b). Although the restriction to V first destroys the relations between V and $\mathcal{V} \setminus V$, the subsequent strengthening join can restore original relations which are compatible with e on V. As shown in Table 9, this operator behaves like the naïve $\mathsf{unassume}$ for other specifications, because they do not involve such relations. □

Remark 5 (No strengthening). Consider a version of $\bowtie$ which does not apply the strengthening operator (or, equivalently, uses a trivial identity one):

$$
\begin{aligned}
d_1 \bowtie' d_2 \stackrel{\text{def}}{=} & \textbf{ let } c = \operatorname{dom} d_1 \cap \operatorname{dom} d_2 \textbf{ in} \\
& (d_1|_c \sqcup d_2|_c)|_{\operatorname{dom} d_1 \cup \operatorname{dom} d_2}.
\end{aligned}
$$

If the dual-narrowing using this is inlined into Definition 13, we obtain:

$$\begin{aligned}[\![\mathsf{unassume}_V(e)]\!]^\sharp_{2'}\, d &\stackrel{\text{def}}{=} d \mathbin{⩏'} ([\![\mathsf{unassume}_V(e)]\!]^\sharp_1\, d)|_V \\ &\vdots \\ &= [\![\mathsf{havoc}(\mathcal{V} \setminus V)]\!]^\sharp([\![\mathsf{unassume}_V(e)]\!]^\sharp_1\, d).\end{aligned}$$

The additional havoc makes it even less precise than the naïve unassume in many cases, including specifications **S2** and **S3**. Therefore, the strengthening operator is not just for refining the result of the naïve unassume, but is also crucial for retaining some basic functionality. □

Remark 6 (Redundant join). If Definition 12 and the strengthening dual-narrowing are inlined into Definition 13, we obtain:

$$[\![\mathsf{unassume}_V(e)]\!]^\sharp_2\, d = d \mathbin{⩏} (d \sqcup ([\![\mathsf{assume}(e)]\!]^\sharp \circ [\![\mathsf{havoc}(V)]\!]^\sharp)\, d)|_V.$$

Instead, we can omit one of the joins:

$$[\![\mathsf{unassume}_V(e)]\!]^\sharp_{2''}\, d \stackrel{\text{def}}{=} d \mathbin{⩏} (([\![\mathsf{assume}(e)]\!]^\sharp \circ [\![\mathsf{havoc}(V)]\!]^\sharp)\, d)|_V.$$

This no longer fits into Definition 13, but is nevertheless a sound unassume operator, because the left argument of the heterogeneous join has the full environment.

Formally, from the definition of ⩏, we have $d \mathbin{⩏} [\![\mathsf{unassume}_V(e)]\!]^\sharp_{2''}\, d$, which by definition gives:

$$\begin{aligned}d &\sqsubseteq ([\![\mathsf{unassume}_V(e)]\!]^\sharp_{2''}\, d)|_{\operatorname{dom} d} \\ &= ([\![\mathsf{unassume}_V(e)]\!]^\sharp_{2''}\, d)|_{\mathcal{V}} \\ &= [\![\mathsf{unassume}_V(e)]\!]^\sharp_{2''}\, d.\end{aligned}$$

Therefore, $[\![\mathsf{unassume}_V(e)]\!]^\sharp_{2''}$ is extensive and sound. □

Remark 7 (Inexact projection). Journault [Jou19, Section 6.5] makes the striking discovery that ⩋ is *unsound* for integer polyhedra. For example, the following holds:

$$\begin{aligned}&\{\mathsf{x} = 3\} \mathbin{⩋} \{\mathsf{x} = 2\mathsf{y}\} \\ &\quad\iff \operatorname{dom}\{\mathsf{x} = 3\} \subseteq \operatorname{dom}\{\mathsf{x} = 2\mathsf{y}\} \wedge \{\mathsf{x} = 3\} \sqsubseteq \{\mathsf{x} = 2\mathsf{y}\}|_{\operatorname{dom}\{\mathsf{x}=3\}} \\ &\quad\iff \{\mathsf{x}\} \subseteq \{\mathsf{x}, \mathsf{y}\} \wedge \{\mathsf{x} = 3\} \sqsubseteq \{\mathsf{x} = 2\mathsf{y}\}|_{\{\mathsf{x}\}} \\ &\quad\iff \mathsf{true} \wedge \{\mathsf{x} = 3\} \sqsubseteq \{\mathsf{x} \mapsto \top\} \\ &\quad\iff \mathsf{true}.\end{aligned}$$

However, intuitively (and formally), it should not: x = 2y in integer polyhedra states that x is even, but that does not include x = 3.

The problem arises because projection is inexact for integer polyhedra. In the above example, evenness of x is forgotten by projection, causing x = 3 to now be

included in the abstraction, while it would not be concretely. This issue is avoided when projection is *exact*, as is the case for octagons and non-integer polyhedra. Alternatively, a sound projection-inclusion operator is needed.

Luckily, this does not break the soundness of the dual-narrowing (unassume) $d_1 \mathbin{\tilde{\Delta}} d_2 = d_1 \mathbin{\bowtie} d_2|_V$ where $d_1 \sqsubseteq d_2$. Even if strengthening unsoundly adds back all the constraints from d_1 and d_2, then we get $d_1 \mathbin{\tilde{\Delta}} d_2 = d_1$, which is sound in our use case. It just prevents the unassume from relaxing the abstract state. □

The naïve unassume also does not satisfy specification **S11** when assume is not precise:

$$[\![\mathsf{unassume}_{\{\mathtt{x}\}}(\mathtt{x} \bmod 2 = 0)]\!]_1^\sharp \{\mathtt{x} \mapsto [0,0]\} = \{\mathtt{x} \mapsto \top\}.$$

Strengthening does not help here because all environments are the same. Technically, the desired result fits into the bounds from Definition 13, so it would be a valid result of *some* dual-narrowing (unassume). However, it is unclear how to define such a dual-narrowing which is not trivial ($d_1 \mathbin{\tilde{\Delta}} d_2 \stackrel{\text{def}}{=} d_1$). We leave it as future work to define an unassume operator fulfilling specification **S11**.

4.4.4. Pointers

We now turn to the unassuming of invariants which include indirection via pointers. The naïve unassume is unable to achieve the desired precision for specification **S9** with $d = \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto [0,0], \mathtt{j} \mapsto [0,0]\}$:

$$\begin{aligned}
&[\![\mathsf{unassume}_{\{\mathtt{i},\mathtt{j}\}}(*\mathtt{p} \geq 0)]\!]_1^\sharp\, d \\
&\quad = d \sqcup ([\![\mathsf{assume}(*\mathtt{p} \geq 0)]\!]^\sharp \circ [\![\mathsf{havoc}(\{\mathtt{i},\mathtt{j}\})]\!]^\sharp)\, d \\
&\quad = d \sqcup [\![\mathsf{assume}(*\mathtt{p} \geq 0)]\!]^\sharp \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto \top, \mathtt{j} \mapsto \top\} \\
&\quad = d \sqcup \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto \top, \mathtt{j} \mapsto \top\} \\
&\quad = \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto \top, \mathtt{j} \mapsto \top\}.
\end{aligned}$$

This is due to both integer variables being havoced and the assume operator not being able to soundly refine via ambiguous points-to sets: concretely, only one of the two integer variables is refined, so it would be unsound to claim both to be (see Section 4.5.4 for details).

Technically, the desired result is between the bounds from Definition 13:

$$d \sqsubseteq \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto [0,\infty], \mathtt{j} \mapsto [0,\infty]\} \sqsubseteq [\![\mathsf{unassume}_{\{\mathtt{i},\mathtt{j}\}}(*\mathtt{p} \geq 0)]\!]_1^\sharp\, d,$$

so it would be a valid result of *some* dual-narrowing (unassume). However, it is not obvious how to define such a sufficiently general dual-narrowing.

To address the disjunctive nature of the ambiguous points-to set, we propose an improved unassume operator. Suppose we are provided a family of mappings $\mathrm{explode}_V(e) : \mathbb{D} \to 2^{\mathbb{D}}$ which explode any given abstract state d into a non-empty finite subset $\mathrm{explode}_V(e)\, d \subseteq \mathbb{D}$ of abstract states where for each resulting

element d' we have $d' \sqsubseteq d$. The *explode* operator can be used to make disjunctive information in abstract states explicit, e.g., resolve non-singleton points-to sets for pointer variables not contained in V.

Definition 14 (Exploding unassume)**.** Let $V \subseteq \mathcal{V}$, $e \in \mathcal{I}$ and $d \in \mathbb{D}$. Let $\text{explode}_V(e)$ be an explode operator. Then the *exploding* unassume is defined as

$$[\![\text{unassume}_V(e)]\!]^\sharp_3\, d \stackrel{\text{def}}{=} \bigsqcap_{d' \in \text{explode}_V(e)\, d} d \sqcup ([\![\text{assume}(e)]\!]^\sharp \circ [\![\text{havoc}(V)]\!]^\sharp)\, d'. \qquad \lrcorner$$

This improved unassume operator is extensive and, therefore, sound for any choice of $\text{explode}_V(e)$. One might want to establish that $\bigsqcup \text{explode}_V(e)\, d = d$ holds, but this is not necessary for soundness; whereas $\text{explode}_V(e)\, d = \{\bot\}$ would make the unassume a no-op. Note that the exploding unassume is not a special case of the dual-narrowing unassume because the havoc-assume composition is applied to each exploded element d', not the argument d.

Example 13. Consider the following explode operator, which splits ambiguous points-to sets:

$$\begin{aligned}\text{explode}_{\{\mathtt{i},\mathtt{j}\}}(*\mathtt{p} \geq 0)\, \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto [0,0], \mathtt{j} \mapsto [0,0]\} \\ = \{\{\mathtt{p} \mapsto \{\&\mathtt{i}\}, \mathtt{i} \mapsto [0,0], \mathtt{j} \mapsto [0,0]\}, \\ \{\mathtt{p} \mapsto \{\&\mathtt{j}\}, \mathtt{i} \mapsto [0,0], \mathtt{j} \mapsto [0,0]\}\}.\end{aligned}$$

Using this explode operator, specification **S9** is handled as desired with $d = \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto [0,0], \mathtt{j} \mapsto [0,0]\}$:

$$\begin{aligned}&[\![\text{unassume}_{\{\mathtt{i},\mathtt{j}\}}(*\mathtt{p} \geq 0)]\!]^\sharp_3\, d \\ &\quad = (d \sqcup ([\![\text{assume}(*\mathtt{p} \geq 0)]\!]^\sharp \circ [\![\text{havoc}(\{\mathtt{i},\mathtt{j}\})]\!]^\sharp) \\ &\qquad\qquad \{\mathtt{p} \mapsto \{\&\mathtt{i}\}, \mathtt{i} \mapsto [0,0], \mathtt{j} \mapsto [0,0]\}) \\ &\qquad \sqcap (d \sqcup ([\![\text{assume}(*\mathtt{p} \geq 0)]\!]^\sharp \circ [\![\text{havoc}(\{\mathtt{i},\mathtt{j}\})]\!]^\sharp) \\ &\qquad\qquad \{\mathtt{p} \mapsto \{\&\mathtt{j}\}, \mathtt{i} \mapsto [0,0], \mathtt{j} \mapsto [0,0]\}) \\ &\quad = (d \sqcup [\![\text{assume}(*\mathtt{p} \geq 0)]\!]^\sharp \{\mathtt{p} \mapsto \{\&\mathtt{i}\}, \mathtt{i} \mapsto \top, \mathtt{j} \mapsto \top\}) \\ &\qquad \sqcap (d \sqcup [\![\text{assume}(*\mathtt{p} \geq 0)]\!]^\sharp \{\mathtt{p} \mapsto \{\&\mathtt{j}\}, \mathtt{i} \mapsto \top, \mathtt{j} \mapsto \top\}) \\ &\quad = (d \sqcup \{\mathtt{p} \mapsto \{\&\mathtt{i}\}, \mathtt{i} \mapsto [0,\infty], \mathtt{j} \mapsto \top\}) \\ &\qquad \sqcap (d \sqcup \{\mathtt{p} \mapsto \{\&\mathtt{j}\}, \mathtt{i} \mapsto \top, \mathtt{j} \mapsto [0,\infty]\}) \\ &\quad = \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto [0,\infty], \mathtt{j} \mapsto \top\} \\ &\qquad \sqcap \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto \top, \mathtt{j} \mapsto [0,\infty]\} \\ &\quad = \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto [0,\infty], \mathtt{j} \mapsto [0,\infty]\}. \qquad \square\end{aligned}$$

As shown in Table 9, this unassume operator, which explodes points-to sets, behaves like the naïve unassume for other specifications, because they do not include ambiguous points-to sets in the argument state.

Example 14. In particular, consider specification **S10**, where different subexpressions depend on each other (here through p):

$$[\![\mathsf{unassume}_{\{\mathtt{p},\mathtt{i},\mathtt{j}\}}((\mathtt{p} = \&\mathtt{i} \vee \mathtt{p} = \&\mathtt{j}) \wedge *\mathtt{p} \geq 0)]\!]^\sharp \\ \{\mathtt{p} \mapsto \{\&\mathtt{i}\}, \mathtt{i} \mapsto [0,0], \mathtt{j} \mapsto [0,0]\}.$$

In contrast to Example 13, there is no ambiguous points-to set in the abstract state supplied as the argument. All possible explosions lead to the same issue as when using the naïve unassume on this example. After havocing, the environment contains $\mathtt{p} \mapsto \top$, thus, in the assume, a top pointer needs to be dereferenced and its targets refined. The semantics of this is unclear and imprecise at best, when one has to consider assignments to *all* possible (unrelated) memory locations. □

4.5. Propagating Algorithm

We now turn to the unassuming of more complex invariants, which include dependent subexpressions and indirection via pointers (e.g., the aforementioned **S10**). The *HC4-revise* algorithm by Benhamou et al. [Ben+99] can be used to implement the assume operation for complex expressions on non-relational domains in a syntax-directed manner [Min17; Zia19]. It is also known as *backward evaluation* [Cou99]. We describe the algorithm and then apply it to construct an unassume operator in Section 4.5.2.

Syntax. We loosely follow the presentation by Cousot [Cou99]. Let the syntax of expressions e and logical conditions c be defined by the grammar

$$\begin{aligned}
e &::= k && \text{(constant)}\\
&\mid x && \text{(variable, } x \in \mathcal{V}\text{)}\\
&\mid \square\,(e)_{i=1}^{n} && (n\text{-ary operator, } n \in \mathbb{N}, n \geq 1, \square \in \mathcal{O}_n),\\
c &::= e \bowtie e && \text{(binary comparison, } \bowtie \in \{=, \neq, <, \leq, >, \geq\}\text{)}\\
&\mid c \wedge c && \text{(conjunction)}\\
&\mid c \vee c && \text{(disjunction).}
\end{aligned}$$

For each $n \in \mathbb{N}$, $n \geq 1$, let $\mathcal{O}_n$ be the set of n-ary operators. For simplicity of presentation, assume that conditions are in negation normal form (NNF), i.e., negations in conditions have been "pushed down" into binary comparisons using the following standard transformations (recursively):

$$\begin{aligned}
&\neg(e_1 = e_2) \stackrel{\text{def}}{=} e_1 \neq e_2, && \neg(e_1 < e_2) \stackrel{\text{def}}{=} e_1 \geq e_2, && \neg(e_1 > e_2) \stackrel{\text{def}}{=} e_1 \leq e_2,\\
&\neg(e_1 \neq e_2) \stackrel{\text{def}}{=} e_1 = e_2, && \neg(e_1 \leq e_2) \stackrel{\text{def}}{=} e_1 > e_2, && \neg(e_1 \geq e_2) \stackrel{\text{def}}{=} e_1 < e_2,\\
&\neg(c_1 \wedge c_2) \stackrel{\text{def}}{=} \neg c_1 \vee \neg c_2, && \neg(c_1 \vee c_2) \stackrel{\text{def}}{=} \neg c_1 \wedge \neg c_2, && \neg(\neg c) \stackrel{\text{def}}{=} c.
\end{aligned}$$

The logical conditions form an invariant language (Section 4.2). The following algorithms generalize from just variables to lvalues, allowing for languages with pointers, like our example invariant language from before. This generalization is formalized in Section 4.5.4.

Evaluation. Let $\mathbb{V}$ be the abstract domain for individual values and $\mathbb{D} = \mathcal{V} \to \mathbb{V}$ the abstract domain for non-relational environments. Let $\mathbb{B}_\bot^\top \stackrel{\text{def}}{=} \{\bot, \mathsf{true}^\sharp, \mathsf{false}^\sharp, \top\}$ be the flat boolean domain, where $\bot \sqsubseteq \{\mathsf{true}^\sharp, \mathsf{false}^\sharp\} \sqsubseteq \top$.[30] The standard abstract forward evaluation of expressions $\mathbb{E}[\![e]\!]^\sharp : \mathbb{D} \to \mathbb{V}$ and conditions $\mathbb{C}[\![c]\!]^\sharp : \mathbb{D} \to \mathbb{B}_\bot^\top$ are defined as

$$\mathbb{E}[\![k]\!]^\sharp\, d \stackrel{\text{def}}{=} k^\sharp, \tag{9a}$$

$$\mathbb{E}[\![x]\!]^\sharp\, d \stackrel{\text{def}}{=} d\, x, \tag{9b}$$

$$\mathbb{E}[\![\square\, (e_i)_{i=1}^n]\!]^\sharp\, d \stackrel{\text{def}}{=} \square^\sharp\, (\mathbb{E}[\![e_i]\!]^\sharp\, d)_{i=1}^n; \tag{9c}$$

$$\mathbb{C}[\![e_1 \bowtie e_2]\!]^\sharp\, d \stackrel{\text{def}}{=} \mathbb{E}[\![e_1]\!]^\sharp\, d \bowtie^\sharp \mathbb{E}[\![e_2]\!]^\sharp\, d, \tag{9d}$$

$$\mathbb{C}[\![c_1 \wedge c_2]\!]^\sharp\, d \stackrel{\text{def}}{=} \mathbb{C}[\![c_1]\!]^\sharp\, d \wedge^\sharp \mathbb{C}[\![c_2]\!]^\sharp\, d, \tag{9e}$$

$$\mathbb{C}[\![c_1 \vee c_2]\!]^\sharp\, d \stackrel{\text{def}}{=} \mathbb{C}[\![c_1]\!]^\sharp\, d \vee^\sharp \mathbb{C}[\![c_2]\!]^\sharp\, d. \tag{9f}$$

For a constant k, let $k^\sharp$ be its corresponding abstraction, and $\square^\sharp$, $\bowtie^\sharp$, $\wedge^\sharp$, $\vee^\sharp$ be abstract versions of the corresponding operators.

4.5.1. Assuming

The HC4-revise algorithm for the assume operation has three phases:[31]

1. Bottom-up forward propagation on the expression tree abstractly evaluates the expression, as usual.
2. Top-down backward propagation refines each abstract value with the expected result of the subexpression. This relies on backward abstract operators, which refine each argument based on the other arguments and the expected result.
3. Bottom-up result construction returns a new refined environment. Variables in the environment are refined at the leaves and refined environments are combined at the intermediate nodes.

[30] $\mathbb{B}_\bot^\top$ is isomorphic to $2^{\mathbb{B}}$: $\bot = \varnothing$, $\mathsf{true}^\sharp = \{\mathsf{true}\}$, $\mathsf{false}^\sharp = \{\mathsf{false}\}$ and $\top = \mathbb{B}$.

[31] There is no general consensus how many phases there are and what each phase does:

- Benhamou et al. [Ben+99] and Ziat [Zia19] describe the first two phases but then refine the environment imperatively in place.
- Miné [Min17] does not describe environment refinement but has an intermediate phase for refining the root.
- Cousot [Cou99] considers the last two phases as one.

The propagating assume is defined as

$$[\![\mathsf{assume}(e)]\!]^\sharp\, d \stackrel{\text{def}}{=} \overleftarrow{\mathbb{C}}[\![e]\!]^\sharp\, d,$$

where the abstract backward evaluation of expressions $\overleftarrow{\mathbb{E}}[\![e]\!]^\sharp : \mathbb{V} \to \mathbb{D} \to \mathbb{D}$ and conditions $\overleftarrow{\mathbb{C}}[\![c]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ are given by

$$\begin{aligned}
\overleftarrow{\mathbb{E}}[\![k]\!]^\sharp\, v'\, d &\stackrel{\text{def}}{=} \textbf{if } k^\sharp \sqsubseteq v' \textbf{ then } d \textbf{ else } \bot,\\
\overleftarrow{\mathbb{E}}[\![x]\!]^\sharp\, v'\, d &\stackrel{\text{def}}{=} d[x \overset{\bot}{\mapsto} d\, x \sqcap v'],\\
\overleftarrow{\mathbb{E}}[\![\square\, (e_i)_{i=1}^n]\!]^\sharp\, v'\, d &\stackrel{\text{def}}{=} \textbf{let } (v_i)_{i=1}^n = (\mathbb{E}[\![e_i]\!]^\sharp\, d)_{i=1}^n \textbf{ in}\\
&\qquad \textbf{let } (v_i')_{i=1}^n = \overleftarrow{\square}^\sharp\, v'\, (v_i)_{i=1}^n \textbf{ in}\\
&\qquad \bigsqcap_{i=1}^n \overleftarrow{\mathbb{E}}[\![e_i]\!]^\sharp\, (v_i \sqcap v_i')\, d;\\
\overleftarrow{\mathbb{C}}[\![e_1 \bowtie e_2]\!]^\sharp\, d &\stackrel{\text{def}}{=} \textbf{let } (v_1, v_2) = (\mathbb{E}[\![e_1]\!]^\sharp\, d, \mathbb{E}[\![e_2]\!]^\sharp\, d) \textbf{ in}\\
&\qquad \textbf{let } (v_1', v_2') = v_1 \overleftarrow{\bowtie}^\sharp\, v_2 \textbf{ in}\\
&\qquad \overleftarrow{\mathbb{E}}[\![e_1]\!]^\sharp\, (v_1 \sqcap v_1')\, d \sqcap \overleftarrow{\mathbb{E}}[\![e_2]\!]^\sharp\, (v_2 \sqcap v_2')\, d,\\
\overleftarrow{\mathbb{C}}[\![c_1 \wedge c_2]\!]^\sharp\, d &\stackrel{\text{def}}{=} \overleftarrow{\mathbb{C}}[\![c_1]\!]^\sharp\, d \sqcap \overleftarrow{\mathbb{C}}[\![c_2]\!]^\sharp\, d,\\
\overleftarrow{\mathbb{C}}[\![c_1 \vee c_2]\!]^\sharp\, d &\stackrel{\text{def}}{=} \overleftarrow{\mathbb{C}}[\![c_1]\!]^\sharp\, d \sqcup \overleftarrow{\mathbb{C}}[\![c_2]\!]^\sharp\, d.
\end{aligned}$$

Instead of evaluating to an abstract value, they refine values of variables in the abstract environment. In the variable case, the function update with $\overset{\bot}{\mapsto}$ reduces the entire map to $\bot \in \mathbb{D}$ if the updated value is $\bot \in \mathbb{V}$. For each $n \in \mathbb{N}$, $n \geq 1$ and $\square \in O_n$, let $\overleftarrow{\square}^\sharp : \mathbb{V} \to \mathbb{V}^n \to \mathbb{V}^n$ be the abstract backward version of the n-ary operator $\square$. It returns abstract values for its arguments under the assumption that the operator evaluates to the given abstract value v' and the other arguments have the given abstract values. For example, if $n = 2$, then

$$\begin{aligned}
\overleftarrow{\square}^\sharp\, v'\, (v_1, v_2) = (v_1', v_2') \implies &\{x_1 \in \gamma\, \mathbb{V} \mid \exists x_2 \in \gamma\, v_2 : \square\, (x_1, x_2) \in \gamma\, v'\} \subseteq \gamma\, v_1'\\
&\wedge \{x_2 \in \gamma\, \mathbb{V} \mid \exists x_1 \in \gamma\, v_1 : \square\, (x_1, x_2) \in \gamma\, v'\} \subseteq \gamma\, v_2'.
\end{aligned}$$

Unlike Cousot [Cou99] and Miné [Min17], we require that the backward operators *do not* intersect an argument's backward-computed value with its current value. Instead, we make this explicit in the algorithm like Benhamou et al. [Ben+99]. Similarly, let $\overleftarrow{\bowtie}^\sharp : \mathbb{V} \to \mathbb{V} \to \mathbb{V} \times \mathbb{V}$ be the abstract backward version of the comparison $\bowtie$. Since conditions are in NNF, the expected result is always $\mathsf{true}^\sharp$ and no v' argument is needed for it. The evaluations $\mathbb{E}[\![e]\!]^\sharp\, d$ should all be cached and reused from a single forward evaluation as the argument environment is passed around without changes [Ben+99; Min17]. The following example shows the algorithm in action.

Example 15. Consider computing $[\![\mathsf{assume}(\mathsf{x} + 2 = 42)]\!]^\sharp\, d$ where $d = \{\mathsf{x} \mapsto [40, \infty]\}$ by this algorithm. First, the condition $\mathsf{x} + 2 = 42$ is evaluated forward

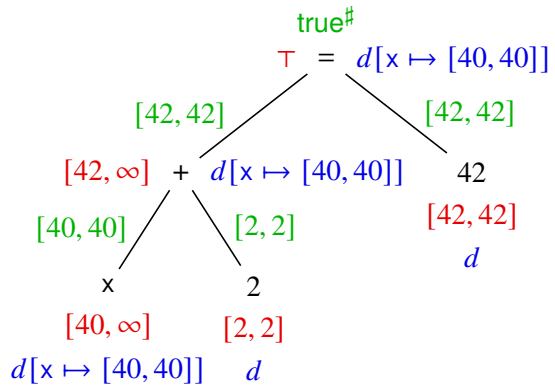


Figure 8. Phases of propagating algorithm for $[\![\mathsf{assume}(\mathsf{x}+2=42)]\!]^\sharp\, d$ where $d = \{\mathsf{x} \mapsto [40,\infty]\}$: forward propagation (in red), backward propagation (in green) and result construction (in blue)

(bottom-up) according to Eq. (9). Figure 8 depicts its syntax tree where each node is labeled with the abstract value of the subexpression.

Second, the expected values of subexpressions are calculated by backward propagation (top-down) using abstract backward evaluation operations. Figure 8 also depicts the expected value of each subexpression (already intersected with its forward-propagated value) on its parent edge. In particular:

$$[42,\infty] \mathbin{\overleftarrow{=}^\sharp} [42,42] = ([42,42],[42,\infty]) \quad (\text{in } \overleftarrow{\mathbb{C}}[\![\mathsf{x}+2=42]\!]^\sharp\, d),$$
$$\overleftarrow{+}^\sharp\,[42,42]\,([40,\infty],[2,2]) = ([40,40],[-\infty,2]) \quad (\text{in } \overleftarrow{\mathbb{E}}[\![\mathsf{x}+2]\!]^\sharp\,[42,42]\,d),$$

which yields the abstract value $[40,40]$ for x.

Third, the resulting abstract state with this refined value is returned (bottom-up), as depicted in Fig. 8. Thus,

$$[\![\mathsf{assume}(\mathsf{x}+2=42)]\!]^\sharp\, d = \{\mathsf{x} \mapsto [40,40]\}. \qquad \square$$

Remark 8 (Joint evaluation and assuming). The forward evaluation of conjunction and disjunction in Eqs. (9e) and (9f) is not as precise as it could be. Instead, consider defining their evaluation using assume:[32]

$$\mathbb{C}[\![c_1 \wedge c_2]\!]^\sharp\, d \stackrel{\text{def}}{=} \mathbb{C}[\![c_2]\!]^\sharp\,([\![\mathsf{assume}(c_1)]\!]^\sharp\, d) \sqcup \mathbb{C}[\![\mathsf{false}]\!]^\sharp\,([\![\mathsf{assume}(\neg c_1)]\!]^\sharp\, d),$$
$$\mathbb{C}[\![c_1 \vee c_2]\!]^\sharp\, d \stackrel{\text{def}}{=} \mathbb{C}[\![\mathsf{true}]\!]^\sharp\,([\![\mathsf{assume}(c_1)]\!]^\sharp\, d) \sqcup \mathbb{C}[\![c_2]\!]^\sharp\,([\![\mathsf{assume}(\neg c_1)]\!]^\sharp\, d),$$

where:

1. $\neg c_1$ is transformed to NNF,

[32] Mopsa uses this technique directly. Goblint uses it indirectly: CIL performs the equivalent transformation for logical operators in the parsed program. However, CIL cannot do so when parsing side-effect–free expressions, so witness validation in Goblint does not get this benefit.

2. $\mathbb{E}[\![e]\!]^\sharp$ and $\mathbb{C}[\![c]\!]^\sharp$ are implicitly strict,
3. $\mathbb{C}[\![\mathsf{false}]\!]^\sharp\, d \stackrel{\text{def}}{=} \mathsf{false}^\sharp$ and $\mathbb{C}[\![\mathsf{true}]\!]^\sharp\, d \stackrel{\text{def}}{=} \mathsf{true}^\sharp$ if $d \neq \bot$.

This makes evaluation and assuming mutually recursive.

It allows the second operand to be evaluated in a refined environment, which incorporates knowledge about the first operand. For example, this handles the imprecise case from Example 10 (p. 80) with $d = \{\mathsf{x} \mapsto [1, 2]\}$:

$$\begin{aligned}
&\mathbb{C}[\![\mathsf{x} = 1 \vee \mathsf{x} = 2]\!]^\sharp\, d \\
&\quad = \mathbb{C}[\![\mathsf{true}]\!]^\sharp\, ([\![\mathsf{assume}(\mathsf{x} = 1)]\!]^\sharp\, d) \sqcup \mathbb{C}[\![\mathsf{x} = 2]\!]^\sharp\, ([\![\mathsf{assume}(\mathsf{x} \neq 1)]\!]^\sharp\, d) \\
&\quad = \mathbb{C}[\![\mathsf{true}]\!]^\sharp\, \{\mathsf{x} \mapsto [1, 1]\} \sqcup \mathbb{C}[\![\mathsf{x} = 2]\!]^\sharp\, \{\mathsf{x} \mapsto [2, 2]\} \\
&\quad = \mathsf{true}^\sharp \sqcup \mathsf{true}^\sharp = \mathsf{true}^\sharp.
\end{aligned}$$

We do not consider this improvement in the rest of this thesis. □

4.5.2. Unassuming

The HC4-revise algorithm can be adapted into the *propagating* unassume operator which is defined as

$$[\![\mathsf{unassume}(e)]\!]^\sharp\, d \stackrel{\text{def}}{=} d \mathbin{⩲} \widetilde{\mathbb{C}}[\![e]\!]^\sharp\, d,$$

where $\widetilde{\mathbb{E}}[\![e]\!]^\sharp : \mathbb{V} \to \mathbb{D} \to \mathbb{D}$ and $\widetilde{\mathbb{C}}[\![c]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ are given by

$$\begin{aligned}
\widetilde{\mathbb{E}}[\![k]\!]^\sharp\, v'\, d &\stackrel{\text{def}}{=} \varnothing, \\
\widetilde{\mathbb{E}}[\![x]\!]^\sharp\, v'\, d &\stackrel{\text{def}}{=} \{x \mapsto v'\}, \\
\widetilde{\mathbb{E}}[\![\square\, (e_i)_{i=1}^n]\!]^\sharp\, v'\, d &\stackrel{\text{def}}{=} \mathbf{let}\ (v_i)_{i=1}^n = (\mathbb{E}[\![e_i]\!]^\sharp\, d)_{i=1}^n\ \mathbf{in} \\
&\qquad \mathbf{let}\ (v_i')_{i=1}^n = \overleftarrow{\square}^\sharp\, v'\, (v_i)_{i=1}^n\ \mathbf{in} \\
&\qquad \left\lfloor \bullet \right\rfloor_{i=1}^n \widetilde{\mathbb{E}}[\![e_i]\!]^\sharp\, v_i'\, d; \\
\widetilde{\mathbb{C}}[\![e_1 \bowtie e_2]\!]^\sharp\, d &\stackrel{\text{def}}{=} \mathbf{let}\ (v_1, v_2) = (\mathbb{E}[\![e_1]\!]^\sharp\, d, \mathbb{E}[\![e_2]\!]^\sharp\, d)\ \mathbf{in} \\
&\qquad \mathbf{let}\ (v_1', v_2') = v_1 \mathbin{\overleftarrow{\bowtie}^\sharp} v_2\ \mathbf{in} \\
&\qquad \widetilde{\mathbb{E}}[\![e_1]\!]^\sharp\, v_1'\, d \mathbin{⩀} \widetilde{\mathbb{E}}[\![e_2]\!]^\sharp\, v_2'\, d, \\
\widetilde{\mathbb{C}}[\![c_1 \wedge c_2]\!]^\sharp\, d &\stackrel{\text{def}}{=} \widetilde{\mathbb{C}}[\![c_1]\!]^\sharp\, d \mathbin{⩀} \widetilde{\mathbb{C}}[\![c_2]\!]^\sharp\, d, \\
\widetilde{\mathbb{C}}[\![c_1 \vee c_2]\!]^\sharp\, d &\stackrel{\text{def}}{=} \widetilde{\mathbb{C}}[\![c_1]\!]^\sharp\, d \mathbin{⩲} \widetilde{\mathbb{C}}[\![c_2]\!]^\sharp\, d.
\end{aligned}$$

Changes (highlighted) are required to achieve the following properties:

Variable set variance. In specification **S10**, the first conjunct should relax {p}, while the second should relax {i, j} via the relaxed pointer. In order to allow different subexpressions to relax different variable sets, the abstract environments returned by $\widetilde{\mathbb{E}}[\![e]\!]^\sharp$ and $\widetilde{\mathbb{C}}[\![c]\!]^\sharp$ are partial: they only contain variables which have been relaxed at the leaves in the corresponding subexpression.

Thus, the heterogeneous join $\veebar$ from Example 12 (p. 87) is used. However, here in the non-relational case, its definition is simpler: values are joined pointwise, while using $\bot$ for missing variables.[33] The heterogeneous meet $\barwedge$ is defined analogously, using $\top$ for missing variables. Note that $\barwedge$ is *not* the meet w.r.t. $\sqsubseteq$, because $\barwedge$ must preserve all relaxed variables from both operands, not just the common ones. Instead, it is a minimal-environment lower bound operator w.r.t. another environment-aware order $\sqsubseteq$, which is defined for $d_1, d_2 \in \mathbb{D}$ by

$$d_1 \sqsubseteq d_2 \quad \overset{\text{def}}{\iff} \quad \operatorname{dom} d_2 \subseteq \operatorname{dom} d_1 \wedge d_1|_{\operatorname{dom} d_2} \sqsubseteq d_2.$$

Soundness. The result is joined with the pre-unassume environment to ensure soundness.

Relaxation. Backward propagation only propagates backward values v_i' and does not refine them using abstract values v_i computed by forward propagation. Otherwise, subexpressions cannot be relaxed at all from their current values. Note that the forward values are still necessary for evaluating the backward operators. This modification on its own yields the HC4-revise* algorithm described by Benhamou et al. [Ben+99].

Unlike our previous unassume operators, this algorithm does not take a set V, but implicitly chooses it to be those variables which are relaxed in the process. This is further discussed in Remark 9 below. We do not give an exact characterization of the result computed by the modified algorithm as it has remained an open problem for HC4-revise itself [Ben+99; GG05].

The following example illustrates the adapted algorithm dually to Example 15:

$$\{\mathsf{x} \mapsto [40, \infty]\} \underset{\text{unassume}(\mathsf{x}+2\geq 42)}{\overset{\text{assume}(\mathsf{x}+2=42)}{\rightleftarrows}} \{\mathsf{x} \mapsto [40, 40]\}.$$

Example 16. Consider computing $[\![\text{unassume}(\mathsf{x} + 2 \geq 42)]\!]^\sharp\, d$ where $d = \{\mathsf{x} \mapsto [40, 40]\}$ by this algorithm. First, the condition $\mathsf{x} + 2 \geq 42$ is evaluated forward (bottom-up) according to Eq. (9). Figure 9 depicts its syntax tree where each node is labeled with the abstract value of the subexpression.

Second, the expected values of subexpressions are calculated by backward propagation (top-down) using abstract backward evaluation operations. Figure 9 also depicts the expected value of each subexpression on its parent edge. In particular:

$$[42, 42] \mathbin{\overleftarrow{\geq}^\sharp} [42, 42] = ([42, \infty], [-\infty, 42]) \quad (\text{in } \overleftarrow{\mathbb{C}}[\![\mathsf{x} + 2 \geq 42]\!]^\sharp\, d),$$
$$\mathbin{\overleftarrow{+}^\sharp}\, [42, \infty]\, ([40, 40], [2, 2]) = ([40, \infty], [2, \infty]) \quad (\text{in } \overleftarrow{\mathbb{E}}[\![\mathsf{x} + 2]\!]^\sharp\, [42, \infty]\, d),$$

which yields the abstract value $[40, \infty]$ for x.

[33] This folk technique served as a starting point for the work of Journault [Jou19, Section 6.3.3].

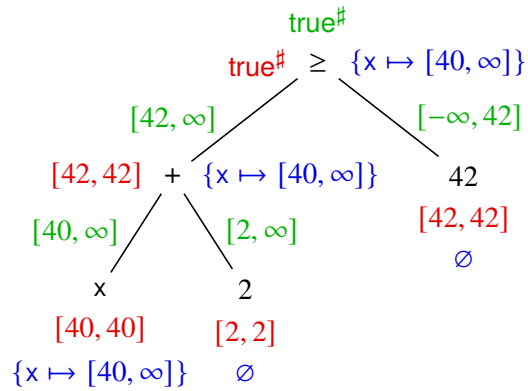


Figure 9. Phases of propagating algorithm for $[\![\mathsf{unassume}(\mathsf{x}+2 \geq 42)]\!]^\sharp\, d$ where $d = \{\mathsf{x} \mapsto [40, 40]\}$: forward propagation (in red), backward propagation (in green) and result construction (in blue)

Third, the resulting abstract state with this refined value is returned (bottom-up), as depicted in Fig. 9. Thus,

$$[\![\mathsf{unassume}(\mathsf{x}+2 \geq 42)]\!]^\sharp\, d = \{\mathsf{x} \mapsto [40, \infty]\}. \qquad \square$$

Remark 9 (Implicit V). Thanks to the variable set variance property, this algorithm implicitly chooses V to be those variables which are relaxed in the process. Moreover, this happens per subexpression, so the propagating unassume gives

$$\begin{aligned}[\![\mathsf{unassume}(\mathsf{x} \geq 0 \vee \mathsf{y} \geq 0)]\!]^\sharp \{\mathsf{x} \mapsto [0,0], \mathsf{y} \mapsto [0,0]\} \\ = \{\mathsf{x} \mapsto [0,\infty], \mathsf{y} \mapsto [0,\infty]\},\end{aligned}$$

because different variables are relaxed in different disjuncts. Using the induced $V = \{\mathsf{x}, \mathsf{y}\}$, previous unassume operators return the less precise $\{\mathsf{x} \mapsto \top, \mathsf{y} \mapsto \top\}$.

Note that the implicit choice of V is different from the set of variables syntactically occurring in e in a more complex invariant language, such as our example language with pointers, which is described in Section 4.5.4. $\square$

Remark 10 (Explicit V). Implicitly choosing V causes this unassume operator to not fulfil specification **S3**: it will also relax $\mathsf{n} \mapsto [0, \infty]$.[34] However, the propagating unassume can be adapted to an explicit V by passing the set recursively down and changing the variable case to

$$\widetilde{\mathbb{E}}[\![x]\!]^\sharp\, v'\, d \stackrel{\text{def}}{=} \textbf{if } x \in V \textbf{ then } \{x \mapsto v'\} \textbf{ else } \varnothing.$$

This restricts relaxation to variables in the explicit set V and ignores others. Unlike the naïve unassume, this does not force the variables to be relaxed if they are not actually constrained by the invariant expression. $\square$

[34] With local iteration (described below) the final result will be even worse: $\{\mathsf{i} \mapsto \top, \mathsf{n} \mapsto \top\}$.

4.5.3. Local Iteration

Repeated application of the propagating algorithm for assuming can improve precision in the presence of dependent subexpressions, i.e., when the same variable occurs multiple times in the condition [Gra92; Cou99; Zia19; Min17]. Analogously, repeated application of the propagating algorithm for unassuming can perform more relaxation in the presence of dependent subexpressions. Both repetitions can be iterated to a local fixpoint. This is demonstrated by the following examples. If the previous unassume operators build on a locally-iterating assume operator, then the propagating unassume should also perform local iteration to not fall short.

Example 17. Consider using the above algorithm to compute the following:

$$[\![\mathsf{unassume}(0 < \mathtt{i} < \mathtt{j} < 10)]\!]^\sharp \{\mathtt{i} \mapsto [3,3], \mathtt{j} \mapsto [7,7]\},$$

which is short for

$$[\![\mathsf{unassume}(0 < \mathtt{i} \wedge \mathtt{i} < \mathtt{j} \wedge \mathtt{j} < 10)]\!]^\sharp \{\mathtt{i} \mapsto [3,3], \mathtt{j} \mapsto [7,7]\}.$$

In the first iteration, each conjunct is handled as follows:

1. The first conjunct computes $[0,0] <^\sharp [3,3]$. Backward propagation then returns $\{\mathtt{i} \mapsto [1,\infty]\}$.
2. The second conjunct computes $[3,3] <^\sharp [7,7]$. Backward propagation then returns $\{\mathtt{i} \mapsto [-\infty,6], \mathtt{j} \mapsto [4,\infty]\}$.
3. The third conjunct computes $[7,7] <^\sharp [10,10]$. Backward propagation then returns $\{\mathtt{j} \mapsto [-\infty,9]\}$.

The combined result is their intersection: $\{\mathtt{i} \mapsto [1,6], \mathtt{j} \mapsto [4,9]\}$.

In the second iteration, each conjunct is handled as follows:

1. The first conjunct computes $[0,0] <^\sharp [1,6]$. Backward propagation then returns $\{\mathtt{i} \mapsto [1,\infty]\}$.
2. The second conjunct computes $[1,6] <^\sharp [4,9]$. Backward propagation then returns $\{\mathtt{i} \mapsto [-\infty,8], \mathtt{j} \mapsto [2,\infty]\}$.
3. The third conjunct computes $[4,9] <^\sharp [10,10]$. Backward propagation then returns $\{\mathtt{j} \mapsto [-\infty,9]\}$.

The combined result is their intersection: $\{\mathtt{i} \mapsto [1,8], \mathtt{j} \mapsto [2,9]\}$. This is also the final result because another iteration yields the same result, having reached a fixpoint. □

In case the value domain has infinite strictly descending chains, the local iteration of propagating assume must use narrowing to ensure termination [Gra92, Section 2.3; Cou99, Section 11.3; Min17, Section 4.6.3].[35] Similarly, the local iteration of propagating unassume must use widening instead.

[35] Curiously, none of the cited authors provide an example where narrowing is necessary. For

Example 18. Consider using the above algorithm to compute the following:

$$[\![\mathsf{unassume}(\mathtt{i} \leq \mathtt{i} + 1)]\!]^\sharp \{\mathtt{i} \mapsto [0, 0]\}.$$

The algorithm makes the following iterations:

1. The first forward propagation computes $[0, 0] \leq^\sharp [1, 1]$. Backward propagation then returns $\{\mathtt{i} \mapsto [-\infty, 1] \sqcap [-1, \infty] = [-1, 1]\}$. If no widening is applied, then these bounds keep growing by one per iteration. If widening is applied, then we get $[0, 0] \nabla [-1, 1] = \top$.
2. The second forward propagation computes $\top \leq^\sharp \top$. Backward propagation cannot relax anything further, so the result is $\{\mathtt{i} \mapsto \top\}$. This is consistent with expectation: all values of $\mathtt{i}$, where the tautology holds. □

Interestingly, it can also be beneficial to continue with narrowing iterations after the widening iterations, as the following example demonstrates.

Example 19. In Example 17, we used local iteration *without* widening to compute the following:

$$[\![\mathsf{unassume}(0 < \mathtt{i} \wedge \mathtt{i} < \mathtt{j} \wedge \mathtt{j} < 10)]\!]^\sharp \{\mathtt{i} \mapsto [3, 3], \mathtt{j} \mapsto [7, 7]\}.$$

We now include widening into the same computation. From Example 17, we know that the combined result of the first iteration is $\{\mathtt{i} \mapsto [1, 6], \mathtt{j} \mapsto [4, 9]\}$. After widening, it becomes: $\{\mathtt{i} \mapsto \top, \mathtt{j} \mapsto \top\}$, which is less precise than the result of Example 17.

Thus, we continue with narrowing iterations. Now in the second iteration, each conjunct is handled as follows:

1. The first conjunct computes $[0, 0] <^\sharp \top$. Backward propagation then returns $\{\mathtt{i} \mapsto [1, \infty]\}$.
2. The second conjunct computes $\top <^\sharp \top$. Backward propagation then returns $\{\mathtt{i} \mapsto \top, \mathtt{j} \mapsto \top\}$.
3. The third conjunct computes $\top <^\sharp [10, 10]$. Backward propagation then returns $\{\mathtt{j} \mapsto [-\infty, 9]\}$.

The combined result is their intersection: $\{\mathtt{i} \mapsto [1, \infty], \mathtt{j} \mapsto [-\infty, 9]\}$. After narrowing, the widened result is improved: $\{\mathtt{i} \mapsto [1, \infty], \mathtt{j} \mapsto [-\infty, 9]\}$.

---

completeness, consider the following (contrived) example:

$$[\![\mathsf{assume}(\mathtt{i} + 1 \leq \mathtt{i})]\!]^\sharp \{\mathtt{i} \mapsto [0, 100]\}.$$

Without narrowing, the local iteration terminates after ~50 iterations (depending on exact counting) with $\{\mathtt{i} \mapsto \bot\}$. With standard interval narrowing, the local iteration terminates after one iteration with $\{\mathtt{i} \mapsto [0, 100]\}$, because there are no infinite bounds to improve. In fact, the same happens in all cases when trying to refine bounded intervals. Thus, a *narrowing delay* [Cou15, Section 6.2] of at least 1 is probably warranted.

The necessity of narrowings has been studied by Amato et al. [Ama+18], but for program loops, not local iterations of assume operators.

In the third iteration, each conjunct is handled as follows:

1. The first conjunct computes $[0,0] <^\sharp [1,\infty]$. Backward propagation then returns $\{\mathtt{i} \mapsto [1,\infty]\}$.
2. The second conjunct computes $[1,\infty] <^\sharp [-\infty,9]$. Backward propagation then returns $\{\mathtt{i} \mapsto [-\infty,8], \mathtt{j} \mapsto [2,\infty]\}$.
3. The third conjunct computes $[-\infty,9] <^\sharp [10,10]$. Backward propagation then returns $\{\mathtt{j} \mapsto [-\infty,9]\}$.

The combined result is their intersection: $\{\mathtt{i} \mapsto [1,8], \mathtt{j} \mapsto [2,9]\}$, which is also the result of another narrowing. At this point, we have reached the same final fixpoint result as in Example 17. □

In this example, widening causes a complete loss of precision, but two narrowing iterations can recover desired precision: the outer inequalities bound the variables from the outside, while the inner inequality mutually refines the variables further.

Combining Examples 18 and 19, the following example requires both widening and narrowing to get the precise result with guaranteed termination.

Example 20. Consider using the above algorithm to compute the following:

$$[\![\mathsf{unassume}(0 \le \mathtt{i} \le \mathtt{i}+1 \le 100)]\!]^\sharp \{\mathtt{i} \mapsto [50,50]\},$$

which is short for

$$[\![\mathsf{unassume}(0 \le \mathtt{i} \wedge \mathtt{i} \le \mathtt{i}+1 \wedge \mathtt{i}+1 \le 100)]\!]^\sharp \{\mathtt{i} \mapsto [50,50]\}.$$

The algorithm makes the following iterations:

1. The first backward propagation returns $\{\mathtt{i} \mapsto [49,51]\}$, which after widening becomes $\{\mathtt{i} \mapsto \top\}$.
2. The second backward propagation returns $\{\mathtt{i} \mapsto [0,99]\}$, which is also the result of narrowing. □

In this example, the inner inequality pushes the unassume to widen, while the outer inequalities can refine by narrowing some precision lost by widening.

For reference, Table 10 presents a summary of which propagating unassume operators support which specifications given by examples in this section. Local iteration will also be necessary for specification **S10**, tackled in the following section.

Remark 11 (Local iteration on partial maps). Let $d \in \mathbb{D}$ be the initial abstract state and $f : \mathbb{D} \to \mathbb{D}$ an abstract state transformer. Then $\lim_d f$ denotes the local iteration of f, starting from d, to a fixpoint, using widening and narrowing. Thus, the obvious way to iterate the propagating unassume to a fixpoint is

$$[\![\mathsf{unassume}(e)]\!]^\sharp\, d \stackrel{\text{def}}{=} \lim_d (\lambda d.\, d \mathbin{\widetilde{\sqcap}} \widetilde{\mathbb{C}}[\![e]\!]^\sharp\, d),$$

where the transformer is given by a lambda function corresponding to the uniterated algorithm.

Table 10. Supported specifications for propagating unassume operators: supported (✓), supported without termination guarantee (✓✗) or not supported (✗)

Specification	No local iteration	Local iteration		
		w/o widening	w/ widening	w/ narrowing
Example 17	✗	✓	✗[a]	✓[a]
Example 18	✗	✓✗	✓	✓
Example 20	✗	✓✗	✗	✓
S10	✗	✓	✓	✓

[a] See Example 19.

However, since $\widetilde{\mathbb{C}}[\![e]\!]^\sharp$ returns a partial map, an alternative way would be

$$[\![\mathsf{unassume}(e)]\!]^\sharp\, d \stackrel{\text{def}}{=} d \between \mathrm{lim}_\varnothing\, (\lambda d'.\, \widetilde{\mathbb{C}}[\![e]\!]^\sharp\, (d \between d')),$$

which only iterates the partial unassume map to a fixpoint, starting from the empty map ($\varnothing$). It is then joined with d to return a complete abstract state, but also to apply $\widetilde{\mathbb{C}}[\![e]\!]^\sharp$, which requires a complete state to evaluate expressions.

The two ways are reminiscent of naïve and semi-naïve evaluation of Datalog, respectively. In particular, the latter iterates on smaller maps which can be more efficient if only a few variables of a large state are unassumed. However, in the presence of widening and narrowing, it is unclear how (if at all) the two versions relate in terms of precision. □

4.5.4. Pointers

In the previous sections, we claim that the algorithms generalize from variables to lvalues. The key machinery for this is the variable set variance property of the propagating unassume. While the original HC4-revise algorithm directly refines variables at leaf nodes, we handle pointers by letting leaf nodes create partial states containing the dynamically resolved variables.

Syntax. Let the syntax of lvalues l be defined by the grammar[36,37]

$$\begin{aligned} l &::= x && \text{(variable, } x \in \mathcal{V}\text{)} \\ &\mid *l && \text{(dereferencing).} \end{aligned}$$

Let lvalues replace variables in the syntax of expressions. Let constants include address-taking as $\&x$ for a variable $x \in \mathcal{V}$.

[36] This is more general than the extended version of the original paper [Saa+23c, Appendix A] which defines $l ::= x \mid *x$.

[37] Lvalues in C are even more general as they allow dereferencing of expressions (e.g., involving pointer arithmetic). Our definition is not more restrictive, though, as $*e$, where e is a pointer-typed expression, can be achieved by first assigning the value of e to some fresh temporary variable and then dereferencing that.

Evaluation. Let the abstract value domain for pointer variables be points-to sets of variables $2^{\mathcal{V}}$. The forward evaluation of address-taking is defined by $(\&x)^\sharp \stackrel{\text{def}}{=} \{x\}$. In examples, we write elements of points-to sets as address-taking constants to make them more explicit, e.g., $\{\&\mathtt{i}, \&\mathtt{j}\}$ instead of $\{\mathtt{i}, \mathtt{j}\}$. The forward evaluation of lvalues is defined as

$$\mathbb{E}[\![l]\!]^\sharp\, d \stackrel{\text{def}}{=} \bigsqcup_{x \in \mathbb{L}[\![l]\!]^\sharp\, d} d\, x,$$

where the helper function $\mathbb{L}[\![l]\!]^\sharp : \mathbb{D} \to 2^{\mathcal{V}}$ for evaluating lvalues to points-to sets is given by

$$\mathbb{L}[\![x]\!]^\sharp\, d \stackrel{\text{def}}{=} \{x\}, \qquad\qquad \mathbb{L}[\![*l]\!]^\sharp\, d \stackrel{\text{def}}{=} \bigcup_{x \in \mathbb{L}[\![l]\!]^\sharp\, d} d\, x.$$

Remark 12. Alternatively, they can be defined mutually recursively, which reduces redundancy:

$$\begin{aligned} \mathbb{L}[\![x]\!]^\sharp\, d &\stackrel{\text{def}}{=} \{x\}, & \mathbb{E}[\![x]\!]^\sharp\, d &\stackrel{\text{def}}{=} d\, x, \\ \mathbb{L}[\![*l]\!]^\sharp\, d &\stackrel{\text{def}}{=} \mathbb{E}[\![l]\!]^\sharp\, d, & \mathbb{E}[\![*l]\!]^\sharp\, d &\stackrel{\text{def}}{=} \bigsqcup_{x \in \mathbb{L}[\![*l]\!]^\sharp\, d} \mathbb{E}[\![x]\!]^\sharp\, d. \end{aligned}$$

□

Assuming. The backward evaluation of lvalues is defined analogously:

$$\overleftarrow{\mathbb{E}}[\![l]\!]^\sharp\, v'\, d \stackrel{\text{def}}{=} \bigsqcup_{x \in \mathbb{L}[\![l]\!]^\sharp\, d} d[x \overset{\perp}{\mapsto} d\, x \sqcap v'].$$

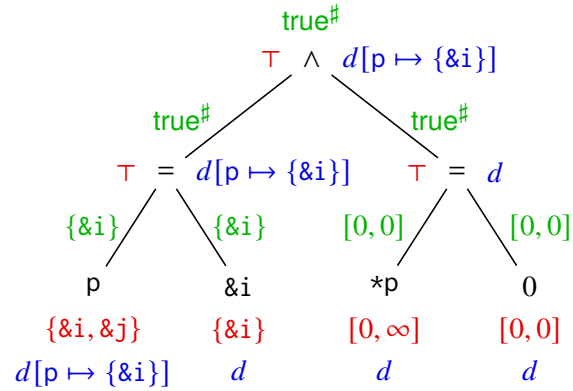
If the points-to set is ambiguous and none of updates reduce to $\perp$, then no refinement takes place. This is because each variable is refined in one of the joinees, but left untouched in the others, leaving the joined value also unchanged.

Example 21. Consider using the above algorithm for:

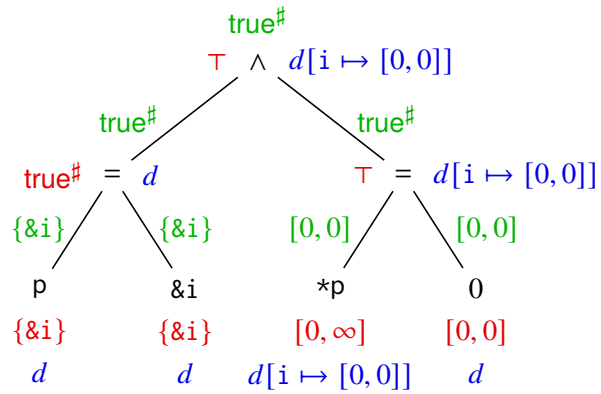
$$[\![\mathsf{assume}(\mathtt{p} = \&\mathtt{i} \wedge *\mathtt{p} = 0)]\!]^\sharp \{\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}, \mathtt{i} \mapsto [0, \infty], \mathtt{j} \mapsto [0, \infty]\}.$$

The first iteration is shown in Fig. 10a. As formalized above, backward propagation returns for $\mathtt{p} = \&\mathtt{i}$ the map where $\mathtt{p}$ has been refined to $\{\&\mathtt{i}\}$ and uses $*\mathtt{p} = 0$ to refine $*\mathtt{p}$. To do so, the backward operator $\overleftarrow{=}^\sharp$ uses the expected $\mathsf{true}^\sharp$ result and its forward-evaluated right argument $[0, 0]$ to propagate the expected value $[0, 0]$ into its left argument. Still using the initial forward evaluated $\mathtt{p} \mapsto \{\&\mathtt{i}, \&\mathtt{j}\}$, backward propagation of the lvalue $*\mathtt{p}$, acting as a leaf, returns the environment unchanged since the points-to set is ambiguous. The new constructed environment is $\{\mathtt{p} \mapsto \{\&\mathtt{i}\}, \mathtt{i} \mapsto [0, \infty], \mathtt{j} \mapsto [0, \infty]\}$.

The second iteration is shown in Fig. 10b. Backward propagation does all of the above but now refines $\mathtt{i} \mapsto [0, 0]$, due to the new unambiguous points-to set. The final fixpoint environment is $d = \{\mathtt{p} \mapsto \{\&\mathtt{i}\}, \mathtt{i} \mapsto [0, 0], \mathtt{j} \mapsto [0, \infty]\}$. □



(a) Iteration 1 with $d = \{\mathtt{p} \mapsto \{\mathtt{\&i}, \mathtt{\&j}\}, \mathtt{i} \mapsto [0, \infty], \mathtt{j} \mapsto [0, \infty]\}$



(b) Iteration 2 with $d = \{\mathtt{p} \mapsto \{\mathtt{\&i}\}, \mathtt{i} \mapsto [0, \infty], \mathtt{j} \mapsto [0, \infty]\}$

Figure 10. Phases of propagating algorithm for $[\![\mathsf{assume}(\mathtt{p} = \mathtt{\&i} \wedge *\mathtt{p} = 0)]\!]^\sharp\, d$: forward propagation (in red), backward propagation (in green) and result construction (in blue)

Remark 13 (Pointer refinement via value). With these definitions, the computation

$$[\![\mathsf{assume}(*\mathtt{p} = 2)]\!]^\sharp\, \{\mathtt{p} \mapsto \{\mathtt{\&i}, \mathtt{\&j}\}, \mathtt{i} \mapsto [1, 1], \mathtt{j} \mapsto [2, 2]\}$$

does not refine the state at all, however, the pointer p could be soundly refined to $\{\mathtt{\&j}\}$.

The following alternative definition refines both values and (nested) pointers to them:

$$\overleftarrow{\mathbb{E}}[\![x]\!]^\sharp\, v'\, d \stackrel{\text{def}}{=} d[x \mathrel{\dot{\mapsto}} d\, x \sqcap v'],$$
$$\overleftarrow{\mathbb{E}}[\![*l]\!]^\sharp\, v'\, d \stackrel{\text{def}}{=} \bigsqcup_{x \in \mathbb{L}[\![*l]\!]^\sharp\, d} \overleftarrow{\mathbb{E}}[\![l]\!]^\sharp\, \{x\}\, (\overleftarrow{\mathbb{E}}[\![x]\!]^\sharp\, v'\, d).$$

In the latter, refining x before l is more efficient: if it ends up being $\bot$ (the pointed value does not match), then the recursion to l (to refine the pointer) can be avoided. □

Unassuming. The unassuming of lvalues is also defined analogously:

$$\widetilde{\mathbb{E}}[\![l]\!]^\sharp\, v'\, d \stackrel{\text{def}}{=} \left\lfloor \star \right\rfloor_{x \in \mathbb{L}[\![l]\!]^\sharp d} \{x \mapsto v'\}.$$

However, it is easier understood using the following equivalent formulation:

$$\widetilde{\mathbb{E}}[\![l]\!]^\sharp\, v'\, d \stackrel{\text{def}}{=} \{x \mapsto v' \mid x \in \mathbb{L}[\![l]\!]^\sharp\, d\}.$$

This can be seen as a special case of abstract assignment:

$$\widetilde{\mathbb{E}}[\![l]\!]^\sharp\, v'\, d \stackrel{\text{def}}{=} [\![l \leftarrow \gamma\, v']\!]^\sharp\, \{x \mapsto \bot \mid x \in \mathbb{L}[\![l]\!]^\sharp\, d\},$$

where $[\![l \leftarrow e]\!]^\sharp\, d$ is the abstract operator for assigning the value of expression e to the lvalue l. In this special case, the assigned abstract value is v', not $\mathbb{E}[\![e]\!]^\sharp\, d$.

Therefore, the propagating unassume for lvalues can be implemented using existing primitives: backward operators and abstract assignment. This is not limited to just simple pointers but works the same way for C lvalues, which also include array indexing and struct field offsets.

We can now see how the propagating algorithm can handle what was out of reach for the exploding unassume (see Table 9).

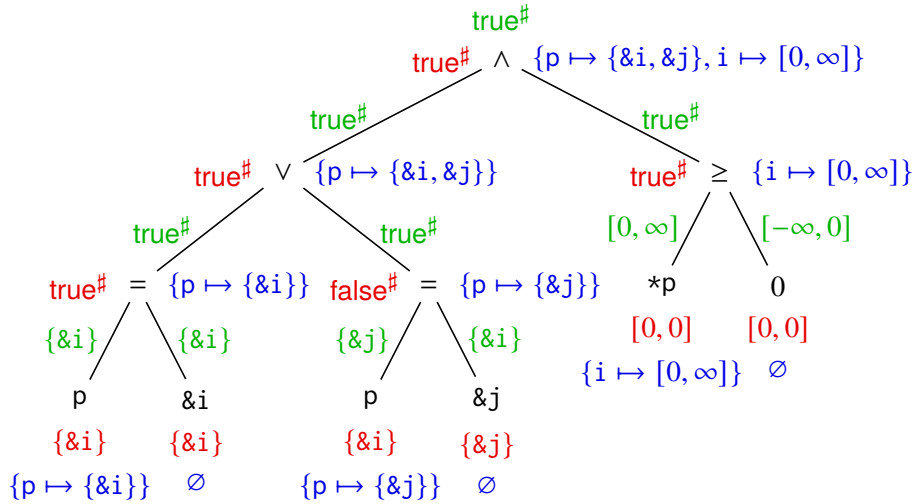
Example 22. Consider using the propagating unassume with local iteration for specification **S10**:

$$[\![\mathsf{unassume}((\mathtt{p} = \mathtt{\&i} \vee \mathtt{p} = \mathtt{\&j}) \wedge {*\mathtt{p}} \geq 0)]\!]^\sharp\, \{\mathtt{p} \mapsto \{\mathtt{\&i}\}, \mathtt{i} \mapsto [0,0], \mathtt{j} \mapsto [0,0]\}.$$

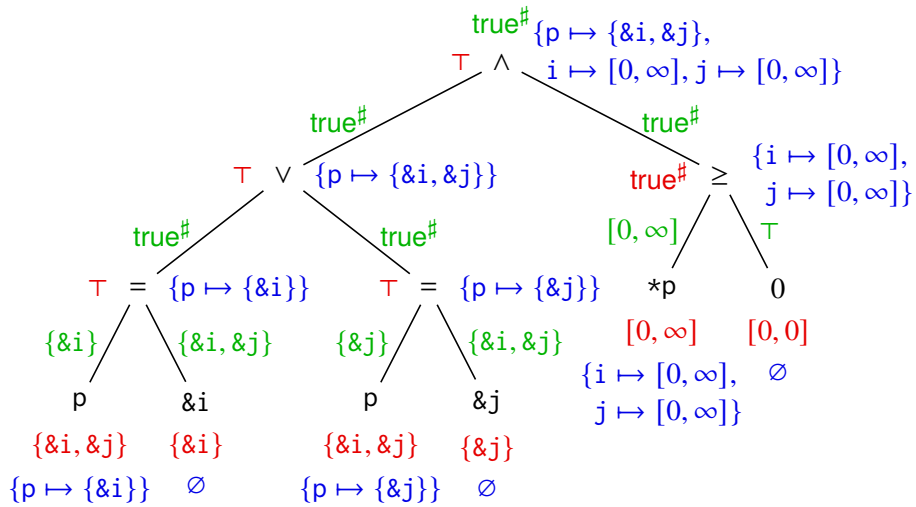
The first iteration is shown in Fig. 11a. As formalized above, backward propagation returns for $\mathtt{p} = \mathtt{\&i} \vee \mathtt{p} = \mathtt{\&j}$ the partial map $\{\mathtt{p} \mapsto \{\mathtt{\&i}, \mathtt{\&j}\}\}$ and uses ${*\mathtt{p}} \geq 0$ to relax ${*\mathtt{p}}$. To do so, the backward operator $\overleftarrow{\geq}^\sharp$ uses the expected $\mathsf{true}^\sharp$ result and its forward-evaluated right argument $[0,0]$ to propagate the expected value $[0,\infty]$ into its left argument. Using the forward evaluated $\mathtt{p} \mapsto \{\mathtt{\&i}\}$, backward propagation of the lvalue ${*\mathtt{p}}$, acting as a leaf, returns the partial map $\{\mathtt{i} \mapsto [0,\infty]\}$. The new constructed environment is $\{\mathtt{p} \mapsto \{\mathtt{\&i}, \mathtt{\&j}\}, \mathtt{i} \mapsto [0,\infty], \mathtt{j} \mapsto [0,0]\}$. If local iteration was not used, then the desired result for specification **S10** would not be achieved (as indicated in Table 10).

The second iteration is shown in Fig. 11b. Backward propagation does all of the above and also includes $\mathtt{j} \mapsto [0,\infty]$, due to the new points-to set. The final fixpoint environment is $\{\mathtt{p} \mapsto \{\mathtt{\&i}, \mathtt{\&j}\}, \mathtt{i} \mapsto [0,\infty], \mathtt{j} \mapsto [0,\infty]\}$. □

In this example, the two iterations induced $V = \{\mathtt{p}, \mathtt{i}, \mathtt{j}\}$, which used with the naïve unassume yields the issue described in Example 14 (p. 91). Therefore, the propagating unassume is not equivalent to simply using its induced variable set with the naïve unassume. The propagating unassume fuses the multiple steps involved together into one algorithm which avoids intermediate imprecision and undefined behavior.



(a) Iteration 1 with $d = \{\texttt{p} \mapsto \{\texttt{\&i}\}, \texttt{i} \mapsto [0,0], \texttt{j} \mapsto [0,0]\}$



(b) Iteration 2 with $d = \{\texttt{p} \mapsto \{\texttt{\&i}, \texttt{\&j}\}, \texttt{i} \mapsto [0,\infty], \texttt{j} \mapsto [0,0]\}$

Figure 11. Phases of propagating algorithm for $[\![\texttt{unassume}((\texttt{p} = \texttt{\&i} \vee \texttt{p} = \texttt{\&j}) \wedge *\texttt{p} \geq 0)]\!]^\sharp\, d$: forward propagation (in red), backward propagation (in green) and result construction (in blue)

4.6. Implementation

We have extended the GOBLINT abstract interpretation framework (Chapter 3) with:

1. YAML-witness–guided verification with unassume operators (Section 4.6.1),
2. YAML witness validation (Section 4.6.2),
3. YAML witness generation (Section 4.6.3).

To this end, our implementation includes parsing of YAML witnesses (Section 2.5.3) and matching the provided invariants to program locations such that they can be unassumed or validated.

4.6.1. Witness-Guided Verification

Sound unassume operators must preserve all reaching concrete states, thus, preserving soundness of the entire analysis. Our implementation contains the following operators for the value analyses described in Section 3.3.1:

Base analysis. This non-relational analysis implements the *propagating* unassume (Section 4.5.2) for its domains (e.g., numeric intervals or points-to sets). The existing propagating assume in GOBLINT could be generalized and directly reused, yielding an unassume operator capable of handling, e.g., C lvalues, not just variables, with no extra effort (as described in Section 4.5.4).

There is an option for the local iteration of propagating unassume, following Remark 11 (p. 101). It is disabled by default because it makes the unassume operator at least twice as expensive: in most cases a single iteration is sufficient, but an extra iteration is needed just for the fixpoint check.

APRON analysis. This numeric relational analysis implements two unassume operators to choose from for its domains (octagons and polyhedra):

- The *strengthening-based dual-narrowing* unassume (Section 4.4.3) is implemented using the generic approach described in Example 12, because APRON [JM09] does not provide dual-narrowing. The optimization from Remark 6 is also used.
- The *naïve* unassume (Section 4.4.2) is implemented as an alternative, because the strengthening operator can be very expensive, even in cases where it restores no additional precision.[38]

Since this analysis is just numeric, V is collected syntactically.

Variable equalities analysis. This symbolic relational analysis implements the *naïve* unassume with V also collected syntactically. Since this analysis does not refine on assume, the unassume operator degrades to $[\![\mathsf{unassume}_V(e)]\!]^\sharp_1\, d = [\![\mathsf{havoc}(V)]\!]^\sharp\, d$.

[38] Unfortunately, at the time of SV-COMP 2024 and 2025, the cheaper alternative was instead implemented as described in Remark 5 (p. 88).

Instead of altering the CFGs, unassume statements are inserted implicitly as events that activated analyses can handle. In the modular architecture of GOBLINT (Section 3.2), the unassume analysis is responsible for emitting these events after transfer functions corresponding to witness invariants.

Widening Tokens. To prevent unintended precision loss when widening from the initially reached abstract values to the unassumed ones, we must take care to delay the application of widening. We tag abstract values with unique identifiers of incorporated witness invariants (UUID of the `invariant_set` entry from the YAML witness together with the index of the invariant in the set) and delay the widening if this set increases [MSS13]. Such *widening tokens*[39] ensure that each witness invariant can be incorporated without immediate overshooting.

Example 23. Recall Example 9 (p. 78) with the program in Listing 3. Now, suppose that a witness provides the invariant $1 \leq \mathsf{x} \leq 40$ in the loop body on line 4 instead. When guiding the abstract interpreter with this witness, it first reaches the loop head on line 3 with the interval $[40, 40]$ for x. Continuing in the loop body, the unassume operator relaxes the lower bound of the interval using the provided invariant. After one iteration, the loop head is reached with $[0, 39]$, so the abstract value at that point is $[0, 40]$. To accelerate fixpoint iteration for termination, standard interval widening is applied, which abandons the unstable lower bound, resulting in $[-\infty, 40]$. Thus, the lower bound is irrecoverably lost, just like in Example 8. Table 11 lists the abstract evaluation steps of this analysis in order, similarly to Table 7. □

Table 11. Analysis progress and result (highlighted) for the program in Listing 3 unassuming invariant $1 \leq \mathsf{x} \leq 40$ on line 4

Line	x interval	Comment
2	$\top$	
3	$[40, 40]$	
4	$[1, 40]$	$[\![\mathsf{unassume}_{\{\mathsf{x}\}}(1 \leq \mathsf{x} \leq 40)]\!]^\sharp \{\mathsf{x} \mapsto [40, 40]\}$
5	$[0, 39]$	
3	$[-\infty, 40]$	$[40, 40] \nabla ([40, 40] \sqcup [0, 39])$
4	$[-\infty, 40]$	
5	$[-\infty, 39]$	
3	$[-\infty, 40]$	$[-\infty, 40] \nabla ([40, 40] \sqcup [-\infty, 39])$
3	$[-\infty, 40]$	$[-\infty, 40] \Delta ([40, 40] \sqcup [-\infty, 39])$
6	$[0, 0]$	

[39] Mihaila, Sepp, and Simon [MSS13] use such a delaying domain specifically with a set of program locations and do not assign a separate name to its elements. The term token originates from Bagnara et al. [Bag+05], although they only count tokens without distinguishing them.

Table 12. Analysis progress and result (highlighted) for the program in Listing 3 unassuming invariant $1 \leq \mathsf{x} \leq 40$ on line 4 with a widening token (tagged x)

Line	x interval	Comment
2	$\top$	
3	$[40, 40]$	
4	$[1, 40]_{\mathsf{x}}$	$[\![\mathsf{unassume}_{\{\mathsf{x}\}}(1 \leq \mathsf{x} \leq 40)]\!]^\sharp \{\mathsf{x} \mapsto [40, 40]\}$
5	$[0, 39]_{\mathsf{x}}$	
3	$[0, 40]_{\mathsf{x}}$	$[40, 40] \nabla ([40, 40] \sqcup [0, 39]_{\mathsf{x}})$
4	$[1, 40]_{\mathsf{x}}$	
5	$[0, 39]_{\mathsf{x}}$	
3	$[0, 40]_{\mathsf{x}}$	$[0, 40]_{\mathsf{x}} \nabla ([40, 40] \sqcup [0, 39]_{\mathsf{x}})$
3	$[0, 40]_{\mathsf{x}}$	$[0, 40]_{\mathsf{x}} \Delta ([40, 40] \sqcup [0, 39]_{\mathsf{x}})$
6	$[0, 0]_{\mathsf{x}}$	

Example 24. Now, suppose that the widening token x corresponding to the witness invariant is attached to the abstract value by the unassume operator. The loop head is again first reached with the singleton interval $[40, 40]$. After one loop iteration, the loop head is now reached with $[0, 39]_{\mathsf{x}}$, which makes the abstract value at that point $[0, 40]_{\mathsf{x}}$ by delaying the widening due to the addition of the token. Another iteration with this interval reaches the loop head with $[0, 39]_{\mathsf{x}}$, which is subsumed by the previous abstract value. Thus, a fixpoint is reached without the need for widening or narrowing, and the stronger invariant $[0, 40]$ is confirmed. Table 12 lists the analysis steps when using the unassume operator. Unlike Example 9, this witness-guided analysis took the same number of steps as the original. □

The use of widening tokens does not negatively impact Example 9 in terms of analysis steps. As the unassume operator has already been used when the loop head is reached, the initially reaching abstract value is already tagged. Therefore, the widening does not get delayed. For this reason, widening tokens related to unassumed invariants are preferable over standard counter-based widening delay in our use case.

4.6.2. Witness Validation

Our implementation of witness validation involves two related but independent components: (witness-guided) verification and witness-only validation. By Definition 11 (p. 37), the overall verdict depends on the individual verdicts of both components. Since the analyzer is sound, this yields a sound witness validator.

Witness-Only Validation. Correctness of witness invariants is determined by essentially treating them as additional proof obligations. However, they are not inserted into the program as assert statements before the usual verification. Instead, the witness invariants are checked by evaluating them on the abstract states computed

by the analysis, as described in Section 4.2. Hence, invalid witness invariants cannot undermine soundness of the verification process via refinement.

Independent Witnesses. The separation of concerns offers an interesting possibility: the witness used for witness-guided verification can be different from the one used for witness-only validation. To this end, GOBLINT provides separate configuration options for the two independent features. In the case of standard witness validation, the same YAML file is passed to both. However, the evaluation in Section 6.4 exploits the flexibility to use different witnesses.

4.6.3. Witness Generation

The implementation simply converts abstract domain states into invariant expressions $\mathcal{I}$ (e.g., side-effect–free C expressions) for output as witness invariants. We assume abstract domains provide the following functions:

- For a relational domain $\mathbb{R}$, let $\text{asExpr}_{\mathbb{R}} : \mathbb{R} \to \mathcal{I}$ do this by converting the constraints of a relational abstract value into an expression (often a conjunction).
- For a non-relational value domain $\mathbb{V}$, let $\text{asExpr}^{v}_{\mathbb{V}} : \mathbb{V} \to \mathcal{I}$ do this for a variable $v \in \mathcal{V}$. Unlike in the relational case, this additionally requires the variable itself because a non-relational abstract value (stored in a domain $\mathbb{D} = \mathcal{V} \to \mathbb{V}$) only describes the value. For example, $\text{asExpr}^{\mathsf{x}}_{\mathbb{V}}\,[0, 42] \stackrel{\text{def}}{=} 0 \leq \mathsf{x} \leq 42$.
- For a domain $\mathbb{D}$, let $\text{asExpr}_{\mathbb{D}} : \mathbb{D} \to \mathcal{I}$ do this in a suitable manner. If $\mathbb{D}$ is relational, then $\text{asExpr}_{\mathbb{D}} \stackrel{\text{def}}{=} \text{asExpr}_{\mathbb{R}}$. If $\mathbb{D}$ is non-relational, i.e., $\mathbb{D} = \mathcal{V} \to \mathbb{V}$, then
$$\text{asExpr}_{\mathbb{D}}\ d \stackrel{\text{def}}{=} \bigwedge_{v \in \mathcal{V}} \text{asExpr}^{v}_{\mathbb{V}}\,(d\,v).$$

For generating a witness for property Φ of a program P by abstract interpretation, we assume that the analyzer provides us with a mapping $\sigma : \mathcal{N} \to \mathbb{D}$ from locations to abstract values. This yields a witness (W, P, Φ), where $Wn \stackrel{\text{def}}{=} \text{asExpr}_{\mathbb{D}}\,(\sigma\,n)$.

Large witnesses that express full proofs with numerous invariants can be problematic for a validator [Bey+16; Alb+06], which must manipulate, use, and/or verify them. In our case, the validator performs an unassume operation for each invariant each time the corresponding transfer function is evaluated. The speedup gained from using the witness must outweigh the overhead to truly benefit from witnesses. This amounts to witnesses containing partial proofs, only containing key invariants. Hence, we apply simple heuristics for choosing at which program locations and about which program variables the witness generator emits invariants.

Loop invariants are key in the correctness proofs of single-threaded programs [Bey+16]. Thus, loops are the obvious locations for emitted invariants. More specifically, widening is also commonly applied at loop heads, whereas abstract interpretation of straight-line code does not involve fixpoint iteration, which

could be sped up or made more precise in this manner. Witness generation for multi-threaded programs is considered in Sections 6.3.1 and 6.7.1.

Unassumable Invariants. Our witness-guided verification approach does not take advantage of exact invariants, such as equalities outside of disjunctions, since these do not relax a reached state already representing such exact values. Even if such invariants are useful for some validators, they do not benefit our witness-guided verifier. Therefore, only inexact invariants, e.g., non-constant intervals, non-singleton points-to sets, etc., can be emitted in order to tailor a witness to such a validator. However, this tailoring might be detrimental to validation by other approaches.

4.7. Evaluation

A sound witness-guided verifier can trivially be obtained by replacing the unassume operator with the identity operator and ignoring the witnesses entirely. Thus, our experimental evaluation aims to demonstrate that our witness-guided verifier effectively uses witnesses. More specifically, we seek to confirm *content-effort dependence* (Section 1.2) by posing the following questions:

Precision. Can the witness-guided verifier leverage witnesses to validate verification results that it could not confirm without a witness?

Performance. Do witnesses influence the verification effort in the application domain of the analyzer?

It is worth noting that the performance improvement from technology-agnostic correctness witnesses is expected to be modest. In fact, Beyer et al. [Bey+16] observed no consistent trend in performance gains.

Experimental Setup. Our benchmarks are executed on a laptop running Ubuntu 22.04.3 on an AMD RYZEN 7 PRO 4750U processor. Each tool execution is limited to 900 s of CPU time on one core and 4 GB of RAM.

4.7.1. Precision

We collected and provide a set of 11 example programs (excluding duplicates) from literature [Min17; HH12; BH17; AS13] where more advanced abstract interpretation techniques are developed to infer certain invariants, while standard accelerated solving strategies fail. For example, this set includes the program in Listing 3. We configure GOBLINT the same as in SV-COMP 2023 (Section 3.4.3), except autotuning is disabled and relational analysis using polyhedra is unconditionally enabled. We manually created YAML witnesses containing suitable loop invariants for these programs. We also used two state-of-the-art verifiers from SV-COMP 2023 to generate real witnesses: CPACHECKER [BK11; DLW15] and ULTIMATE AUTOMIZER (UAUTOMIZER for short) [HHP13; Hei+23]. Both verifiers are able to verify these programs and produce GraphML witnesses. At that time,

Table 13. Witness validation verdicts for literature examples: program verified/witness validated (✓), program verified with witness-guidance but witness not validated (✓✗) or program not verified (✗)

Paper	Example	GOBLINT w/o witness	GOBLINT w/ witness from Manual	GOBLINT w/ witness from CPACHECKER	GOBLINT w/ witness from UAUTOMIZER
[Min17]	4.6	✗	✓	✓✗	✓
	4.7	✗	✓	✓✗	✓
	4.8	✓	✓	✓	✓
	4.10	✓	✓	✓✗	✓
[HH12]	1.b	✓	✓	✓✗	✓✗
	2.b	✗	✓	✓✗	✓
	3	✗	✓	✓✗	✓✗
[BH17]	1 (polyhedra)	✗	✓	✓✗	✓✗
	3	✗	✓	✗	✓
	"additional"	✗	✓	✗	✓
[AS13]	`hybrid`	✗	✓	✓✗	✓
Total	11	✓: 3	✓: 11	✓✗: 8, ✓: 1	✓✗: 3, ✓: 8

YAML witnesses had not yet been introduced to SV-COMP. Thus, following Beyer, Spiessl, and Umbricht [BSU22], we use CPACHECKER in its `witness2invariant` configuration to convert GraphML witnesses into YAML witnesses that GOBLINT can consume.

The results are summarized in Table 13. GOBLINT manages to verify the desired property for 3 of these programs without any witness, but can validate all handwritten witnesses, despite not implementing any of the advanced techniques needed for their inference. With CPACHECKER witnesses, our validator can verify 9 out of 11 programs and validate 1 out of 11 witnesses. With UAUTOMIZER witnesses, our validator can verify all 11 programs and validate 8 out of 11 witnesses. Furthermore, Fig. 12 shows that in many of these cases our abstract interpreter can validate (or at least reverify the program using) the witnesses from model checkers *orders of magnitude* faster than it took to generate them.

The evaluation, however, shows numerous instances where the program was only verified thanks to witness-guidance, but not all witness invariants could be validated, especially for CPACHECKER. This is precisely due to the phenomenon described in Example 10 (p. 80): on these small programs, explicit-state model checking is successful and yields disjunctive invariants over all the finitely-many cases. Surprisingly, an invariant is useful for witness-guided verification, even when it cannot be proven to hold abstractly.

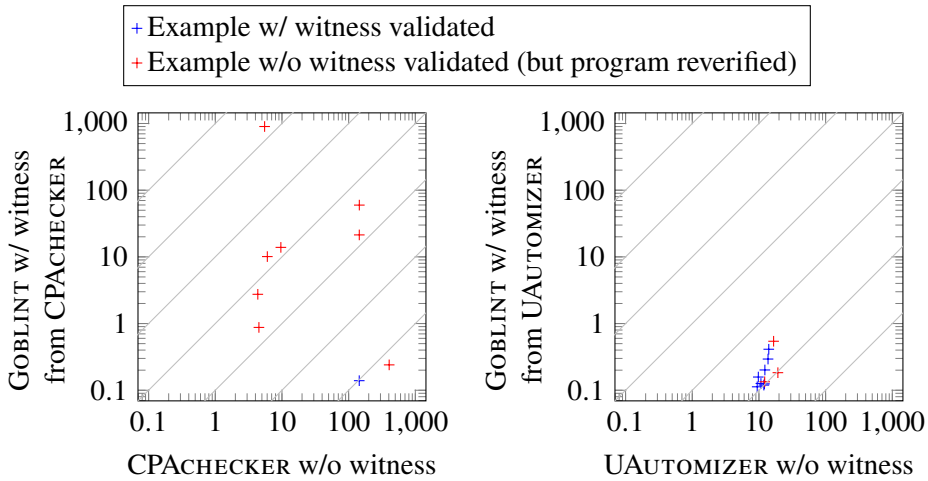


Figure 12. Cross-framework witness validation effort in CPU time (s) for literature examples; gray lines at powers of 10 from identity

Example 25. The invariant from Example 10 relaxes an abstract state, as set out by specification **S6**:

$$[\![\mathsf{unassume}_{\{x\}}(x = 1 \vee x = 2)]\!]^\sharp \{x \mapsto [1,1]\} = \{x \mapsto [1,2]\}.$$

Unlike specification **S6**, this relaxation happens to be exact. □

The disjunctive invariants from CPACHECKER tend to be large and, thus, expensive to unassume, as noted in Section 4.6.3. For this reason, they do not result in as consistent of a speedup as for UAUTOMIZER (in Fig. 12), which produces smaller invariants that are similar to the handwritten ones. In one instance [HH12, Example 3], witness-guided verification with GOBLINT ends up being two orders of magnitude slower than the original analysis and witness validation times out after 900 s.

4.7.2. Performance

To explore whether a suitable witness can reduce verification effort, we consider larger programs, as runtimes for the literature examples are negligible. Since GOBLINT specializes in the analysis of multi-threaded programs, we examine a set of multi-threaded POSIX programs from the GOBLINT benchmark suite.[40] Preliminary experiments show that witnesses generated by GOBLINT for these programs contain excessive information, which hinders witness-guided verification and validation. Therefore, we manually construct witnesses that contain core invariants for four of these programs (listed in Table 14), based on how widenings were applied during fixpoint solving (by studying the debug/trace output of

[40] `https://github.com/goblint/bench`

Table 14. Witness-guided verification effort for select GOBLINT benchmarks

		w/o witness		w/ witness (change)			
Program	LLoC	Evals	CPU time (s)	Evals		CPU time (s)	
`pfscan`	559	4,194	0.86	2,919	(−30%)	0.73	(−15%)
`aget`	587	7,932	2.23	4,683	(−41%)	1.68	(−25%)
`knot`	981	29,588	4.92	21,432	(−28%)	4.54	(−8%)
`smtprc`	3,037	48,559	15.00	24,091	(−50%)	7.95	(−47%)
Average					(−37%)		(−24%)

the solver). Moreover, this allows witness-guided verification to be evaluated independently of witness generation.

We configure GOBLINT as described earlier, but with relational analysis disabled. In addition to CPU time, we measure analysis effort without a witness and with witness-guidance via transfer function evaluation counts (evals). This metric of evaluations is proportional to CPU time, but excludes irrelevant pre- and post-processing, and is independent of hardware.

The results, aggregated in Table 14, show a noticeable performance improvement in the abstract interpreter when guided by a witness. For rough comparison, Albert, Puebla, and Hermenegildo [APH05] used analyzer-specific certificates (see Section 4.8) to achieve a weighted[41] average 1.61× time speedup in a different context. According to this metric, our approach achieves 1.54× CPU time improvement. However, the fixpoint-solving process still requires numerous widening iterations. This is due to the various abstractions used by GOBLINT that cannot be expressed as C expressions, including but not limited to array index ranges in abstract addresses, and various concurrency aspects. This demonstrates that measurable performance improvement is achievable by witness-guidance, but it remains future work to automatically generate such witnesses.

4.7.3. SV-COMP 2024

SV-COMP 2024 constitutes the first large-scale evaluation of our approach and its implementation [Saa+24a]. GOBLINT VALIDATOR is an extension of the GOBLINT verifier for validation of correctness witnesses in the YAML format. The extension involves two related but independent components: witness invariants are unassumed for speedup and checked for correctness.

Setup. The tool-info module for BENCHEXEC is named `goblint` (and shared with the GOBLINT verifier), while the benchmark definition for SV-COMP is named

[41] For each program, its speedup is weighted by its analysis time with a certificate/witness. This weighted average is equal to the total time w/o witness divided by the total time w/ witness.

`goblint-validate-correctness-witnesses-2.0`. They correspond to running the tool as follows:

```
./goblint --conf conf/svcomp25-validate.json \
          --set witness.yaml.unassume witness.yml \
          --set witness.yaml.validate witness.yml \
          --set ana.specification property.prp \
          --set exp.architecture 64bit \
          input.c
```

This is similar to the verifier (see Section 3.4), with the following differences:

- Witness-guided verification is enabled by activating the unassume analysis. For efficiency reasons, the Apron analysis is configured to use the naïve unassume.
- Widening tokens are enabled.
- The witness is provided by command-line argument: once for witness-guided verification and once for validation, since the two features are independent.
- The overall verdict now depends on both the (witness-guided) verification verdict and the witness-only validation verdict.
- Witness generation is disabled.

Goblint Validator version `svcomp24-0-gc2e9465a7` [Saa+23a] took part in *all* categories except *FalsificationOverall* of SV-COMP 2024 [Bey24]. In the validation track for YAML correctness witnesses, our validator ranked 3rd in *MemSafety*, 3rd in *NoOverflows*, 2nd in *SoftwareSystems* and 3rd in *Overall* out of 4 participants (the others being CPAchecker, Mopsa and UAutomizer).

Overall, Goblint Validator inherits the strengths and weaknesses of Goblint, which are described in Section 3.4. Thanks to the generic validation approach, the validator works in *all* the same SV-COMP categories as the Goblint verifier, including those that are currently excluded from correctness witness validation, e.g., *ConcurrencySafety*. Due to over-approximation, the verifier can only prove the absence of bugs, but not their presence. Consequently, the validator can currently only *confirm* correctness witnesses, but not reject them.

We evaluate our validator according to the same three aspects considered by Beyer et al. [Bey+16]: same-framework consistency, content-effort dependence and cross-framework validation (Section 1.2). The first two only focus on witnesses produced by the Goblint verifier.

Same-Framework Consistency. Table 15 lists how many tasks with each property it can verify and how many of those witnesses Goblint Validator can confirm. The overall confirmation rate of 78% is lower than the 90% Beyer et al. [Bey+16] report for CPAchecker and UAutomizer with GraphML witnesses.

Table 15. Witness validation consistency in the number of tasks for GOBLINT in SV-COMP 2024

Property	Correct tasks	GOBLINT verified	GOBLINT VALIDATOR Confirmed		GOBLINT VALIDATOR Unconfirmed
`unreach-call`	11,351	1,894	1,064	(56%)	830
`no-overflow`	5,562	3,932	3,416	(87%)	516
`termination`	1,536	619	297	(48%)	322
`no-data-race`	781	695	510	(73%)	185
`valid-memsafety`	2,796	1,963	1,801	(92%)	162
`valid-memcleanup`	2	0	—		—
Total	22,028	9,103	7,088	(78%)	2,015

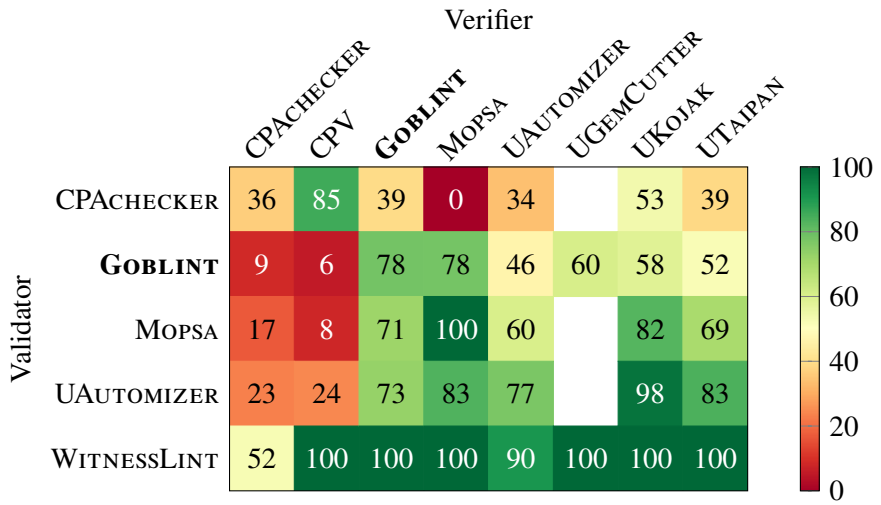


Figure 13. YAML correctness witness confirmation rates (%) for all verifier-validator pairs in SV-COMP 2024

Figure 13 presents the percentage of YAML correctness witnesses from each verifier confirmed by each validator in SV-COMP 2024. In this fairer comparison, GOBLINT VALIDATOR actually beats CPACHECKER (36%) and UAUTOMIZER (77%). MOPSA [Mon+24] achieves 99.9% self-validation by only producing trivial witnesses containing no invariants.

Note that SV-COMP 2024 was the first edition of the competition to use YAML witnesses. Thus, the implementations of YAML witnesses in the tools were very new compared to GraphML witnesses, which had been battle-tested for years. The relatively immature implementations may have led to subpar results for some tools. In particular, Ayaziová et al. [Aya+24] significantly improved CPACHECKER briefly after the competition and reached 70% self-validation on the `unreach-call` property with newly-generated correctness witnesses.

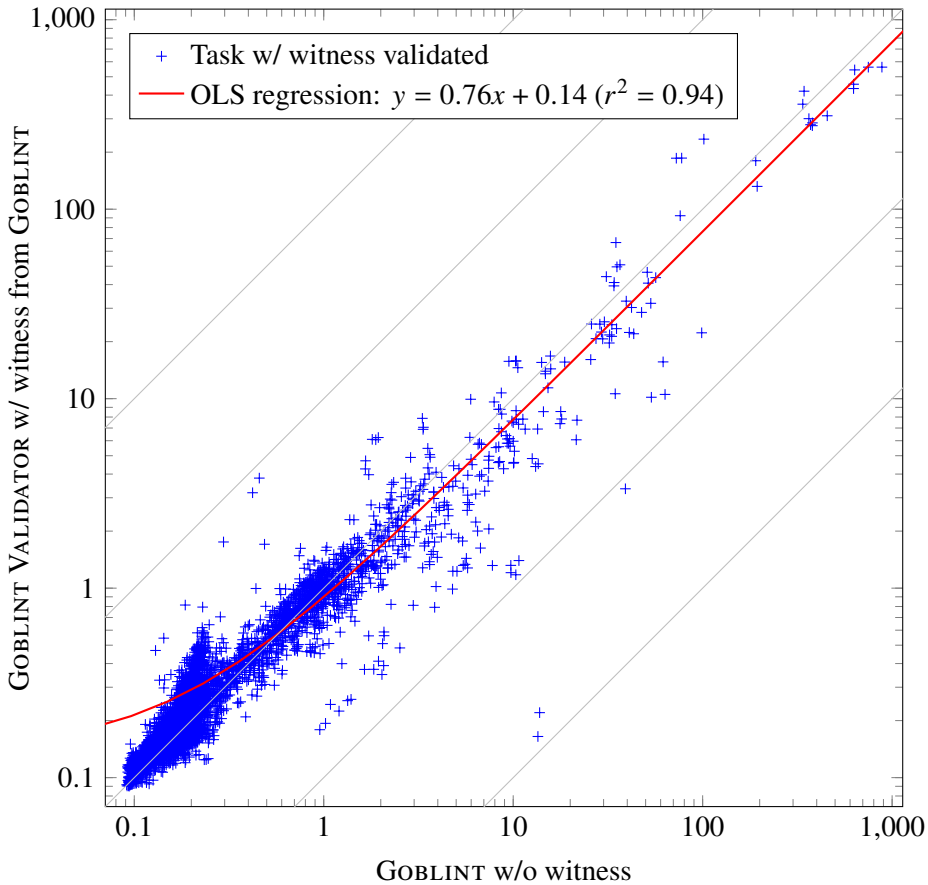


Figure 14. Witness validation effort in CPU time (s) for GOBLINT in SV-COMP 2024; gray lines at powers of 10 from identity

Reasons for self-unconfirmed GOBLINT witnesses range from excessive precision loss by unassuming to validator crashes. In some cases, the validator exceeds resource limits, likely due to large witnesses with many unhelpful invariants. A handful of instances indicate mismatches between witness generation and their interpretation due to implementation errors in either the verifier or the validator. Fixing such issues could improve the overall quality of the framework [Bey+16, Footnote 8, p. 333].

Content-Effort Dependence (Performance). Figure 14 plots the corresponding verification and validation times in the 7,088 confirmed cases. While the results at the low end (< 1 s) are noisy, the results at the high end (> 5 s) show the benefit of witness validation, with up to 10× improvements. Regression analysis estimates an average speedup of 24%, which matches our results from the previous section, albeit with greater variance. This is unlike CPACHECKER and UAUTOMIZER for which no overall performance improvement from consuming witnesses was observed [Bey+16].

Cross-Framework Validation. The necessary data for cross-framework comparison is also presented in Fig. 13, where we focus on the GOBLINT VALIDATOR row. For the ULTIMATE tool family, the confirmation rate is between 46% and 60%, which is similar to what Beyer et al. [Bey+16] observed. For CPACHECKER and CPV, the rate is below 10%, worst among all validators, probably due to large disjunctive invariants, as discussed in Section 4.7.1. We have a high rate for the MOPSA abstract interpreter which only produces trivial witnesses, on which GOBLINT VALIDATOR effectively reduces to the GOBLINT verifier. Nevertheless, the overwhelming success of MOPSA in the *SoftwareSystems* category warrants independent validation of its abstract interpretation results.

Note that UAUTOMIZER confirmed 23% of YAML correctness witnesses from CPACHECKER in SV-COMP 2024. Briefly after the competition, Ayaziová et al. [Aya+24] improved the witness generation of CPACHECKER and reached 44% confirmation by UAUTOMIZER on the `unreach-call` property. Hence, our low confirmation rate for CPACHECKER may have also been (at least partially) due to the same witness generation issues.

Content-Effort Dependence (Precision). Section 4.7.1 provides a number of programs that GOBLINT cannot verify without being guided by a witness. However, the results of SV-COMP 2024 reveal no such tasks where GOBLINT returns the verdict `unknown` but GOBLINT VALIDATOR returns `true` when validating a YAML correctness witness for the task (generated by *any* tool). Thus, such occurrences, although possible in theory, are extremely rare in practice. This is supported by various factors like:

1. GOBLINT VALIDATOR performs witness validation, not just witness-guided verification. Therefore, even if witness-guidance may have actually helped, then it might simply have failed to verify the unassumed invariants themselves. Witness validation verdicts in SV-COMP do not distinguish the two different sources of non-verification.
2. On larger programs, it may be that witness-guidance improves precision in some but not all parts of the program. The coarse verdicts in SV-COMP do not capture this.
3. Unassuming invariants cannot improve the expressivity of the abstractions, which likely is the limiting factor for precision of GOBLINT in most cases. The precision difference from unassuming tends to be subtle, and might be unrelated or too subtle as to the precision needed for verifying the property.

Thus, more fine-grained evaluations are needed to obtain a definite understanding.

4.7.4. Threats to Validity

Construct Validity. We have evaluated our approach according to the three pillars from Beyer et al. [Bey+16]. First, we just studied the feasibility of content-effort dependence in the precision and performance evaluations (Sections 4.7.1 and 4.7.2). Then, we went beyond feasibility and broadened the study to all the pillars in the SV-COMP evaluation (Section 4.7.3).

Internal Validity. Both our evaluations and SV-COMP were performed using the BENCHEXEC framework [BLW17] to ensure reliable benchmarking (as described in Section 3.4.6). Performance measurements using CPU time depend on the hardware and are subject to noise, which cannot be eliminated by BENCHEXEC. This was mitigated by also considering the GOBLINT-internal evals metric, where available. This metric is independent of hardware and deterministic (as long as GOBLINT is). Although CPU time and evals are correlated, the latter does not account for different transfer functions having different cost.

The precision evaluation used CPACHECKER as a black box to convert GraphML witnesses into YAML witnesses. Possible errors in this conversion may have worsened the results with the non-handwritten YAML witnesses. Moreover, the conversion may be better suited for GraphML witnesses produced by CPACHECKER than for those produced by UAUTOMIZER. In spite of these possible worsenings, the results are positive and Table 13 does not show bias against UAUTOMIZER.

Note that in SV-COMP 2024, GOBLINT also produced GraphML witnesses (for other validators) during the same verification run. As such, the SV-COMP performance evaluation does not account for the possible performance impact of this extra work that is not done by GOBLINT VALIDATOR.

External Validity. The benefits of witness-guided verification are specific to an abstract interpreter and a program. Abstract interpreters with different abstract domains and widening strategies may gain less from unassuming. The improvements are also limited by what abstractions can be expressed in the invariant language. Additionally, programs with (relatively) fewer or smaller loops offer fewer opportunities for effective unassuming.

The precision evaluation only considered 11 programs from abstract interpretation literature, and the performance evaluation considered just four programs from our own benchmark suite. Hence, improvements from witness-guided verification may not be possible in general. These are complemented by the SV-COMP evaluation which uses a much larger and more diverse benchmark suite (as described in Section 3.4.6). For example, the latter evaluation found no evidence of precision improvements due to unassuming.

The handwritten witnesses for the performance evaluation are constructed specifically to benefit GOBLINT. Thus, these witnesses may not be beneficial to other tools or even other abstract interpreters. The handcrafted witnesses for the precision evaluation, however, are constructed independently of GOBLINT and have later been submitted to the validation track in SV-COMP 2026.[42]

Albert, Puebla, and Hermenegildo [APH05] evaluate their approach using a different tool and different benchmarks in a different application domain. Nevertheless, both tools are tuned to their own benchmarks. Comparing high-level performance with them offers some context from the abstract interpretation literature, but does not imply anything about the conceptual approaches.

[42] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1628`

4.8. Related Work

Fixpoint iterations involving widening and narrowing are well studied [CC77; AS13; Ama+16; SV21; Cou15], but focus mostly on improving precision and ensuring termination. Halbwachs and Henry [HH12] extend fixpoint iteration with partial restarting, which derives from the narrowing result a new initial value for the following widening iteration, hoping it improves the result. Their restarted value is analogous to our partial initialization and could be used as such. They focus on finding such values automatically, while we focus on using them to avoid all the computation leading up to it. Boutonnet and Halbwachs [BH17] improve the technique for finding good restarting candidates. Cousot [Cou15] extends fixpoint iteration with dual-narrowing, hoping it improves the result further. Both approaches focus purely on improving precision with more iterations, while we aim to skip that iteration and arrive at the same result quicker, knowing the invariant. Hence, the techniques can be combined: use theirs to find a precise invariant and use ours to directly reuse it.

Arceri, Mastroeni, and Zaffanella [AMZ22] swap the abstract domain for a more precise one when switching from widening to narrowing. This can be considered a precision improvement technique, which makes the narrowing phase more expensive. However, it can also be viewed as an optimization, which makes the widening phase cheaper. Either way, it can be combined with our approach: immediately using the more precise domain with the final invariant, ideally skipping iteration in both ways.

Widening operators themselves are also well studied [Bla+03; Bag+05; GR06; MSS13; Min17]. Widening *up to* or *with threshold* use candidate invariants as intermediates to avoid irrecoverably losing precision. Such automated techniques can identify which candidates are true invariants. Using these as input to our approach is an effective way of supplying known good thresholds at specific program locations, removing the need for retrying all the candidates on reverification. This is one instance of dealing with the inherent non-monotonicity of widening operators [Cou15]. Furthermore, widening thresholds require domain-specific implementation, whereas our approach is more generic.

Our naïve unassume with its havoc and assume bears some similarity to the generation of *verification conditions* from user-supplied loop invariants [FS01]. However, there one additionally refines the state with the loop condition, and then checks that the loop invariant is preserved. We avoid the former to remain sound by construction for witness-guided verification, but effectively do the latter when validating witness invariants.

The classic approach to *(abstract) fixpoint checking* [Cou21, Chapter 35] can also be used for witness validation by abstract interpretation. In this approach, witness invariants are assumed to hold during fixpoint iteration and checked during

an additional iteration, which no longer assumes them. Assuming the invariants during fixpoint iteration can limit precision loss from widening and make the analysis more precise. However, it does not speed up the iteration process like our unassume approach. The classic approach is sound for witness validation but not for witness-guided verification, which omits the final crucial check. Nevertheless, our approach could possibly be combined with the classic one, which can only improve precision.

Albert, Puebla, and Hermenegildo [APH05] introduce *abstraction-carrying code* (ACC) as an abstract-interpretation–based instance of proof-carrying code (PCC). For validation, they use a simplified analyzer which only performs a single pass of abstract interpretation and no fixpoint iteration. Thus, it requires certificates to supply invariants for all loops. This is more restrictive than our validation approach, which runs in a single pass if all necessary invariants are provided, but also allows some fixpoint iteration if this is not the case. Nevertheless, we could handicap our analyzer with this stronger restriction.

Albert et al. [Alb+06] develop a notion of reduced certificates which can be smaller and which are used by the validator to reconstruct full certificates. Besson, Jensen, and Pichardie [BJP06] propose a fixpoint compression algorithm to further compact the certificates. In follow-up work, Besson, Jensen, and Turpin [BJT07] develop a theory for studying the issue of certificate size. Rather than using the strongest information from the least fixpoint of an analysis, they seek the weakest information still sufficient for implying correctness. This omits irrelevant information, leading to smaller witnesses. Such techniques could also be used when generating witnesses for our validator.

Correctness witness validators in SV-COMP fall into three broad categories:

1. Verifiers which treat witness invariants as additional proof obligations (validation via verification). As such, witness validation is just burden on top of the usual verification and, thus, (beneficial) content-effort dependence is not to be expected. UAutomizer [Bey+16] and Mopsa [Mon+24] perform the instrumentation internally, while MetaVal [BS20] performs program transformation and can delegate to any verifier.
2. Verifiers which can additionally benefit from witness invariants. They alter the normal verification using witness invariants and, thus, should exhibit content-effort dependence. CPAchecker [Bey+16] replaces its internal invariant generator, whereas Goblint uses our unassume operator.
3. Validators which cannot act as standalone verifiers. That is because they intentionally use a simpler approach which requires sufficient invariants as input and, thus, content-effort dependence cannot be evaluated. Such tools are LIV [BS23b] and UReferee.

4.9. Conclusion and Future Work

We have demonstrated how to turn abstract-interpretation–based tools into witness-guided verifiers and witness validators by equipping them with unassume operations. These can be constructed from abstract transformers for assumptions, non-deterministic assignments, joins and dual-narrowings, which allow retaining more precision for relational abstract interpretation. A powerful syntax-directed unassume operation for non-relational domains can be derived from a classic algorithm with minimal changes. Our implementation and evaluation demonstrate that unassuming invariants from witnesses can both speed up the analysis and make it more precise. The experiments further show that the abstract interpreter can benefit from witnesses produced by model checkers, and, thus, the results indicate that the approach is suited even for cross-technology collaboration.

Future Work. Nevertheless, witness validation using abstract interpretation can be improved and extended in numerous ways.

*Specification **S11**.* As seen in Table 9, none of the unassume operators presented here support specification **S11**. Abstract interpretation classically focuses on over-approximation, however, some kind of *under-approximation* may be preferable to fulfil this specification: if the abstraction is not precise, then preferably unassume less, rather than too much. Under-approximating abstract interpretation has been explored by Miné [Min14a], and Ascari, Bruni, and Gori [ABG22].

Witness Generation. As mentioned in Section 4.7.2, the witness generation described in Section 4.6.3 still yields too large witnesses which hinder validation. Therefore, the challenge remains for us to generate more compact but still non-trivial witnesses. For example, instead of outputting loop invariants over all program variables, only include those variables which are modified by the loop or were involved in widening.

Witness Validation. If the goal is witness validation, then extra care could be taken to avoid losing precision by unassuming invariants which can then no longer be proven by abstract evaluation. As a heuristic, if the invariant e does not abstractly evaluate to $\mathsf{true}^\sharp$ after its unassume operation, then the pre-unassume state could be kept (unchanged).

Violation Witness Rejection. GOBLINT VALIDATOR could be extended to *reject* violation witnesses. In the simplest form, this means rejecting violation witnesses for programs that it can prove correct, without considering the witness contents at all. Doing so scores points in the validation track. However, preliminary experiments indicate that there are very few violation witnesses for correct programs,[43] meaning this has limited applicability and usefulness. Note that in SV-COMP 2024, only three tools (CPACHECKER, CPV, SYMBIOTIC) produced YAML violation witnesses at all. Thus, the usefulness might increase as more tools adopt YAML violation witnesses instead of GraphML ones.

[43] https://github.com/goblint/analyzer/pull/1512

Violation Witness Refinement. Either way, a more promising extension would be to refine the abstract interpretation using the violation sequence provided by the witness, in the spirit of Saan [Saa20]. If GOBLINT VALIDATOR can prove a program correct after excluding the paths avoided by the violation witness and using its assumptions, then the witness can also be rejected because the remaining state space cannot lead to a violation. This could reject violation witnesses also for incorrect programs, but only those that claim a violation in some safe portion of the program.

Cooperative Witness Validation. For correctness witnesses which can neither be proven valid nor invalid (by providing a counterexample either to the program property or one of the invariants), a detailed output of which invariants the tool can validate might be interesting. The percentage of validated witness invariants lends itself to a notion of fractional witness validation. On the other hand, it also makes it possible to have cooperative witness validation, where no tool can validate all witness invariants but collectively all of them are covered. This would allow verifiers to output a variety of data into a witness without requiring all validators to validate all of it.

Proof-Like Counterexamples. In the context of model checking, Gurfinkel and Chechik [GC03] have noted that counterexamples are just proofs by example (of a violation). This proof method may be the simplest, but it is not the only one, nor necessarily the best one in every case. Therefore, they enrich counterexamples with additional proof steps. In the context of software verification, this would correspond to correctness-witness–like violation witnesses. It is to be determined whether a set of invariants could also come in handy for validating a violation. This may be another use case for the `unknown` witnesses of GOBLINT (Remark 2, p. 55).

SV-COMP Evaluation. At the end of Section 4.7.3, we find that no instances of precision improvement from witnesses were observed in SV-COMP 2024 verdicts. However, we also note that internally there may be precision improvements, but due to various factors, these are not reflected in the verdicts. For a definite (negative) conclusion, more fine-grained evaluation is necessary to isolate the effects of witness validation from witness-guided verification.

5. THREAD-MODULAR ABSTRACT INTERPRETATION

This chapter puts witnesses on hold and turns to the second branch of the thesis: *thread-modular abstract interpretation*. Specifically, it addresses goal **G2** to investigate the precision of existing approaches from literature.

Thread-modular abstract interpretation has been independently studied by Miné [Min12] and Vojdani [Voj10]. By formulating their non-relational value analyses in the common framework of side-effecting constraint systems, we find that neither subsumes the other in terms of precision. We propose novel refined and generalized analyses based on each other in order to mend the incomparability. Finally, we construct a relational thread-modular analysis based on the concept of common protecting mutexes from Vojdani.

5.1. Introduction

In a series of papers starting around 2012, Antoine Miné and his co-authors developed methods for abstract interpretation of concurrent systems [Min12; Min14b; SM16; MM17; SM18], which can be considered the *gold standard* for thread-modular static analysis of these systems. The core analysis from Miné [Min12] consists of a refinement of data flow which takes schedulability into account by propagating values written before unlock operations to corresponding lock operations — provided that appropriate side conditions are met. Due to these side conditions, more flows are generally excluded than in some other approaches [DDN11; Muk+17].

An alternative approach, suggested by Vojdani [Voj10; Voj+16], is realized in the static analyzer GOBLINT. This analysis is not based on data flows. Instead, for each global g, a set of mutexes that definitely protect accesses to g is determined. Then, *side effects* during the analysis of the threads' local states are used to accumulate an abstraction of all possibly written values. This base approach then is enhanced by means of *privatization* to account for exclusive manipulations by individual threads. This approach is similar to the thread-local shape analysis of Gotsman et al. [Got+07], which infers resource invariants [OHe07] by privatizing carved-out sections of the heap owned by a thread. Despite its conceptual simplicity and perhaps to our surprise, it turns out Vojdani's analysis is *not* subsumed by Miné's approach but is incomparable.

The goal of this chapter is to better understand this intriguing incomparability and develop precision improvements to refine these analyses. We mostly concentrate on the basic setting of *non-relational* analysis and a concurrent setting without precise thread IDs. However, Section 5.5 also discusses relational analysis and the use of thread IDs. We leave add-ons such as thread priorities or effects of

weak memory for future work (Section 5.9). As a common framework for the comparison, we use *side-effecting* constraint systems [ASV12].

To prove the given thread-modular analyses correct, we rely on a trace semantics of the concurrent system. We do not rely on a *global* trace semantics, but on a *local* trace semantics which formalizes for each thread that part of the computational past it can observe. The set of all local traces can conveniently be described by the least solution of a side-effecting constraint system which is of a form quite similar to the ones used by the analyses and, thus, well suited for proving their correctness.

Having formulated both the analyses of Miné [Min12] and Vojdani [Voj10; Voj+16] by means of side-effecting constraint systems (Section 5.3), our contributions thus can be summarized as follows:

- we provide enhancements of each of these analyses which significantly increase their precision — but still are incomparable;
- since both analyses are expressed within the same framework, these improved versions can be integrated into one combined analysis;
- moreover, we construct a relational analysis based on the notion of common protecting mutexes;
- we provide implementations of the new analyses to compare their precision and efficiency.

We first introduce our local trace semantics, which is used to formalize the analyses (Section 5.2). Our analyses then are provided in Sections 5.4 and 5.5 The implementation is described in Section 5.6 and the experimental evaluation is provided in Section 5.7.

5.2. Preliminaries

We continue to use the imperative language with concurrency, introduced in Section 2.2, but pose a few additional requirements. Subsequently, we present for this language an interleaving semantics (Section 5.2.2) and our local trace semantics (Section 5.2.3). Although the two turn out to be equivalent w.r.t. assertion safety, which is sufficient for value analyses, we use the latter as common basis for thread-modular analyses. All analyses are formulated using side-effecting constraint systems, as described in Section 5.2.4.

5.2.1. Language

In this chapter, we use a restricted form of the concurrent imperative language from Section 2.2. We limit how global variables are used in programs, which simplifies analysis definitions. It is important to note that these restrictions are syntactic and do not limit the expressiveness of the language: complex statements which do not satisfy these rules can be transformed to only use sequences of the simpler primitives.

Global Variables. For each global variable $g \in \mathcal{G}$ and local variable $x \in \mathcal{X}$ of the same type, let the language contain the primitives $g \leftarrow x$ and $x \leftarrow g$ for writing and reading the global variable, respectively. Both primitives are *atomic*. This is enforced by introducing for each global $g \in \mathcal{G}$ a special *atomicity mutex* $m_g \in \mathcal{M}$, which is locked just before g is accessed and unlocked immediately after. Thus, writing to global g is encoded as $\mathsf{lock}(m_g); g \leftarrow x; \mathsf{unlock}(m_g)$ and reading from g as $\mathsf{lock}(m_g); x \leftarrow g; \mathsf{unlock}(m_g)$. Besides encoding atomicity, this ensures race freedom for the purposes of value analysis. While these mutexes are implicit in examples (for readability) and the implementation (for efficiency), they will be convenient for theoretical development. Finally, let $\mathcal{M}_\mathcal{G} \stackrel{\text{def}}{=} \{m_g \in \mathcal{M} \mid g \in \mathcal{G}\} \subseteq \mathcal{M}$ denote the set of these atomicity mutexes. We call all other mutexes $\mathcal{M} \setminus \mathcal{M}_\mathcal{G}$ *program mutexes*.

Other Primitives. In this chapter, all other primitives may *only* contain local variables. Hence, all conditions and more complex assignments involving global variables must be rewritten to first read the involved globals into (temporary) local variables on which to operate. For example, $\mathsf{g} \leftarrow \mathsf{g} + 1$ must be represented as

$$\begin{aligned}&\mathsf{lock}(m_\mathsf{g}); \mathsf{x} \leftarrow \mathsf{g}; \mathsf{unlock}(m_\mathsf{g});\\ &\mathsf{x} \leftarrow \mathsf{x} + 1;\\ &\mathsf{lock}(m_\mathsf{g}); \mathsf{g} \leftarrow \mathsf{x}; \mathsf{unlock}(m_\mathsf{g})\end{aligned}$$

As such, it reflects and correctly accounts for the non-atomic nature of the increment. In examples, this transformation is implicit for assertions on global variables for readability.

5.2.2. Interleaving Semantics

The interleaving semantics is the canonical concrete semantics for multi-threaded programs. An *interleaving* is a sequence of program configurations, each of which specifies

1. the values of global variables,
2. the control-flow location for each executing thread,
3. the values of local variables for each executing thread,
4. the state for each mutex (whether it is unlocked or locked by which thread).

An interleaving starts with an initial configuration where

1. global variables have their initial values,
2. the main thread is at location $\mathsf{st}_\mathsf{main}$ and no other threads exist,
3. local variables of main have arbitrary values,
4. all mutexes are unlocked.

In an interleaving, all adjacent configurations are such that the second is reached from the first via a valid transition, where an arbitrary thread executes a valid

primitive according to its control flow. In particular, a thread cannot lock a mutex which is already locked by another thread. Moreover, values of global and local variables are updated according to the primitive (e.g., assignment). Erhard et al. [Erh+25] give a formal description of our interleaving semantics.

An interleaving describes one possible execution of a given program. The interleaving semantics describes all possible executions of a given program as the set of all interleavings. An assertion is *violated in an interleaving* if and only if it evaluates to false in the interleaving. A program is *assertion-safe w.r.t. the interleaving semantics* if and only if there are no interleavings of the program in which any of its assertions are violated.

The interleaving semantics is a straightforward model of *sequential consistency* [Lam79], which we assume in our setting. Thus, we do not cover weak memory effects, such as out-of-order execution, store/load buffering, etc. As such, the interleaving semantics totally orders the actions of an execution. Hence, the interleaving semantics views the execution of a program from the perspective of an external global observer.

Such an observer can distinguish executions which are indistinguishable by the program itself, i.e., it can observe different executions (interleavings) whose differences cannot be observed within the program and, thus, do not affect its safety. For example, thread-local computations of two threads can be interleaved in many ways, all of which progress in the same way in terms of the local variables they manipulate.

One technique to account for this is *Mazurkiewicz equivalence* [Maz86], which is induced from a *commutativity relation* on actions (of different threads) whose relative order does not affect safety. The next section describes a different approach.

5.2.3. Local Trace Semantics

We introduce the local trace semantics [Sch+21a] which abandons the global observer and views the execution of a program from the perspective of a thread. This perspective includes all events which could have affected the thread and excludes those which definitely could not have. This in turn is determined by what the thread can observe from other threads. As such, it weakens the total ordering by the global observer into a partial order which imposes no order between unobservable events.

Observability. Multi-threaded programming constructs come with *observable* and *observing* primitive statements [Sch+23b, Section 3], e.g., those in Table 16. An observing statement in one thread can observe other threads at the corresponding observable statements. This notion of observability is analogous to the *synchronizes-with* relation of C11 [ISO11, Section 5.1.2.4]: release operations are observable and acquire operations are observing.

Here, we consider as observable/observing primitives only locking/unlocking of mutexes. Although we also want to support writing/reading of global variables,

Table 16. Pairs of observable and observing concurrency primitives by programming construct; constructs covered by this chapter are in **bold**

	Primitive	
Construct	Observable	Observing
Global variables (g)	$g \leftarrow x$	$x \leftarrow g$
Mutexes (m)	$\mathsf{unlock}(m)$	$\mathsf{lock}(m)$
Thread returns & joins	$\mathsf{return}\, x$	$x \leftarrow \mathsf{join}(t)$[a]
Condition variables (c)	$\mathsf{signal}(c)$	$\mathsf{wait}(c, m)$
Promises & futures (p)	$\mathsf{resolve}(p, x)$	$x \leftarrow \mathsf{get}(p)$
Channels (c)	$\mathsf{send}(c, x)$	$x \leftarrow \mathsf{receive}(c)$

[a] Using thread ID variables (t) and thread creation of the form $t \leftarrow \mathsf{create}(f)$, as described in Section 3.5.

the atomicity mutexes $\mathcal{M}_{\mathcal{G}}$ reduce their observability to mutexes.[44] Thus, they need not be considered separately. Creation of a thread could also be considered observable by the created thread, although it does not have a corresponding observing primitive.

Global Traces. A *global trace* is a partial order of thread configurations, each of which specifies

1. the thread,
2. the control-flow location for the thread,
3. the values of local variables for the thread,
4. the timestamp ($\mathbb{N}$) for the thread.

Note that a configuration does not specify the values of global variables. A global trace has a least element which is the initial configuration with

1. the main thread,
2. location $\mathrm{st}_{\mathsf{main}}$,
3. local variables have arbitrary values,
4. timestamp 0.

Configurations in a global trace are ordered by the *causality order*, which is the reflexive transitive closure of the union of the following:

Program order that orders configurations for each thread according to its control flow, as dictated by $\mathcal{A}$ for locations and $\leq$ for timestamps. The latter distinguishes configurations where a thread reaches the same location multiple times, e.g., in a loop.

[44] This effectively makes $g \leftarrow x$ also *observing* (the previous write it overwrites) and $x \leftarrow g$ also *observable* (totally ordering accesses to each global, as is the case for sequential consistency (per memory location)).

Create order that orders configurations with outgoing create primitives before the initial configurations of the corresponding (non-main) threads, such that:

1. each create primitive creates at most one thread,
2. each non-main thread is created by exactly one create primitive.

Observability orders that order, for each observable object (e.g., mutex), configurations with incoming observable primitives (and the least element) before configurations with incoming observing primitives, such that:

1. each observable primitive (e.g., unlock) (and the initial configuration) is observed by at most one observing primitive (e.g., lock),
2. each observing primitive (e.g., lock) observes exactly one observable primitive (e.g., unlock) (or the initial configuration).

For this reflexive transitive closure to be a partial order, it must also be antisymmetric. Semantically, this means that it has no causal cycles. Thus, global traces can be viewed as directed acyclic graphs.

Although configurations do not include the values of global variables, their values can be recovered from the global trace by identifying the last write to each global variable according to the causality order. Under sequential consistency, the last write is uniquely defined. For a global trace to be valid, we additionally require that, at each read of a global variable, the read value (stored into a local variable) agrees with the last write to the global variable (or it is the initial value if there is no last write to it). Schwarz et al. [Sch+21a] give a formal description of global traces.

Intuitively, a global trace corresponds to an interleaving if for every thread its subtrace in the global trace matches its subsequence in the interleaving on the control-flow locations and values of local variables of that thread. This is formalized as coincidence by Erhard et al. [Erh+25, Definition 1]. The following theorem establishes an equivalence between the interleaving semantics and the global trace semantics:

Theorem 2. *For a fixed program P:*

1. *For any interleaving, there exists a corresponding global trace.*
2. *For any global trace, there exists a corresponding interleaving.*

Proof. A proof can be found in the extended version of the original paper [Erh+24b, Appendix A.2]. ■

An assertion is *violated in a global trace* if and only if it evaluates to false in the global trace. A program is *assertion-safe w.r.t. the global trace semantics* if and only if there are no global traces of the program in which any of its assertions are violated.

Local Traces. A *local trace* is a global trace with a greatest element, whereas a global trace may contain multiple maximal elements. The *ego thread* of a local trace is the thread of its greatest element. A local trace records all other configurations that may have transitively influenced the ego thread. The local trace

semantics describes all local traces of a given program, which can be formulated as a fixpoint [Sch+21a]. The notions of an assertion being *violated in a local trace* and a program being *assertion-safe w.r.t. the local trace semantics* naturally carry over from global traces. Schwarz et al. [Sch+21a] give a formal description of local traces.

The following theorem establishes an equivalence of assertion safety between the global and the local trace semantics:

Theorem 3. *A program is assertion-safe w.r.t. the global trace semantics if and only if it is assertion-safe w.r.t. the local trace semantics.*

Proof. A proof can be found in the original paper [Erh+25, Theorem 1]. The proof is by contrapositive and boils down to proving for a fixed program P:

1. For any local trace violating an assertion, there exists a global trace violating the same assertion.
2. For any global trace violating an assertion, there exists a local trace violating the same assertion. ■

An equivalence of assertion safety between the interleaving semantics and the local trace semantics follows from the two theorems:

Corollary 1. *A program is assertion-safe w.r.t. interleaving semantics if and only if it is assertion-safe w.r.t. the local trace semantics.*

Proof. A proof can be found in the original paper [Erh+25, Theorem 1]. The proof is a composition of Theorems 2 and 3. ■

Hence, any *value* analysis which is sound w.r.t. the local trace semantics is also sound w.r.t. the interleaving semantics.

5.2.4. Constraints

Local trace semantics and their abstractions can be conveniently formalized using side-effecting constraint systems (on concrete or abstract states). Observable primitives produce side effects to unknowns corresponding to the observable object, while observing primitives depend on and incorporate values from those unknowns. That is, flows from observable to observing primitives are decomposed via side-effected unknowns.

Therefore, all of the following analyses are used to define a side-effecting constraint system of the following form:

$$
\begin{aligned}
(\eta, \eta\,[\mathsf{st}_{\mathsf{main}}, \varnothing]) &\sqsupseteq \mathsf{init}^\sharp; \\
(\eta, \eta\,[v, S \cup \{m\}]) &\sqsupseteq [\![\,[u, S], \mathsf{lock}(m)]\!]^\sharp\,\eta && \forall (u, \mathsf{lock}(m), v) \in \mathcal{A}, S \subseteq \mathcal{M}; \\
(\eta, \eta\,[v, S \setminus \{m\}]) &\sqsupseteq [\![\,[u, S], \mathsf{unlock}(m)]\!]^\sharp\,\eta && \forall (u, \mathsf{unlock}(m), v) \in \mathcal{A}, S \subseteq \mathcal{M}; \\
(\eta, \eta\,[v, S]) &\sqsupseteq [\![\,[u, S], s]\!]^\sharp\,\eta && \forall (u, s, v) \in \mathcal{A}, S \subseteq \mathcal{M}, \\
& && \quad s \neq \mathsf{lock}(m), s \neq \mathsf{unlock}(m);
\end{aligned}
$$

where each analysis defines the corresponding right-hand sides. Each right-hand side $[\![[u, S], s]\!]^\sharp$ is strict w.r.t. $\eta\,[u, S]$. Thus, for simplicity, we only present their definitions for the cases where $\eta\,[u, S] \neq \bot$.

Unless specified otherwise, the right-hand side for thread creation is the same for all analyses:

$$[\![[u, S], \mathsf{create}(f)]\!]^\sharp\,\eta \stackrel{\text{def}}{=} \textbf{let } (_, \sigma') = \mathsf{init}^\sharp \textbf{ in} \quad (\{[\mathrm{st}_f, \varnothing] \mapsto \sigma'\}, \eta\,[u, S]). \tag{10}$$

Side effects from $\mathsf{init}^\sharp$ can be ignored in create because they are already accounted for by the first constraint.

Remark 14 (Demand-driven solving). To solve such side-effecting constraint systems with a demand-driven solver (like TD [SV21]), some unknowns must be queried from the solver; otherwise it computes nothing. For the constraint system above, one needs to query unknowns $[\mathrm{ret}_{\mathsf{main}}, S]$ for all $S \subseteq \mathcal{M}$, because the main thread may return while still holding some mutexes.

A similar consideration applies to all other threads created in the program. Thus, the right-hand side for thread creation additionally queries the return node of the thread for all locksets (highlighted):

$$[\![[u, S], \mathsf{create}(f)]\!]^\sharp\,\eta \stackrel{\text{def}}{=} \textbf{let } (_, \sigma') = \mathsf{init}^\sharp \textbf{ in } \textbf{let } _ = \{\eta\,[\mathrm{ret}_f, S] \mid S \subseteq \mathcal{M}\} \textbf{ in } (\{[\mathrm{st}_f, \varnothing] \mapsto \sigma'\}, \eta\,[u, S]).$$

Since we do not return values from threads, the queried states themselves can just be ignored. The queries are simply necessary to force the demand-driven solver to compute anything in the created thread. Demand-driven solving additionally imposes certain connectivity requirements on CFGs which we do not elaborate here. □

5.3. Prior Analyses

This section describes a variety of thread-modular analyses from the literature and reformulates them in the common framework of side-effecting constraint systems, as set out in the previous section. First, Section 5.3.1 presents the simplest yet still sound thread-modular analyses. Then, Sections 5.3.2 and 5.3.3, respectively, present the analyses of Miné and Vojdani, which improve precision compared to the simple baseline in different ways. Finally, Examples 27 and 28 illustrate how their analyses are incomparable.

5.3.1. Baseline

Before considering thread-modular analyses which improve precision using scheduling information, this section describes the absolute baseline for sound thread-modular analysis. The following three analyses simply lift analyses for single-threaded programs to multi-threaded.

De, D'Souza, and Nasre [DDN11] propose one such lifting which assumes race freedom. In this case, it is sufficient to augment a single-threaded analysis with additional *may-synchronize-with* flows from each unlock to each lock of the same mutex. Thus, reading and writing of global variables can be performed just on thread-local states.

They use a pre-analysis to identify the set of necessary may-synchronize-with flows, which may need to be over-approximated. Side-effecting constraint systems avoid need for such a pre-analysis by decomposing these flows: $\mathsf{unlock}(m)$ side-effects to the unknown $[g, m]$ for each global $g \in \mathcal{G}$, and $\mathsf{lock}(m)$ joins the state from that unknown. Hence, the identification of such flows can benefit from more precise may-alias information from the main analysis. The corresponding right-hand sides are the following:

$$
\begin{aligned}
\mathsf{init}^\sharp &\stackrel{\text{def}}{=} \textbf{let } \sigma = \{x \mapsto \top \mid x \in \mathcal{X}\} \cup \{g \mapsto \bot \mid g \in \mathcal{G}\} \textbf{ in} \\
&\quad \textbf{let } \rho = \{[g, m] \mapsto 0^\sharp \mid g \in \mathcal{G}, m \in \mathcal{M}\} \textbf{ in} \\
&\quad (\rho, \sigma), \\
[\![[u, S], g \leftarrow x]\!]^\sharp \eta &\stackrel{\text{def}}{=} \textbf{let } \sigma = \eta\,[u, S] \textbf{ in} \\
&\quad (\varnothing, \sigma[g \mapsto \sigma\, x]), \\
[\![[u, S], x \leftarrow g]\!]^\sharp \eta &\stackrel{\text{def}}{=} \textbf{let } \sigma = \eta\,[u, S] \textbf{ in} \\
&\quad (\varnothing, \sigma[x \mapsto \sigma\, g]), \\
[\![[u, S], \mathsf{lock}(m)]\!]^\sharp \eta &\stackrel{\text{def}}{=} \textbf{let } \sigma = \eta\,[u, S] \textbf{ in} \\
&\quad \textbf{let } \sigma' = \{g \mapsto \eta\,[g, m] \mid g \in \mathcal{G}\} \textbf{ in} \\
&\quad (\varnothing, \sigma \sqcup \sigma'), \\
[\![[u, S], \mathsf{unlock}(m)]\!]^\sharp \eta &\stackrel{\text{def}}{=} \textbf{let } \sigma = \eta\,[u, S] \textbf{ in} \\
&\quad \textbf{let } \rho = \{[g, m] \mapsto \sigma\, g \mid g \in \mathcal{G}\} \textbf{ in} \\
&\quad (\rho, \sigma),
\end{aligned}
$$

where $0^\sharp$ (in $\mathsf{init}^\sharp$) is the abstract initial value for the global variable.

Since our atomicity mutexes guarantee race freedom by construction, it applies without restriction in our setting. Moreover, this makes flows along program mutexes redundant and allows the analysis to be reformulated using just unknowns $[g]$ (corresponding to $[g, m_g]$ above):

$$
\begin{aligned}
\mathsf{init}^\sharp &\stackrel{\text{def}}{=} \textbf{let } \sigma = \{x \mapsto \top \mid x \in \mathcal{X}\} \cup \{g \mapsto \bot \mid g \in \mathcal{G}\} \textbf{ in} \\
&\quad \textbf{let } \rho = \{[g] \mapsto 0^\sharp \mid g \in \mathcal{G}\} \textbf{ in} \\
&\quad (\rho, \sigma), \\
[\![[u, S], \mathsf{lock}(m_g)]\!]^\sharp \eta &\stackrel{\text{def}}{=} \textbf{let } \sigma = \eta\,[u, S] \textbf{ in} \\
&\quad (\varnothing, \sigma[g \mapsto \sigma\, g \sqcup \eta\,[g]]), \\
[\![[u, S], \mathsf{lock}(m)]\!]^\sharp \eta &\stackrel{\text{def}}{=} (\varnothing, \eta\,[u, S]),
\end{aligned}
$$

$$[\![[u, S], \mathsf{unlock}(m_g)]\!]^\sharp\, \eta \stackrel{\text{def}}{=} \textbf{let } \sigma = \eta\, [u, S] \textbf{ in } (\{[g] \mapsto \sigma\, g\}, \sigma),$$
$$[\![[u, S], \mathsf{unlock}(m)]\!]^\sharp\, \eta \stackrel{\text{def}}{=} (\varnothing, \eta\, [u, S]).$$

Miné [Min12, Section 3.3] proposes an abstract (unscheduled) interference semantics, which lacks synchronization primitives and, thus, propagates values of globals directly from writes to reads. This analysis can also be formulated as a side-effecting constraint system as follows:

$$\begin{aligned}
\mathsf{init}^\sharp &\stackrel{\text{def}}{=} \textbf{let } \sigma = \{x \mapsto \top \mid x \in \mathcal{X}\} \cup \{g \mapsto \bot \mid g \in \mathcal{G}\} \textbf{ in} \\
&\quad\ \textbf{let } \rho = \{[g] \mapsto 0^\sharp \mid g \in \mathcal{G}\} \textbf{ in} \\
&\quad\ (\rho, \sigma), \\
[\![[u, S], g \leftarrow x]\!]^\sharp\, \eta &\stackrel{\text{def}}{=} \textbf{let } \sigma = \eta\, [u, S] \textbf{ in } (\{[g] \mapsto \sigma\, x\}, \sigma[g \mapsto \sigma\, x]), \\
[\![[u, S], x \leftarrow g]\!]^\sharp\, \eta &\stackrel{\text{def}}{=} \textbf{let } \sigma = \eta\, [u, S] \textbf{ in } (\varnothing, \sigma[x \mapsto \sigma\, g \sqcup \eta\, [g]]), \\
[\![[u, S], \mathsf{lock}(m)]\!]^\sharp\, \eta &\stackrel{\text{def}}{=} (\varnothing, \eta\, [u, S]), \\
[\![[u, S], \mathsf{unlock}(m)]\!]^\sharp\, \eta &\stackrel{\text{def}}{=} (\varnothing, \eta\, [u, S]).
\end{aligned}$$

In our setting, this analysis is essentially equivalent to the above: the unknowns $[g]$ are accessed directly in the read/write constraints instead of their surrounding mutex operations.

Vojdani [Voj10, Section 2.4] proposes an even simpler variation, which does not keep values of global variables in local states at all. Its side-effecting constraint system, adapted to our notation, is the following:

$$\begin{aligned}
\mathsf{init}^\sharp &\stackrel{\text{def}}{=} \textbf{let } \sigma = \{x \mapsto \top \mid x \in \mathcal{X}\} \textbf{ in} \\
&\quad\ \textbf{let } \rho = \{[g] \mapsto 0^\sharp \mid g \in \mathcal{G}\} \textbf{ in} \\
&\quad\ (\rho, \sigma), \\
[\![[u, S], g \leftarrow x]\!]^\sharp\, \eta &\stackrel{\text{def}}{=} \textbf{let } \sigma = \eta\, [u, S] \textbf{ in } (\{[g] \mapsto \sigma\, x\}, \sigma), \\
[\![[u, S], x \leftarrow g]\!]^\sharp\, \eta &\stackrel{\text{def}}{=} \textbf{let } \sigma = \eta\, [u, S] \textbf{ in } (\varnothing, \sigma[x \mapsto \eta\, [g]]), \\
[\![[u, S], \mathsf{lock}(m)]\!]^\sharp\, \eta &\stackrel{\text{def}}{=} (\varnothing, \eta\, [u, S]), \\
[\![[u, S], \mathsf{unlock}(m)]\!]^\sharp\, \eta &\stackrel{\text{def}}{=} (\varnothing, \eta\, [u, S]).
\end{aligned}$$

This analysis is essentially equivalent to the previous one: global variables are omitted from local states altogether.

5.3.2. Miné's Analysis

Miné [Min12, Section 4.5] proposes an abstract *scheduled* interference semantics, which also supports synchronization using mutexes (in contrast to the previous section). It summarizes the effect of each thread on the global state using two kinds of inter-thread communications, i.e., interferences:

1. *weakly consistent* (weak for short), which propagate values from writes to reads;
2. *well synchronized* (synchronized for short), which propagate values from unlocks to locks.

To benefit from synchronization at all, both kinds of interferences are partitioned w.r.t. what we call *background locksets*: the lockset S of a thread before locking a mutex m or after unlocking m (so $m \notin S$). Flows from $\mathsf{unlock}(m)$ of a thread with background lockset S' to $\mathsf{lock}(m)$ of a thread with background lockset S can be limited to those where $S \cap S' = \varnothing$, i.e., they are mutually exclusive.

To capture the essence of Miné's analysis and adapt it to our setting, we make the following changes (mostly simplifications):

1. Remove thread priorities. (Miné focuses on ARINC 653 mono-processor real-time scheduling, while we allow multiprocessor and non–real-time systems as per Miné [Min12, Section 4.4.5].)
2. Remove (unique) thread IDs ($t \in \mathcal{T}$). (We consider an unbounded number of thread instances. Miné [Min12, Section 3.3] makes a similar suggestion.)
3. Add dynamic thread creation. (Miné considers embedded software with a fixed set of threads which are created a priori.)
4. Remove yield and islocked statements. (These are only useful in the presence of priorities.)
5. Remove must-unlocked mutex sets from configurations ($u \in 2^{\mathcal{M}}$). (Miné only uses these to benefit from islocked statements.)
6. Use a non-relational domain for local states. (Miné allows a relational domain thread-locally, but still handles interferences non-relationally.)
7. Remove alarms from states ($\Omega \subseteq \mathcal{L}$). (These are an orthogonal feature.)

The analysis is originally formulated as a fixpoint over the execution of all thread bodies, which are abstractly interpreted over program syntax, and which collect and apply a set of interferences. We reformulate this analysis as a side-effecting constraint system which uses additional unknowns $[g, m, S']$ where $g \in \mathcal{G}$, $m \in \mathcal{M}$ and $S' \subseteq \mathcal{M}$ is the background lockset.

Technically, in our setting with the atomicity mutexes, all interferences are synchronized and, thus, weak interferences are unnecessary because they already correspond to the unknowns $[g, m_g, S']$ ($g \in \mathcal{G}$, $S' \subseteq \mathcal{M}$). However, in the original formulation, weak interferences are not joined into the local state when weakly reading or writing. Not doing so can be more precise because it avoids propagating

those values as writes at unlocks in the same thread. Therefore, to remain faithful to Miné's analysis, we will handle each atomicity mutex m_g separately to get similar behavior.

Furthermore, the original formulation relies on thread IDs to convert weak interferences from a thread into synchronized interferences when unlocking. Miné [Min12, Footnote 5, p. 31] remarks that this over-approximates the set of variables written by the thread (during that mutually excluded section). After omitting thread IDs, we achieve the same by explicitly tracking in local states a map $W : \mathcal{M} \to 2^{\mathcal{G}}$ of variables possibly written by the current thread while holding some mutex.[45,46]

We now give the right-hand sides for the side-effecting constraint system formulation of Miné's analysis, starting with initialization:

$$\begin{aligned}\mathsf{init}^\sharp \stackrel{\text{def}}{=} \ & \textbf{let } W = \{m \mapsto \varnothing \mid m \in \mathcal{M}\} \textbf{ in} \\ & \textbf{let } \sigma = \{x \mapsto \top \mid x \in \mathcal{X}\} \cup \{g \mapsto 0^\sharp \mid g \in \mathcal{G}\} \textbf{ in} \\ & (\varnothing, (W, \sigma)).\end{aligned}$$

Unlike all other analyses presented here, Miné's does not side-effect initial values of globals. Instead, it propagates current values of globals (which may be initial) directly to ancestor threads via local states, instead of relying on interferences.[47] Thus, Eq. (10) is replaced by:

$$\begin{aligned}[\![[u, S], \mathsf{create}(f)]\!]^\sharp \eta \stackrel{\text{def}}{=} \ & \textbf{let } (W, \sigma) = \eta\,[u, S] \textbf{ in} \\ & \textbf{let } W' = \{m \mapsto \varnothing \mid m \in \mathcal{M}\} \textbf{ in} \\ & \textbf{let } \sigma' = \sigma[x \mapsto \top \mid x \in \mathcal{X}] \textbf{ in} \\ & \textbf{let } \rho = \{[\mathrm{st}_f, \varnothing] \mapsto (W', \sigma')\} \textbf{ in} \\ & (\rho, (W, \sigma)).\end{aligned}$$

As explained above, we handle weak interferences directly in the right-hand sides of writing and reading, making locking and unlocking of m_g-s no-ops:

$$\begin{aligned}[\![[u, S], g \leftarrow x]\!]^\sharp \eta \stackrel{\text{def}}{=} \ & \textbf{let } (W, \sigma) = \eta\,[u, S] \textbf{ in} \\ & \textbf{let } \sigma' = \sigma[g \mapsto \sigma\, x] \textbf{ in} \\ & \textbf{let } W' = W[m \mapsto W\, m \cup \{g\} \mid m \in S] \textbf{ in} \\ & \textbf{let } \rho = \{[g, m_g, S \setminus \{m_g\}] \mapsto \sigma'\, g\} \textbf{ in} \\ & (\rho, (W', \sigma')),\end{aligned}$$

[45] The published reformulation [Sch+21b, Appendix A] slightly undermines Miné's analysis by using a set $W : 2^{\mathcal{G}}$ of possibly written globals instead, thereby ignoring mutexes held at the time of writing: `https://github.com/goblint/analyzer/pull/1613`.

[46] Since W is tracked in local states, it is flow-sensitive within a thread. This is actually slightly more precise than Miné's formulation, where the over-approximation is flow-insensitive: `https://github.com/goblint/analyzer/pull/1613`.

[47] This can make Miné's analysis more precise than others in ways not considered by the original paper [Sch+21a] (see Example 34, p. 152), which instead assumed that globals start out uninitialized and are initialized by explicit assignment. Goblint tries to compensate for this by distinguishing single- and multi-threaded modes (Section 5.6.1).

$$[\![[u, S], x \leftarrow g]\!]^\sharp\, \eta \stackrel{\text{def}}{=} \textbf{let } (W, \sigma) = \eta\, [u, S] \textbf{ in}$$
$$\textbf{let } g' = \bigsqcup \{\eta\, [g, m_g, S'] \mid S' \subseteq \mathcal{M}, S' \cap S = \varnothing\} \textbf{ in}$$
$$(\varnothing, (W, \sigma[x \mapsto \sigma\, g \sqcup g'])),$$
$$[\![[u, S], \mathsf{lock}(m_g)]\!]^\sharp\, \eta \stackrel{\text{def}}{=} \eta\, [u, S],$$
$$[\![[u, S], \mathsf{unlock}(m_g)]\!]^\sharp\, \eta \stackrel{\text{def}}{=} \eta\, [u, S].$$

Synchronized interferences via $m \in \mathcal{M} \setminus \mathcal{M}_{\mathcal{G}}$ are handled as follows:[48]

$$[\![[u, S], \mathsf{lock}(m)]\!]^\sharp\, \eta \stackrel{\text{def}}{=} \textbf{let } (W, \sigma) = \eta\, [u, S] \textbf{ in}$$
$$\textbf{let } \sigma' =$$
$$\left\{ g \mapsto \bigsqcup \{\eta\, [g, m, S'] \mid S' \subseteq \mathcal{M}, \; S' \cap S = \varnothing\} \;\middle|\; g \in \mathcal{G} \right\}$$
$$\textbf{in}$$
$$(\varnothing, (W, \sigma \sqcup \sigma')),$$
$$[\![[u, S], \mathsf{unlock}(m)]\!]^\sharp\, \eta \stackrel{\text{def}}{=} \textbf{let } (W, \sigma) = \eta\, [u, S] \textbf{ in}$$
$$\textbf{let } \rho = \{[g, m, S \setminus \{m\}] \mapsto \sigma\, g \mid g \in W\, m\} \textbf{ in}$$
$$(\rho, (W, \sigma)).$$

For examples of this analysis in action, see Examples 27 and 28 at the end of the following section.

5.3.3. Vojdani's Analysis

Vojdani [Voj10, Section 5.2; Voj+16] proposes an analysis called *privatization*. It assumes that for every global variable g there is a set of *protecting mutexes* $\mathcal{M}_{\text{rw}}[g] \subseteq \mathcal{M}$ definitely held whenever g is accessed (read or written). Thanks to our atomicity assumption, the set $\mathcal{M}_{\text{rw}}[g]$ is non-empty since $m_g \in \mathcal{M}_{\text{rw}}[g]$ always holds. Since m_g is locked/unlocked immediately before/after g is accessed, $m_{g'} \notin \mathcal{M}_{\text{rw}}[g]$ for every $g' \neq g$. Thus, if no program mutex protects all accesses of g, then $\mathcal{M}_{\text{rw}}[g] = \{m_g\}$. Conversely, for each mutex m, let $\mathcal{G}_{\text{rw}}[m] \stackrel{\text{def}}{=} \{g \in \mathcal{G} \mid m \in \mathcal{M}_{\text{rw}}[g]\}$ be the set of *protected globals*, which could be empty. However, $\mathcal{G}_{\text{rw}}[m_g] = \{g\}$ for each $g \in \mathcal{G}$ due to the properties above.

When a thread holds a protecting mutex, it can manipulate the protected variable g completely thread-locally; otherwise, the thread has to synchronize its value with its global invariant, represented with the constraint unknown $[g]$. The synchronization means that at every local state where g is unprotected the value of $[g]$ is joined into the local state, and vice versa using a side effect.

[48] A slightly improved definition of $\mathsf{lock}(m)$ could update $W\, m$ to $\varnothing$ such that the following $\mathsf{unlock}(m)$ would only produce side effects for those globals written in the current mutually excluded section, not previous ones corresponding to the same mutex m.

Synchronizing at every local state is excessive: if a thread has not written g nor locked/unlocked one of its protecting mutexes, then it will not join in or side-effect any value which it already had not before. Even if the extra synchronization makes no difference to analysis results, it gives rise to poor *dependency structure*. As black boxes, the constraints still have dependencies on other unknowns whose values are irrelevant. When the values of such spurious dependencies change, the constraint solver is forced to perform unnecessary recomputations.

We give the right-hand sides for a reasonable version of this analysis, which avoids excessive synchronization, starting with

$$\begin{aligned}\mathsf{init}^\sharp \stackrel{\text{def}}{=} {} & \textbf{let } \sigma = \{x \mapsto \top \mid x \in \mathcal{X}\} \cup \{g \mapsto \bot \mid g \in \mathcal{G}\} \textbf{ in} \\ & \textbf{let } \rho = \{[g] \mapsto 0^\sharp \mid g \in \mathcal{G}\} \textbf{ in} \\ & (\rho, \sigma).\end{aligned}$$

It suffices to join $[g]$ into the local state whenever g becomes protected, i.e., one of its protecting mutexes is locked while none were held before:[49]

$$\begin{aligned}[\![[u,S], \mathsf{lock}(m)]\!]^\sharp\, \eta \stackrel{\text{def}}{=} {} & \textbf{let } \sigma = \eta\,[u,S] \textbf{ in} \\ & \textbf{let } \sigma' = \\ & \quad \sigma[g \mapsto \sigma\, g \sqcup \eta\,[g] \mid g \in \mathcal{G}_{\mathrm{rw}}[m], S \cap \mathcal{M}_{\mathrm{rw}}[g] = \varnothing] \\ & \textbf{in} \\ & (\varnothing, \sigma').\end{aligned}$$

For $m = m_g \in \mathcal{M}_{\mathcal{G}}$ this simplifies to:

$$\begin{aligned}[\![[u,S], \mathsf{lock}(m_g)]\!]^\sharp\, \eta \stackrel{\text{def}}{=} {} & \textbf{let } \sigma = \eta\,[u,S] \textbf{ in} \\ & \textbf{let } \sigma' = \sigma[g \mapsto \sigma\, g \sqcup \eta\,[g] \mid \mathcal{M}_{\mathrm{rw}}[g] = \{m_g\}] \textbf{ in} \\ & (\varnothing, \sigma').\end{aligned}$$

All reads and writes involving g are performed thread-locally, as long as at least one of its protecting mutexes is held:[50]

$$\begin{aligned}[\![[u,S], g \leftarrow x]\!]^\sharp\, \eta \stackrel{\text{def}}{=} {} & \textbf{let } \sigma = \eta\,[u,S] \textbf{ in} \\ & (\varnothing, \sigma[g \mapsto \sigma\, x]), \\ [\![[u,S], x \leftarrow g]\!]^\sharp\, \eta \stackrel{\text{def}}{=} {} & \textbf{let } \sigma = \eta\,[u,S] \textbf{ in} \\ & (\varnothing, \sigma[x \mapsto \sigma\, g]).\end{aligned}$$

[49] The condition $m \in \mathcal{M}_{\mathrm{rw}}[g]$ in the definition of $g \in \mathcal{G}_{\mathrm{rw}}[m]$ avoids excessive joining of g at an unrelated $\mathsf{lock}(m)$ before g has become protected at all. This check causes the non-monotonicity described in Remark 15 below.

[50] This does not need to be checked due to the definition of $\mathcal{M}_{\mathrm{rw}}[g]$.

The local value of g must be side-effected to $[g]$ when it becomes unprotected, i.e., the last held protecting mutex is unlocked:[51]

$$\begin{aligned}[[\,[u,S],\mathsf{unlock}(m)\,]]^\sharp\,\eta \stackrel{\text{def}}{=}\ & \textbf{let}\ \sigma = \eta\,[u,S]\ \textbf{in}\\ & \textbf{let}\ \rho =\\ & \quad \{[g] \mapsto \sigma\,g \mid g \in \mathcal{G}_{\text{rw}}[m],\\ & \qquad\qquad (S \setminus \{m\}) \cap \mathcal{M}_{\text{rw}}[g] = \varnothing\}\\ & \textbf{in}\\ & (\rho,\sigma).\end{aligned}$$

For $m = m_g \in \mathcal{M}_{\mathcal{G}}$ this simplifies to:

$$\begin{aligned}[[\,[u,S],\mathsf{unlock}(m_g)\,]]^\sharp\,\eta \stackrel{\text{def}}{=}\ & \textbf{let}\ \sigma = \eta\,[u,S]\ \textbf{in}\\ & \textbf{let}\ \rho = \{[g] \mapsto \sigma\,g \mid \mathcal{M}_{\text{rw}}[g] = \{m_g\}\}\ \textbf{in}\\ & (\rho,\sigma).\end{aligned}$$

The sets $\mathcal{M}_{\text{rw}}[g]$ encode a form of race freedom, which is coarser (and, thus, more precise) than just using all atomicity mutexes (cf. Section 5.3.1). They can be used to avoid joining in unnecessary values during locking and to suppress unnecessary side effects during unlocking.

Example 26 (The essence of privatization)**.** Consider the following program, where the protecting mutexes for g are $\mathcal{M}_{\text{rw}}[\mathtt{g}] = \{\mathtt{m}, m_{\mathtt{g}}\}$:

```
1  int g = 0;
2  mutex m;

3  void main() {
4    create(t);
5    lock(m);
6      int x = g;
7      assert(x == 0);
8    unlock(m);
9  }

10 void t() {
11   lock(m);
12     g = 1;
13     int y = g;
14     assert(y == 1);
15     g = 0;
16   unlock(m);
17 }
```

We apply Vojdani's analysis to this program using sets of integers for abstracting integer values. Initialization side-effects $\{0\}$ to $[\mathtt{g}]$ and returns the initial state

$$\{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{g} \mapsto \bot\}.$$

Locking m in the thread t joins $[\mathtt{g}]$ into the local state, yielding

$$\{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{g} \mapsto \{0\}\}.$$

[51] The condition $m \in \mathcal{M}_{\text{rw}}[g]$ in the definition of $g \in \mathcal{G}_{\text{rw}}[m]$ avoids excessive republishing of g at an unrelated $\mathsf{unlock}(m)$ after g has already become unprotected before. This check causes the non-monotonicity described in Remark 15 below.

The subsequent assignment to g (together with the implicit $\mathsf{lock}(m_{\mathtt{g}})$ and $\mathsf{unlock}(m_{\mathtt{g}})$) updates the local state to

$$\{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{g} \mapsto \{1\}\}$$

and triggers no side effects, since the global is still protected. The following read of g into y acts thread-locally, yielding

$$\{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \{1\}, \mathtt{g} \mapsto \{1\}\}.$$

Thus, the assertion in the thread t is proven. The final assignment to g updates the local state to

$$\{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \{1\}, \mathtt{g} \mapsto \{0\}\}$$

and triggers no side effects, since the global is still protected. Unlocking m in the thread t does not change the local state, but side-effects $\{0\}$ to $[\mathtt{g}]$ since the global becomes unprotected. This side-effected value was already included in $[\mathtt{g}]$, so it has no additional consequences.

Meanwhile, locking m in the main thread joins $[\mathtt{g}]$ into the local state, yielding

$$\{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{g} \mapsto \{0\}\}.$$

The following read of g into x acts thread-locally, yielding

$$\{\mathtt{x} \mapsto \{0\}, \mathtt{y} \mapsto \top, \mathtt{g} \mapsto \{0\}\}.$$

Thus, the assertion in the main thread is proven. Unlocking m in the main thread does not change the local state, but side-effects $\{0\}$ to $[\mathtt{g}]$ since the global becomes unprotected, but this was also already included. Therefore, $[\mathtt{g}]$ is $\{0\}$ in the end. □

This example captures *the essence of privatization*. All analyses in Section 5.3.1 cannot prove the two assertions because the value 1 written to g by the thread t flows to the read in the main thread. Additionally, the read from g in the thread t (or the implicit $\mathsf{lock}(m_{\mathtt{g}})$ before it) also joins in the initial value 0. Miné's analysis can also prove both assertions.

Remark 15 (Analysis of protecting mutexes). Above we assumed the sets $\mathcal{M}_{\mathrm{rw}}[g]$ as given for simplicity of presentation. However, these can be computed by a simple pre-analysis. Alternatively, a side-effecting constraint system can compute them *jointly* with the above analysis, allowing for increased precision compared to a pre-analysis.

For that, $\mathcal{M}_{\mathrm{rw}}[g]$ should also be considered as an unknown of the constraint system, taking values from the domain $2^{\mathcal{M}}$ (ordered by superset). They should be initialized to $\mathcal{M}$ in the extended right-hand side (highlighted in ρ):

$$\begin{aligned}\mathsf{init}^\sharp \stackrel{\text{def}}{=}\ & \textbf{let } \sigma = \{x \mapsto \top \mid x \in \mathcal{X}\} \cup \{g \mapsto \bot \mid g \in \mathcal{G}\} \textbf{ in} \\ & \textbf{let } \rho = \{[g] \mapsto 0^\sharp \mid g \in \mathcal{G}\} \cup \{\mathcal{M}_{\mathrm{rw}}[g] \mapsto \mathcal{M} \mid g \in \mathcal{G}\} \textbf{ in} \\ & (\rho, \sigma).\end{aligned}$$

The right-hand sides for reading and writing of globals are extended by side-effecting the current lockset S to these unknowns (highlighted):

$$[\![[u, S], g \leftarrow x]\!]^\sharp \eta \stackrel{\text{def}}{=} \textbf{let } \sigma = \eta\,[u, S] \textbf{ in } (\{\mathcal{M}_{\text{rw}}[g] \mapsto S\}, \sigma[g \mapsto \sigma\, x]),$$

$$[\![[u, S], x \leftarrow g]\!]^\sharp \eta \stackrel{\text{def}}{=} \textbf{let } \sigma = \eta\,[u, S] \textbf{ in } (\{\mathcal{M}_{\text{rw}}[g] \mapsto S\}, \sigma[x \mapsto \sigma\, g]).$$

Since the domain for the new unknowns is ordered by superset (instead of subset), the accumulating side effects yield the intersection of all locksets at accesses. Technically, other right-hand sides should use $\eta\,(\mathcal{M}_{\text{rw}}[g])$ instead of just $\mathcal{M}_{\text{rw}}[g]$ to query the protecting mutexes on the fly.

Note that the right-hand sides of the joint constraint system are non-monotonic in the unknowns $\mathcal{M}_{\text{rw}}[g]$ ($g \in \mathcal{G}$). If $\mathcal{M}_{\text{rw}}[g]$ shrinks (i.e., increases in its domain order) to no longer contain m, then:

- unlock(m) will no longer side-effect to $[g]$, due to the check $m \in \mathcal{M}_{\text{rw}}[g]$ in the definition of $g \in \mathcal{G}_{\text{rw}}[m]$ that avoids excess side effects;
- lock(m) will no longer join in $[g]$, due to the check $m \in \mathcal{M}_{\text{rw}}[g]$ in the definition of $g \in \mathcal{G}_{\text{rw}}[m]$ that avoids excess joins.

Vojdani's original analysis is monotonic, because it does not use the checks $m \in \mathcal{M}_{\text{rw}}[g]$. □

Since this analysis requires common protecting mutexes to yield a precision improvement, there are many cases where Miné's analysis is more precise because it does not require this.

Example 27 (Miné ⊏ Vojdani)**.** Consider the following program:

```
1  int g = 0;
2  mutex m1, m2;

3  void main() {
4    create(t);
5    lock(m1);
6      int x = g;
7      // ...
8  }

9  void t() {
10   lock(m1);
11     g = 1;
12     lock(m2);
13       g = 2;
14     unlock(m1); // !
15     g = 3;
16   unlock(m2); // !
17 }
```

Vojdani's analysis finds no protecting mutexes for g other than m_{g} itself, i.e., $\mathcal{M}_{\text{rw}}[\text{g}] = \{m_{\text{g}}\}$. Therefore, every write immediately publishes the written value. Hence, the local x receives values $\{0, 1, 2, 3\}$.

According to Miné's analysis, the value $\{2\}$ is published at the operation unlock(m1) and $\{3\}$ is published at unlock(m2). Thus, the local x receives the value $\{0, 2, 3\}$. □

Despite its conceptual simplicity and perhaps to our surprise, it turns out Vojdani's analysis is *not* subsumed by Miné's approach but is incomparable.

Example 28 (Vojdani ⊏ Miné)**.** Consider the following program:

```
1  int g = 0;
2  mutex m1, m2;

3  void main() {
4    create(t);
5    lock(m1);
6      lock(m2);
7        int x = g;
8        // ...
9  }

10 void t() {
11   lock(m1);
12     lock(m2);
13       g = 42;
14     unlock(m1); // !
15     g = 17;
16   unlock(m2); // !
17 }
```

According to Miné's analysis, the value $\{42\}$ is merged into the local state at the operation lock(m1), while the value $\{17\}$ is merged at the operation lock(m2). Thus, the local x receives the value $\{0, 17, 42\}$.

Vojdani's analysis, on the other hand, finds out that all accesses to g are protected by the mutex m2, i.e., $\mathcal{M}_{\text{rw}}[\text{g}] = \{\text{m2}, m_{\text{g}}\}$. Unlocking of m1, therefore, does not publish the intermediately written value $\{42\}$, but only the final value $\{17\}$ at unlock(m2) is published. Therefore, the local x only receives the value $\{0, 17\}$. □

5.4. Improved Analyses

Although Miné's analysis and Vojdani's analysis are quite different, we identify two deficiencies that they both share.

The first deficiency is *eager reading*, i.e., importing values of globals into the local state at each lock(m) operation regardless of whether a global will be read by the thread (in that mutually exclusive section). Miné's analysis joins in values of *all* globals, albeit from a possibly limited set of unknowns. Vojdani's analysis only joins in values of globals protected by m, and only when they become protected. Since both analyses use join ($\sqcup$), this can worsen precision (or leave it unaltered at best), but never improve precision. This behavior has a compounding effect together with the following deficiency.

The second deficiency is *republishing* of values that the thread has not written itself, but rather imported from other threads. Side effects from unlock(m) in both analyses publish values from the local state, which not only contains values that the thread has written itself, but also those it has imported. Side-effecting a value to an unknown which it was read from has no impact, because side-effected unknowns simply accumulate their values (see Remark 1, p. 34). However, by joining everything, the local states cannot distinguish which contributions came from where; thus, values imported from one unknown may get published to an unrelated one unnecessarily.

Not only do the deficiencies cause loss of precision, they also yield poor *dependency structure* (both separately and together). Eager reading joins in values from some flow-insensitive unknowns, creating a dependency, but these values may never get used. Republishing can create additional side effects which change values of some flow-insensitive unknowns. Both lead to useless recomputations by the constraint solver.

To overcome the first limitation we want *lazy reading*: locking does not import values into the local state σ at all; instead, everything observable is accounted for when actually reading a global variable, i.e., $\mathsf{lock}(m)$ should not alter σ. Miné's analysis only does so for weak interferences, but not synchronized ones. In fact, lazy reading would also overcome the second limitation by only storing in σ those globals that the ego thread has written.

However, lazy reading also comes with a caveat. If the ego thread has definitely written to a global under sufficient protection and no locking/unlocking affecting the global has occurred since, then the thread should only read its own last write (*the essence of privatization*, see Example 26). Both Miné's analysis and Vojdani's analysis required no additional bookkeeping to detect and handle such cases. As we will see below, adopting lazy reading more-or-less forces the analysis to maintain some additional information to preserve this precision.

We first enhance Vojdani's and Miné's analyses with lazy reading, obtaining *Protection-Based Reading* (Section 5.4.1) and *Lock-Centered Reading* (Section 5.4.2), respectively. Then, we propose *Write-Centered Reading* (Section 5.4.3) which lifts an assumption about program-wide protecting mutexes from Protection-Based Reading. Interestingly, Lock-Centered Reading and Write-Centered Reading are still incomparable. Finally, we propose a combined analysis (Section 5.4.4) which is more precise than both. The lattice in Fig. 15 presents an overview of the analyses considered here and their precision relationships.

This section focuses on non-relational analyses, while relational analyses are explored in Section 5.5. Furthermore, here we do not use thread IDs for additional precision because their benefits are largely orthogonal (see Section 5.5.2).

5.4.1. Protection-Based Reading

We enhance Vojdani's analysis (Section 5.3.3) in two ways.

First, we weaken the notion of protecting mutexes. Instead of assuming for every global g a set of mutexes $\mathcal{M}_{\mathrm{rw}}[g]$ protecting both reads and writes of g, we now just assume a set of *write-protecting mutexes* $\mathcal{M}_{\mathrm{w}}[g] \subseteq \mathcal{M}$ only protecting its writes. This can lead to an improvement in precision when $\mathcal{M}_{\mathrm{w}}[g] \supset \mathcal{M}_{\mathrm{rw}}[g]$, e.g., in Example 31 below. Analogously, the set $\mathcal{M}_{\mathrm{w}}[g]$ is non-empty since $m_g \in \mathcal{M}_{\mathrm{w}}[g]$ always holds thanks to our atomicity assumption. Since m_g is locked/unlocked immediately before/after g is accessed, $m_{g'} \notin \mathcal{M}_{\mathrm{w}}[g]$ for every $g' \neq g$. Thus, if no program mutex protects all writes of g, then $\mathcal{M}_{\mathrm{w}}[g] = \{m_g\}$. Conversely, for each mutex m, let $\mathcal{G}_{\mathrm{w}}[m] \stackrel{\text{def}}{=} \{g \in \mathcal{G} \mid m \in \mathcal{M}_{\mathrm{w}}[g]\}$ be the set of *write-protected*

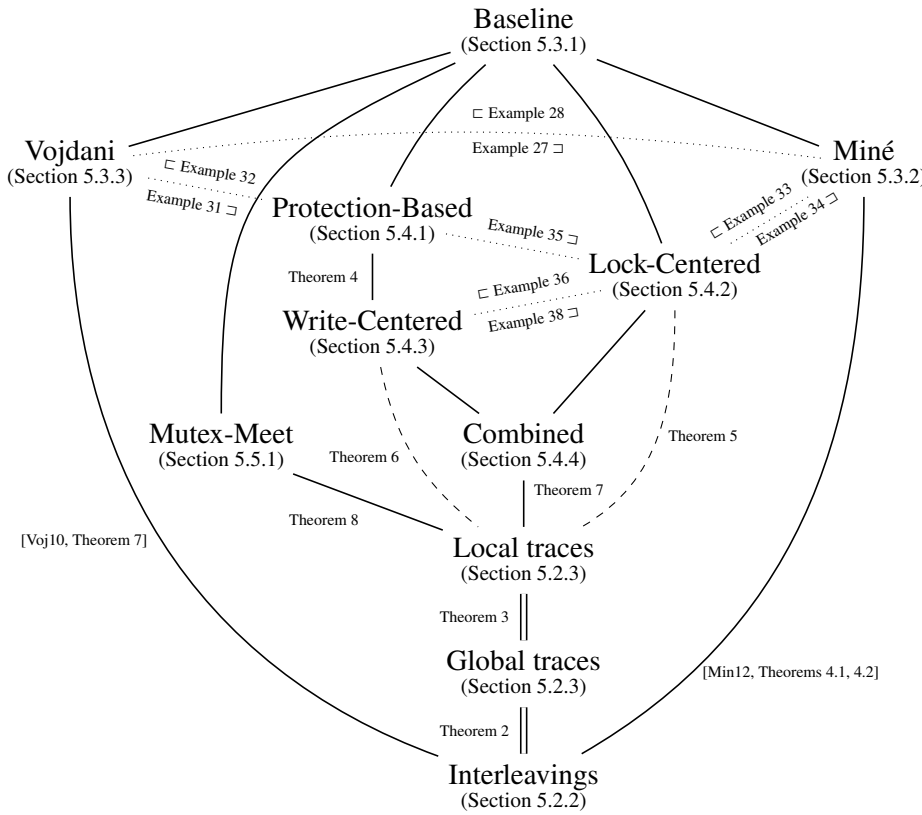


Figure 15. Lattice of thread-modular semantics and analyses as a Hasse diagram

globals, which could be empty. However, $\mathcal{G}_{\mathrm{w}}[m_g] = \{g\}$ for each $g \in \mathcal{G}$ due to the properties above.

Second, the analysis performs lazy reading by delaying the importing of global invariants at locking until the reading of a global variable. Moreover, it stores in the local state mappings σ only those globals that the ego thread has written. To achieve the essence of privatization (Example 26), it also maintains in local states a set P of definitely written globals g that are still write-protected by some mutex, i.e., the ego thread holds some write-protecting mutex of g. When reading those globals, the variable is privatized and it suffices to only read their values from the local state, without incorporating any global invariants.

Due to the weaker form of protection, the program may read a global g without holding any write-protecting program mutexes (i.e., other than m_g). Therefore, the analysis uses two kinds of unknowns for global invariants:

1. *Protected unknowns* $[g]$ for abstracting values that were *last* written before releasing a write-protecting program mutex of g (i.e., other than m_g). These are consulted when reading g while holding a write-protecting program mutex.

2. *Unprotected unknowns* $[g]'$ for abstracting all values ever written to g. These are consulted when reading g without holding a write-protecting program mutex.

Such a situation and the difference between the global invariants is illustrated by the following example.

Example 29. Consider the following program, where $\mathcal{M}_{\mathrm{rw}}[\mathtt{g}] = \{m_{\mathtt{g}}\}$ but $\mathcal{M}_{\mathrm{w}}[\mathtt{g}] = \{\mathtt{m}, m_{\mathtt{g}}\}$:

```
1  int g = 0;
2  mutex m;

3  void main() {
4    create(t);
5    int x = g;
6    lock(m);
7      int y = g;
8    unlock(m);
9  }

10 void t() {
11   lock(m);
12     g = 1;
13     g = 0;
14   unlock(m);
15 }
```

For this program, the protected invariant [g] should be $\{0\}$, while the unprotected invariant [g]′ should be $\{0, 1\}$. Since the `main` thread first reads the global g without locking `m`, the local `x` can have values $\{0, 1\}$ (from [g]′). It then reads the global g while holding `m`, so the local `y` can have values $\{0\}$ (from [g]).

Due to the presence of an unprotected read, Vojdani's analysis is also less precise for the write-protected read. □

We give the right-hand sides for this analysis, starting with the initial state:

$$\begin{aligned}\mathsf{init}^\sharp \stackrel{\text{def}}{=} {} & \textbf{let } \sigma = \{x \mapsto \top \mid x \in \mathcal{X}\} \cup \{g \mapsto \bot \mid g \in \mathcal{G}\} \textbf{ in} \\ & \textbf{let } \rho = \{[g] \mapsto 0^\sharp \mid g \in \mathcal{G}\} \cup \{[g]' \mapsto 0^\sharp \mid g \in \mathcal{G}\} \textbf{ in} \\ & (\rho, (\varnothing, \sigma)).\end{aligned}$$

Due to lazy reading, locking does not affect the local state:

$$[\![[u, S], \mathsf{lock}(m)]\!]^\sharp \eta \stackrel{\text{def}}{=} (\varnothing, \eta\,[u, S]),$$
$$[\![[u, S], \mathsf{lock}(m_g)]\!]^\sharp \eta \stackrel{\text{def}}{=} (\varnothing, \eta\,[u, S]).$$

Unlocking an atomicity mutex m_g publishes the value of g to the corresponding unprotected unknown.[52] It also removes from P the globals which are no longer protected by any mutex:

$$\begin{aligned}[\![[u, S], \mathsf{unlock}(m_g)]\!]^\sharp \eta \stackrel{\text{def}}{=} {} & \textbf{let } (P, \sigma) = \eta\,[u, S] \textbf{ in} \\ & \textbf{let } P' = \{h \in P \mid (S \setminus \{m_g\}) \cap \mathcal{M}_{\mathrm{w}}[h] \neq \varnothing\} \textbf{ in} \\ & \textbf{let } \rho = \{[g]' \mapsto \sigma\, g\} \textbf{ in} \\ & (\rho, (P', \sigma)).\end{aligned}$$

[52] Schwarz et al. [Sch+21a] also side-effect the value to $[g]$ if $\mathcal{M}_{\mathrm{w}}[g] = \{m_g\}$. This is convenient for proving Theorem 4 below, but unnecessary and unobservable by the analysis.

Unlocking $m \in \mathcal{M} \setminus \mathcal{M}_{\mathcal{G}}$ publishes values of the globals it protects to the corresponding protected unknowns and updates P:

$$\begin{aligned}[[\,[u, S], \mathsf{unlock}(m)\,]]^\sharp\, \eta \stackrel{\text{def}}{=} & \textbf{ let } (P, \sigma) = \eta\,[u, S] \textbf{ in} \\ & \textbf{let } P' = \{g \in P \mid (S \setminus \{m\}) \cap \mathcal{M}_{\mathrm{w}}[g] \neq \varnothing\} \textbf{ in} \\ & \textbf{let } \rho = \{[g] \mapsto \sigma\, g \mid g \in \mathcal{G}_{\mathrm{w}}[m]\} \textbf{ in} \\ & (\rho, (P', \sigma)).\end{aligned}$$

Unlike Vojdani's analysis, values are published to $[g]$ at every unlock of *any* protecting program mutex of g, not when all of them have been unlocked (and g removed from P). That is because the variable is no longer *fully* protected and another thread could already lock one of the unlocked protecting program mutexes, observing its value. However, $\mathsf{unlock}(m)$ does not need to check this: m is a protecting mutex and it is no longer held.

Writing to global g is performed thread-locally, since all of its write-protecting mutexes $\mathcal{M}_{\mathrm{w}}[g]$ are held by definition:

$$\begin{aligned}[[\,[u, S], g \leftarrow x\,]]^\sharp\, \eta \stackrel{\text{def}}{=} & \textbf{ let } (P, \sigma) = \eta\,[u, S] \textbf{ in} \\ & (\varnothing, (P \cup \{g\}, \sigma[g \mapsto \sigma\, x])).\end{aligned}$$

Reading from global g distinguishes three cases:

1. g was definitely written by the ego thread and is still write-protected, in which case only the thread's own write is observed.
2. g is write-protected by a program mutex (but was not definitely written by the ego thread), in which case the protected invariant $[g]$ can be also observed.
3. g is not write-protected by a program mutex, in which case the unprotected invariant $[g]'$ can be observed.[53]

Formally:

$$\begin{aligned}[[\,[u, S], x \leftarrow g\,]]^\sharp\, \eta \stackrel{\text{def}}{=} & \textbf{ let } (P, \sigma) = \eta\,[u, S] \textbf{ in} \\ & \textbf{if } g \in P \textbf{ then} \\ & \quad (\varnothing, (P, \sigma[x \mapsto \sigma\, g])) \\ & \textbf{else if } S \cap \mathcal{M}_{\mathrm{w}}[g] \neq \{m_g\} \textbf{ then} \\ & \quad (\varnothing, (P, \sigma[x \mapsto \sigma\, g \sqcup \eta\,[g]])) \\ & \textbf{else} \\ & \quad (\varnothing, (P, \sigma[x \mapsto \eta\,[g]'])).\end{aligned}$$

For guards and computations on locals, the right-hand sides are defined in the intuitive manner — they operate on σ only, leaving P unchanged.

[53] Schwarz et al. [Sch+21a] also observe and join with the thread's own writes from $\sigma\, g$. This makes the right-hand side monotonic, but is unnecessary and unobservable by the analysis, because $[g]'$ includes those writes.

Example 30. Consider analyzing the program from Example 26 (p. 138) using this analysis. The write-protecting mutexes for g are also $\mathcal{M}_{\mathrm{w}}[\mathtt{g}] = \{\mathtt{m}, m_{\mathtt{g}}\}$. Initialization side-effects $\{0\}$ to $[\mathtt{g}]$ and $[\mathtt{g}]'$, and returns the initial state

$$(\varnothing,\ \{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{g} \mapsto \bot\}).$$

Locking m in the thread t has no effect. The subsequent assignment to g (together with the implicit $\mathsf{lock}(m_{\mathtt{g}})$ and $\mathsf{unlock}(m_{\mathtt{g}})$) updates the local state to

$$(\{\mathtt{g}\},\ \{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{g} \mapsto \{1\}\})$$

and side-effects $\{1\}$ to $[\mathtt{g}]'$, but not to $[\mathtt{g}]$. Thus, $[\mathtt{g}]'$ now has the value $\{0, 1\}$. The following read of g into y acts thread-locally since $\mathtt{g} \in P$, yielding

$$(\{\mathtt{g}\},\ \{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \{1\}, \mathtt{g} \mapsto \{1\}\}).$$

Thus, the assertion in thread t is proven. The final assignment to g updates the local state to

$$(\{\mathtt{g}\},\ \{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \{1\}, \mathtt{g} \mapsto \{0\}\})$$

and side-effects $\{0\}$ to $[\mathtt{g}]'$, but not to $[\mathtt{g}]$. This side-effected value was already included in $[\mathtt{g}]'$, so it has no additional consequences. Unlocking m in the thread t removes g from P, yielding

$$(\varnothing,\ \{\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \{1\}, \mathtt{g} \mapsto \{0\}\})$$

and side-effects $\{0\}$ to $[\mathtt{g}]$ since the global becomes unprotected. This side-effected value was already included in $[\mathtt{g}]$, so it has no additional consequences.

Meanwhile, locking m in the main thread has no effect. The following read of g into x gets the value from $[\mathtt{g}]$ since the global is protected by a program mutex, yielding

$$(\varnothing,\ \{\mathtt{x} \mapsto \{0\}, \mathtt{y} \mapsto \top, \mathtt{g} \mapsto \bot\}).$$

Thus, the assertion in the main thread is proven. Unlocking m in the main thread does not change the local state, but side-effects $\bot$ to $[\mathtt{g}]$ since the global becomes unprotected, but this was already trivially included. Therefore, in the end, $[\mathtt{g}]$ is $\{0\}$ and $[\mathtt{g}]'$ is $\{0, 1\}$. □

Theorem 4. *Protection-Based Reading is sound w.r.t. the local trace semantics.*

Proof. The extended version of the original paper [Sch+21b, Section 5.3] proves that this analysis is an abstraction of Write-Centered Reading (Section 5.4.3), which sound by Theorem 6. ■

Remark 16 (Minimizing local states). This analysis never removes any values from σ, even if a variable becomes unprotected and its value is side-effected. However, in practice, it is desirable to do so in order to keep local state representations as small as possible. This can be achieved by maintaining in local states a set W of

globals (domain ordered by subset) which may have been written while protected and are still write-protected. The transfer functions handle W analogously to P — they only differ in their domain ordering. The value of global g can be removed from σ if $g \notin P \cup W$.[54] This is sound because in that case $\sigma\, g$ has already been side-effected, and is included in $[g]$ and $[g]'$, which are to be consulted at the following read of g.

This optimization is not just for reducing memory usage but can also speed up fixpoint iteration. For example, this happens when a loop body acquires a mutex, writes a protected variable and releases the mutex, as will be the case in Example 45 in Section 6.2. □

Remark 17 (Analysis of write-protecting mutexes). Above we assumed the sets $\mathcal{M}_{\mathrm{w}}[g]$ as given for simplicity of presentation. Analogously to Remark 15 (p. 139), these can be computed by pre-analysis or jointly with the analysis itself. However, for write-protecting mutexes, the additional side effects (highlighted) are only to be performed when writing, not reading:

$$[\![[u, S], g \leftarrow x]\!]^\sharp\, \eta \stackrel{\text{def}}{=} \textbf{let } (P, \sigma) = \eta\, [u, S] \textbf{ in } (\{\mathcal{M}_{\mathrm{w}}[g] \mapsto S\}, (P \cup \{g\}, \sigma[g \mapsto \sigma\, x])).$$

Similarly to Vojdani's analysis, the right-hand side for $\mathsf{unlock}(m)$ is non-monotonic in $\mathcal{M}_{\mathrm{w}}[g]$ ($g \in \mathcal{G}$) due to the check $m \in \mathcal{M}_{\mathrm{w}}[g]$ in the definition of $g \in \mathcal{G}_{\mathrm{w}}[m]$. While the analogous check in Vojdani's analysis was just for efficiency, it is necessary for soundness of Protection-Based Reading. Thus, the non-monotonicity is unavoidable. Furthermore, it can have an effect on precision, as seen in Example 32 below. □

Example 31 (Protection-Based ⊏ Vojdani)**.** Consider the following program, where $\mathcal{M}_{\mathrm{rw}}[\mathtt{g}] = \{\mathtt{m2}, m_{\mathrm{g}}\}$ but $\mathcal{M}_{\mathrm{w}}[\mathtt{g}] = \{\mathtt{m1}, \mathtt{m2}, m_{\mathrm{g}}\}$:

```
1  int g = 0;
2  mutex m1, m2;

3  void main() {
4    create(t);
5    lock(m2);
6      int x = g;
7    unlock(m2);
8  }

9  void t() {
10   lock(m1);
11     lock(m2);
12       g = 5;
13     unlock(m2);
14     lock(m2);
15       assert(g == 5);
16       g = 0;
17     unlock(m2);
18   unlock(m1);
19 }
```

[54] At this point, the check could be simplified to $g \notin W$ because $P \subseteq W$. However, later Remark 22 (p. 167) will no longer guarantee this.

According to Vojdani's analysis, the global becomes unprotected in `t` when `m2` is temporarily unlocked. Thus, the value $\{5\}$ gets published at unlock, but $\{0, 5\}$ is joined at lock, accounting for initialization. Therefore, it cannot prove the assertion.

According to Protection-Based Reading, the global does not become write-unprotected at the same point. Hence, `g` remains in the set P, and thread `t` can read its own written value from local state, proving the assertion. □

Example 32 (Vojdani ⊏ Protection-Based)**.** Consider the following program, where $\mathcal{M}_{\text{rw}}[\texttt{g}] = \{\texttt{m1}, m_{\texttt{g}}\}$ but $\mathcal{M}_{\text{w}}[\texttt{g}] = \{\texttt{m1}, \texttt{m2}, m_{\texttt{g}}\}$:

```
1  int g = 0;
2  mutex m1, m2;

3  void main() {
4    create(t);
5    lock(m1);
6      assert(g == 0);
7    unlock(m1);
8  }

9  void t() {
10   lock(m1);
11     lock(m2);
12       g = 5;
13     unlock(m2);
14     lock(m2);
15       g = 0;
16     unlock(m2);
17   unlock(m1);
18 }
```

Here `m1` is meant to protect `g`, but `m2` just happens to also be held at every one of its writes, perhaps because it protects some other variable (not shown here). Turns out, having more precise protection information (i.e., more protecting mutexes) can actually make the analysis less precise. This is an example of the non-monotonicity mentioned in Remark 17.

According to Protection-Based Reading, $\{5\}$ is published to `[g]` when `m2` is temporarily unlocked in `t`, but `g` remains write-protected. Nevertheless, the `main` thread reads $\{0, 5\}$ after locking `m1`, accounting for initialization. Thus, it cannot prove the assertion.

According to Vojdani's analysis, the global does not become unprotected in `t` during the temporary unlocking of `m2`. Therefore, nothing is published there and the final unlock of `m1` in `t` publishes just $\{0\}$, which matches the initialization.[55] Hence, when the `main` thread locks `m1`, it joins $\{0\}$ into the local state and can prove the assertion. □

In order to remain sound, Protection-Based Reading must publish a write-protected variable when unlocking *any* of its write-protecting mutexes, even if it remains write-protected (i.e., in P). That is because some other thread may then read it without writing, like in Example 32. Intuitively, Vojdani's analysis avoids

[55] With the less precise (yet still sound) $\mathcal{M}_{\text{w}}[\texttt{g}] = \{\texttt{m1}, m_{\texttt{g}}\}$, Protection-Based Reading would behave similarly and be as precise.

such publishing because it is also aware of reads in other threads via $\mathcal{M}_{\mathrm{rw}}[g]$, and does not publish when no other thread could possibly read it.

Remark 18. This incomparability can be fixed (without proof) by adding the side condition based on $\mathcal{M}_{\mathrm{rw}}[g]$ from Vojdani's analysis to unlocking $m \in \mathcal{M} \setminus \mathcal{M}_{\mathcal{G}}$ of Protection-Based Reading (highlighted in ρ):

$$
\begin{aligned}
[\![[u, S], \mathsf{unlock}(m)]\!]^\sharp\, \eta \stackrel{\text{def}}{=}\ & \textbf{let } (P, \sigma) = \eta\,[u, S] \textbf{ in} \\
& \textbf{let } P' = \{g \in P \mid (S \setminus \{m\}) \cap \mathcal{M}_{\mathrm{w}}[g] \neq \varnothing\} \textbf{ in} \\
& \textbf{let } \rho = \\
& \quad \{[g] \mapsto \sigma\, g \mid g \in \mathcal{G}_{\mathrm{w}}[m], \\
& \qquad\qquad (S \setminus \{m\}) \cap \mathcal{M}_{\mathrm{rw}}[g] = \varnothing\} \\
& \textbf{in} \\
& (\rho, (P', \sigma)).
\end{aligned}
$$

Adding the simplified side condition $\mathcal{M}_{\mathrm{rw}}[g] = \{m_g\}$ to $\mathsf{unlock}(m_g)$ ($g \in \mathcal{G}$) does not provide additional precision: it avoids a side effect to $[g]'$ when all accesses of g are protected by a program mutex (i.e., $\{m_g\} \subset \mathcal{M}_{\mathrm{rw}}[g]$), in which case the transfer function for reading g would never enter the case that incorporates $[g]'$ anyway. □

5.4.2. Lock-Centered Reading

We enhance Miné's analysis (Section 5.3.2) with lazy reading by delaying the importing of synchronized interferences at locking until reading of a global variable. When reading a global g we want to join in (a sound subset of) unknowns $[g, m, S']$ which would have been joined in at preceding $\mathsf{lock}(m)$ operations by Miné's analysis. To be able to account for background locksets, the analysis needs to remember in local states the background locksets at encountered $\mathsf{lock}(m)$ operations. In fact, it suffices for each mutex m to only account for values imported at the *last* $\mathsf{lock}(m)$ before reading the global.[56] Note that these may have been different in different paths reaching the current node and lockset.

Let $\mathcal{U}_{\mathcal{M}} \subseteq 2^{2^{\mathcal{M}}}$ denote the set of all upward closed subsets of $2^{\mathcal{M}}$. A subset $U \subseteq 2^{\mathcal{M}}$ is *upward closed* if for every $S \in U$ all of its supersets are also included in U, i.e., $S' \in U$ for every $S' \supseteq S$. Each non-empty value in $\mathcal{U}_{\mathcal{M}}$ can be represented by the set of its minimal elements, because non-minimal elements are implicitly included by upward closure anyway. Furthermore, $\mathcal{U}_{\mathcal{M}}$ forms an abstract domain which is ordered by subset inclusion. The *least* element of this is $\varnothing$, while the *greatest* element is given by the *full* power set of mutexes (represented by $\{\varnothing\}$).

We now maintain a map $L : \mathcal{M} \to \mathcal{U}_{\mathcal{M}}$ in the local state that tracks for each mutex m all minimal background locksets that were held when m was last acquired.

[56] The eager reading of Miné's analysis also accounts for the non-last $\mathsf{lock}(m)$ because it has irreversibly joined them into the local state before the last one.

The reason that minimal background locksets suffice will become apparent below. This abstraction of acquisition histories [KIG05; Kah+07] allows us to achieve lazy reading. Moreover, σ in the local state only maintains values that the ego thread has written itself.

In case the ego thread has definitely written to a variable and no additional locks have occurred since, only the local copy needs to be read. To achieve that, we introduce an additional map $V : \mathcal{M} \to 2^{\mathcal{G}}$. For mutex m, $V\,m$ is the set of global variables (domain ordered by superset) that were definitely written since m was last acquired. In case that m has never been acquired by the ego thread, we set $V\,m$ to the set of all global variables that have definitely been written since the start of the thread.

We start by giving the right-hand side for the initial state:

$$\begin{aligned}
\mathsf{init}^\sharp \stackrel{\text{def}}{=} \ & \textbf{let } V = \{m \mapsto \varnothing \mid m \in \mathcal{M}\} \textbf{ in} \\
& \textbf{let } L = \{m \mapsto \varnothing \mid m \in \mathcal{M}\} \textbf{ in} \\
& \textbf{let } \sigma = \{x \mapsto \top \mid x \in \mathcal{X}\} \cup \{g \mapsto \bot \mid g \in \mathcal{G}\} \textbf{ in} \\
& \textbf{let } \rho = \{[g, m_g, \varnothing] \mapsto 0^\sharp \mid g \in \mathcal{G}\} \textbf{ in} \\
& (\rho, (V, L, \sigma)).
\end{aligned}$$

For guards and computations on locals, the right-hand sides are once more defined in the obvious way — they access σ only, leaving V and L unchanged.

Locking a mutex m resets $V\,m$ to $\varnothing$ and updates L, whereas unlocking side-effects the value of globals to the appropriate unknowns.

$$\begin{aligned}
[\![[u, S], \mathsf{lock}(m)]\!]^\sharp\,\eta \stackrel{\text{def}}{=} \ & \textbf{let } (V, L, \sigma) = \eta\,[u, S] \textbf{ in} \\
& \textbf{let } V' = V[m \mapsto \varnothing] \textbf{ in} \\
& \textbf{let } L' = L[m \mapsto \{S\}] \textbf{ in} \\
& (\varnothing, (V', L', \sigma)), \\
[\![[u, S], \mathsf{unlock}(m)]\!]^\sharp\,\eta \stackrel{\text{def}}{=} \ & \textbf{let } (V, L, \sigma) = \eta\,[u, S] \textbf{ in} \\
& \textbf{let } \rho = \{[g, m, S \setminus \{m\}] \mapsto \sigma\,g \mid g \in \mathcal{G}\} \textbf{ in} \\
& (\rho, (V, L, \sigma)).
\end{aligned}$$

The right-hand side for writing to a global g then consists of a strong update to the local copy and addition of g to $V\,m$ for all mutexes m:

$$\begin{aligned}
[\![[u, S], g \leftarrow x]\!]^\sharp\,\eta \stackrel{\text{def}}{=} \ & \textbf{let } (V, L, \sigma) = \eta\,[u, S] \textbf{ in} \\
& \textbf{let } V' = \{m \mapsto V\,m \cup \{g\} \mid m \in \mathcal{M}\} \textbf{ in} \\
& (\varnothing, (V', L, \sigma[g \mapsto \sigma\,x])).
\end{aligned}$$

For reading from a global g, those values $[g, m, S']$ need to be taken into account where m is one of the mutexes acquired in the past and the intersection of some set

in $L\,m$ with the set of mutexes S' held while publishing is empty:

$$\begin{aligned}[[\,[u,S],x \leftarrow g\,]]^\sharp\,\eta \stackrel{\text{def}}{=}\ & \textbf{let } (V,L,\sigma) = \eta\,[u,S] \textbf{ in}\\ & \textbf{let } d =\\ & \quad \bigsqcup \{\eta\,[g,m,S'] \mid m \in \mathcal{M}, S' \subseteq \mathcal{M},\\ & \qquad\qquad\qquad\quad g \notin V\,m,\\ & \qquad\qquad\qquad\quad \exists B \in L\,m : B \cap S' = \varnothing\}\\ & \textbf{in}\\ & (\varnothing, (V, L, \sigma[x \mapsto \sigma\,g \sqcup d])).\end{aligned}$$

In case that $L\,m = \varnothing$, i.e., if according to the analysis no thread reaching u with lockset S has ever locked mutex m, then no values from $[g,m,S']$ will be read.

Example 33 (Lock-Centered ⊏ Miné)**.** Consider the following program:

```
1  int g = 0;
2  mutex m1, m2;

3  void main() {
4    create(t);
5    lock(m2);
6    unlock(m2);
7    lock(m1);
8      lock(m2);
9        int x = g;
10       assert(x == 0);
11       // ...
12 }
```

```
13 void t() {
14   lock(m1);
15     lock(m2);
16       g = 1;
17     unlock(m2);
18     g = 0;
19   unlock(m1);
20 }
```

Lock-Centered Reading produces five side effects on this program:

$$\begin{aligned}[\mathtt{g}, m_\mathtt{g}, \varnothing] &\mapsto \{0\} && \text{(initialization)},\\ [\mathtt{g}, m_\mathtt{g}, \{\mathtt{m1},\mathtt{m2}\}] &\mapsto \{1\} && (\mathtt{g} \leftarrow 1 \text{ in } \mathtt{t}),\\ [\mathtt{g}, \mathtt{m2}, \{\mathtt{m1}\}] &\mapsto \{1\} && (\mathsf{unlock}(\mathtt{m2}) \text{ in } \mathtt{t}),\\ [\mathtt{g}, m_\mathtt{g}, \{\mathtt{m1}\}] &\mapsto \{0\} && (\mathtt{g} \leftarrow 0 \text{ in } \mathtt{t}),\\ [\mathtt{g}, \mathtt{m1}, \varnothing] &\mapsto \{0\} && (\mathsf{unlock}(\mathtt{m1}) \text{ in } \mathtt{t}).\end{aligned}$$

When reading g into x, Lock-Centered Reading has the local state

$$\begin{aligned}&(\{\mathtt{m1} \mapsto \varnothing, \mathtt{m2} \mapsto \varnothing, m_\mathtt{g} \mapsto \varnothing\},\\ &\{\mathtt{m1} \mapsto \{\varnothing\}, \mathtt{m2} \mapsto \{\{\mathtt{m1}\}\}, m_\mathtt{g} \mapsto \{\{\mathtt{m1},\mathtt{m2}\}\}\},\\ &\{\mathtt{x} \mapsto \top, \mathtt{g} \mapsto \bot\}).\end{aligned}$$

Thus, it only reads $\{0\}$ (either from initialization or $\mathsf{unlock}(\mathtt{m1})$ in t) and can prove the assertion.

At unlock(m2) in t, Miné's analysis produces the side effect $[\mathtt{g}, \mathtt{m2}, \{\mathtt{m1}\}] \mapsto \{1\}$. At the first lock(m2) in the main thread, the analysis eagerly joins this 1 into the local state for g, because the background locksets $\varnothing$ (at this point) and $\{\mathtt{m1}\}$ (in the side effect) are disjoint. Thus, it reads $\{0, 1\}$ before the assertion, which it cannot prove. □

Example 34 (Miné ⊏ Lock-Centered)**.** Consider the following program:

```
1 int g = 0;

2 void main() {
3   g = 1;
4   int x = g;
5   assert(x == 1);
6 }
```

Lock-Centered Reading produces two side effects on this program:

$$[\mathtt{g}, m_{\mathtt{g}}, \varnothing] \mapsto \{0\} \qquad \text{(initialization)},$$
$$[\mathtt{g}, m_{\mathtt{g}}, \varnothing] \mapsto \{1\} \qquad (\mathtt{g} \leftarrow 1).$$

When reading g into x, Lock-Centered Reading has the local state

$$(\{m_{\mathtt{g}} \mapsto \varnothing\},\ \{m_{\mathtt{g}} \mapsto \{\varnothing\}\},\ \{\mathtt{x} \mapsto \top, \mathtt{g} \mapsto \{1\}\}).$$

Thus, it reads $\{0, 1\}$ (including the initial side effect) and cannot prove the assertion.

Miné's analysis does no such initial side effect and overwrites the value of g in the local state when writing. Thus, it reads $\{1\}$ before the assertion, which it can then prove. □

Remark 19 (Not reading initial values). As the previous example also shows, the side effects from initialization are actually such that *any* read of a global joins it in. In practice, this imprecision is inconvenient, but also avoidable with a small modification. In initialization, instead of side-effecting $0^\sharp$ to $[g, m_g, \varnothing]$ for each $g \in \mathcal{G}$, the side effect would go to $[g, m_g, \mathcal{M}]$. The existing right-hand side for reading globals would never include this due to the full background lockset (which cannot arise from the program itself). To be sound, that right-hand side must additionally join in $[g, m_g, \mathcal{M}]$ whenever reading g, but only when $\nexists m \in \mathcal{M} : g \in V\, m$. This excludes the initial value when g is known to have been overwritten while still holding some mutex m. □

Example 35 (Lock-Centered ⊏ Protection-Based)**.** Consider the following program:

```
1  int g = 0;
2  mutex m1, m2;

3  void main() {
4    create(t1);
5    create(t2);
6    lock(m2);
7      g = 31;
8      lock(m1);
9        int x = g;
10       // ...
11 }

12 void t1() {
13   lock(m1);
14     g = 17;
15   unlock(m1);
16 }

17 void t2() {
18   lock(m2);
19     g = 59;
20   unlock(m2);
21 }
```

Protection-Based Reading cannot exclude any values of `x` as $\mathcal{M}_{\mathrm{w}}[\mathtt{g}] = \{m_{\mathtt{g}}\}$ and, thus, has $\{0, 17, 31, 59\}$ for `x`.

Lock-Centered Reading produces two side effects in `t2`:

$$[\mathtt{g}, m_{\mathtt{g}}, \{\mathtt{m2}\}] \mapsto \{59\} \qquad (\mathtt{g} \leftarrow 59)$$
$$[\mathtt{g}, \mathtt{m2}, \varnothing] \mapsto \{59\} \qquad (\mathsf{unlock}(\mathtt{m2}))$$

Lock-Centered Reading has $V\,\mathtt{m2} = \{\mathtt{g}\}$ and $L\,m_{\mathtt{g}} = \{\{\mathtt{m1}, \mathtt{m2}\}\}$ at the read. These exclude both side effects from the write by `t2` and, thus, results in $\{0, 17, 31\}$ for `x`. □

Theorem 5. *Lock-Centered Reading is sound w.r.t. the local trace semantics.*

Proof. Deferred to the extended version of the original paper [Sch+21b, Section 5.1]. ■

5.4.3. Write-Centered Reading

Recall that Protection-Based Reading (Section 5.4.1) assumes that each global g is write-protected by some set of mutexes $\mathcal{M}_{\mathrm{w}}[g]$. We now propose a refinement of that analysis which lifts this assumption about program-wide protecting mutexes. The new analysis uses two auxiliary data structures in local states to achieve that:

1. A map $W : \mathcal{G} \to \mathcal{U}_{\mathcal{M}}$ tracks for each global g the set of (minimal) locksets held when g was last written by the ego thread. At the beginning of a thread, it has not yet written to any globals; hence, $W\,g = \varnothing$ for all globals g.
2. A map $P : \mathcal{G} \to \mathcal{U}_{\mathcal{M}}$ tracks for each global g all (minimal) locksets the ego thread has held since its last write to g. A global g which the ego thread has not yet written to is mapped to the *full* power set of mutexes (represented by $\{\varnothing\}$). This is a refinement of the set P from Protection-Based Reading.

The unknowns for global invariants in this analysis are refinements of those from Lock-Centered Reading: they are of the form $[g, m, S', w]$ where $g \in \mathcal{G}$, $m \in \mathcal{M}$, $S' \subseteq \mathcal{M}$ is the background lockset at $\mathsf{unlock}(m)$, and $w \subseteq \mathcal{M}$ is a minimal lockset when g was last written before $\mathsf{unlock}(m)$. We start by giving the right-hand side for the initial state:

$$\begin{aligned}
\mathsf{init}^\sharp \stackrel{\text{def}}{=} & \textbf{ let } W = \{g \mapsto \varnothing \mid g \in \mathcal{G}\} \textbf{ in} \\
& \textbf{let } P = \{g \mapsto \{\varnothing\} \mid g \in \mathcal{G}\} \textbf{ in} \\
& \textbf{let } \sigma = \{x \mapsto \top \mid x \in \mathcal{X}\} \cup \{g \mapsto \bot \mid g \in \mathcal{G}\} \textbf{ in} \\
& \textbf{let } \rho = \{[g, m_g, \varnothing, \mathcal{M}] \mapsto 0^\sharp \mid g \in \mathcal{G}\} \textbf{ in} \\
& (\rho, (W, P, \sigma)).
\end{aligned}$$

For guards and computations on locals, the right-hand sides are once more defined intuitively — they operate on σ only, leaving W and P unchanged.

While nothing happens at locking, unlocking now updates the data structure P and additionally side-effects the current local values for each global g to the corresponding unknowns.

$$\begin{aligned}
[\![[u, S], \mathsf{lock}(m)]\!]^\sharp \eta &\stackrel{\text{def}}{=} (\varnothing, \eta\,[u, S]), \\
[\![[u, S], \mathsf{unlock}(m)]\!]^\sharp \eta &\stackrel{\text{def}}{=} \textbf{let } (W, P, \sigma) = \eta\,[u, S] \textbf{ in} \\
&\qquad \textbf{let } P' = \{g \mapsto P\,g \sqcup \{S \setminus \{m\}\} \mid g \in \mathcal{G}\} \textbf{ in} \\
&\qquad \textbf{let } \rho = \\
&\qquad\quad \{[g, m, S \setminus \{m\}, w] \mapsto \sigma\,g \mid g \in \mathcal{G}, w \in W\,g\} \\
&\qquad \textbf{in} \\
&\qquad (\rho, (W, P', \sigma)).
\end{aligned}$$

When writing to a global g, on top of recording the written value in σ, $W\,g$ and $P\,g$ are set to the set $\{S\}$ for the current lockset S:

$$\begin{aligned}
[\![[u, S], g \leftarrow x]\!]^\sharp \eta \stackrel{\text{def}}{=} & \textbf{let } (W, P, \sigma) = \eta\,[u, S] \textbf{ in} \\
& \textbf{let } W' = W[g \mapsto \{S\}] \textbf{ in} \\
& \textbf{let } P' = P[g \mapsto \{S\}] \textbf{ in} \\
& (\varnothing, (W', P', \sigma[g \mapsto \sigma\,x])).
\end{aligned}$$

When reading from a global g, now only values stored at $\eta\,[g, m, S', w]$ are taken into account, provided:

1. $m \in S$, i.e., m is one of the currently held mutexes.
2. $S \cap S' = \varnothing$, i.e., the intersection of the current lockset S with the background lockset at the corresponding operation $\mathsf{unlock}(m)$ after the write producing the value stored at this unknown is empty.

3. $w \cap S'' = \varnothing$ for some $S'' \in P\,g$, i.e., the background lockset at the write producing the value stored at this unknown is disjoint with one of the locksets in $P\,g$. This excludes writes where the ego thread has since its last thread-local write always held at least one of the mutexes in w. In this case, that write can not have happened between the last thread-local write of the reading ego thread and its read.
4. $m \notin S'''$ for some $S''' \in P\,g$, i.e., m has not been continuously held by the thread since its last write to g.

Accordingly, we define

$$\begin{aligned}[[\,[u,S], x \leftarrow g\,]]^\sharp\,\eta \stackrel{\text{def}}{=} \textbf{let } (W,P,\sigma) = \eta\,[u,S] \textbf{ in}\\ \textbf{let } d =\\ \bigsqcup \{\eta\,[g,m,S',w] \mid\ & m \in S, S' \subseteq \mathcal{M}, w \subseteq \mathcal{M},\\ & S \cap S' = \varnothing,\\ & \exists S'' \in P\,g : S'' \cap w = \varnothing,\\ & \exists S''' \in P\,g : m \notin S'''\}\\ \textbf{in}\\ (\varnothing, (W, P, \sigma[x \mapsto \sigma\,g \sqcup d])).\end{aligned}$$

Example 36 (Write-Centered $\sqsubset$ Lock-Centered)**.** Consider the program from Example 35. At the read `x = g`, the current lockset is $\{\texttt{m1}, \texttt{m2}, m_{\texttt{g}}\}$ and in the local state $P\,\texttt{g} = \{\{\texttt{m2}\}\}$. All of the above conditions for reading are fulfilled only for the unknown $[\texttt{g}, \texttt{m1}, \varnothing, \{\texttt{m1}, m_{\texttt{g}}\}]$, which has value $\{17\}$. Hence, this is the only value read from the unknowns for g, and together with the value $\{31\}$ from $\sigma\,\texttt{g}$, the final value for `x` is $\{17, 31\}$.

From Example 35, we know that Lock-Centered Reading has $\{0, 17, 31\}$ for `x`. Hence, Write-Centered Reading is more precise. □

Example 37 (Write-Centered $\sqsubset$ Protection-Based)**.** Since Examples 35 and 36 are on the same program, it also follows that Protection-Based Reading, with $\{0, 17, 31, 59\}$ for `x`, is less precise than Write-Centered Reading. □

Theorem 6. *Write-Centered Reading is sound w.r.t. the local trace semantics.*

Proof. Deferred to the extended version of the original paper [Sch+21b, Section 5.2]. ■

5.4.4. Combined Reading

Despite numerous precision improvements, Lock-Centered Reading and Write-Centered Reading still turn out to be incomparable. There are cases where either is strictly more precise than the other, as evidenced by Example 36 and the following example.

Example 38 (Lock-Centered ⊏ Write-Centered)**.** Consider the following program:

```
1  int g = 0;
2  mutex m1, m2;

3  void main() {
4    create(t);
5    lock(m2);
6      lock(m1);
7      unlock(m2); // !
8      int x = g;
9      // ...
10 }

11 void t() {
12   lock(m2);
13     lock(m1);
14       g = 42;
15     unlock(m1);
16     g = 17;
17   unlock(m2);
18 }
```

For Write-Centered Reading, the side effects include:

$$\begin{aligned}
[\mathtt{g}, m_{\mathtt{g}}, \varnothing, \{\mathtt{m1}, \mathtt{m2}, m_{\mathtt{g}}\}] &\mapsto \{0\} && \text{(initialization)},\\
[\mathtt{g}, \mathtt{m1}, \{\mathtt{m2}\}, \{\mathtt{m1}, \mathtt{m2}, m_{\mathtt{g}}\}] &\mapsto \{42\} && (\mathsf{unlock}(\mathtt{m1}) \text{ in } \mathtt{t}),\\
[\mathtt{g}, m_{\mathtt{g}}, \{\mathtt{m2}\}, \{\mathtt{m2}, m_{\mathtt{g}}\}] &\mapsto \{17\} && (\mathtt{g} \leftarrow 17 \text{ in } \mathtt{t}).
\end{aligned}$$

When reading g in `main`, it has $P\,\mathtt{g} = \{\varnothing\}$, thus, values from all three side effects are joined into the local state, resulting in the value $\{0, 17, 42\}$ for x.

For Lock-Centered Reading, the side effects involving the value 42 are:

$$\begin{aligned}
[\mathtt{g}, m_{\mathtt{g}}, \{\mathtt{m1}, \mathtt{m2}\}] &\mapsto \{42\} && (\mathtt{g} \leftarrow 42 \text{ in } \mathtt{t}),\\
[\mathtt{g}, \mathtt{m1}, \{\mathtt{m2}\}] &\mapsto \{42\} && (\mathsf{unlock}(\mathtt{m1}) \text{ in } \mathtt{t}).
\end{aligned}$$

When reading g in `main`, it has $L\,m_{\mathtt{g}} = \{\{\mathtt{m1}, \mathtt{m2}\}\}$ and $L\,\mathtt{m1} = \{\{\mathtt{m2}\}\}$. Hence, neither side effect fulfils the conditions under which its value is taken into account, resulting in the value $\{0, 17\}$ for x. □

We construct a combined analysis which is more precise than both analyses. The local states of the combined analysis are straightforward products of the two: they track the data structures W, P, V, L together. Most of the right-hand sides are also products/concatenations of the right-hand sides of the individual analyses, updating their data structures simultaneously but independently:

$$\begin{aligned}
\mathsf{init}^\sharp \stackrel{\text{def}}{=}\ & \textbf{let } W = \{g \mapsto \varnothing \mid g \in \mathcal{G}\} \textbf{ in}\\
& \textbf{let } P = \{g \mapsto \{\varnothing\} \mid g \in \mathcal{G}\} \textbf{ in}\\
& \textbf{let } V = \{m \mapsto \varnothing \mid m \in \mathcal{M}\} \textbf{ in}\\
& \textbf{let } L = \{m \mapsto \varnothing \mid m \in \mathcal{M}\} \textbf{ in}\\
& \textbf{let } \sigma = \{x \mapsto \top \mid x \in \mathcal{X}\} \cup \{g \mapsto \bot \mid g \in \mathcal{G}\} \textbf{ in}\\
& \textbf{let } \rho = \{[g, m_g, \varnothing, \mathcal{M}] \mapsto 0^\sharp \mid g \in \mathcal{G}\} \textbf{ in}\\
& (\rho, (W, P, V, L, \sigma)),
\end{aligned}$$

$$
\begin{aligned}
[\![[u,S],\mathsf{lock}(m)]\!]^\sharp\,\eta \stackrel{\text{def}}{=}\ & \textbf{let } (W,P,V,L,\sigma) = \eta\,[u,S] \textbf{ in}\\
& \textbf{let } V' = V[m \mapsto \varnothing] \textbf{ in}\\
& \textbf{let } L' = L[m \mapsto \{S\}] \textbf{ in}\\
& (\varnothing, (W,P,V',L',\sigma)),\\
[\![[u,S],\mathsf{unlock}(m)]\!]^\sharp\,\eta \stackrel{\text{def}}{=}\ & \textbf{let } (W,P,V,L,\sigma) = \eta\,[u,S] \textbf{ in}\\
& \textbf{let } P' = \{g \mapsto P\,g \sqcup \{S \setminus \{m\}\} \mid g \in \mathcal{G}\} \textbf{ in}\\
& \textbf{let } \rho =\\
& \quad \{[g,m,S\setminus\{m\},w] \mapsto \sigma\,g \mid g \in \mathcal{G}, w \in W\,g\}\\
& \textbf{in}\\
& (\rho, (W,P',V,L,\sigma)),\\
[\![[u,S],g \leftarrow x]\!]^\sharp\,\eta \stackrel{\text{def}}{=}\ & \textbf{let } (W,P,V,L,\sigma) = \eta\,[u,S] \textbf{ in}\\
& \textbf{let } W' = W[g \mapsto \{S\}] \textbf{ in}\\
& \textbf{let } P' = P[g \mapsto \{S\}] \textbf{ in}\\
& \textbf{let } V' = \{m \mapsto V\,m \cup \{g\} \mid m \in \mathcal{M}\} \textbf{ in}\\
& (\varnothing, (W',P',V',L,\sigma[g \mapsto \sigma\,x])).
\end{aligned}
$$

Note that the unknowns $[g,m,S',w]$ are those of Write-Centered Reading, which are a refined version of the unknowns of Lock-Centered Reading.

The precision improvement comes at the reading of a global, where the read values of both analyses are intersected by meet ($\sqcap$). Furthermore, the data structure P is not only used to filter the set of unknowns according to Write-Centered Reading, but also the set of unknowns for Lock-Centered Reading (highlighted in d_m):

$$
\begin{aligned}
[\![[u,S],x \leftarrow g]\!]^\sharp\,\eta \stackrel{\text{def}}{=}\ & \textbf{let } (W,P,V,L,\sigma) = \eta\,[u,S] \textbf{ in}\\
& \textbf{let } d_m =\\
& \quad \bigsqcup \{\eta\,[g,m,S',w] \mid m \in \mathcal{M}, S' \subseteq \mathcal{M}, w \subseteq \mathcal{M},\\
& \qquad\qquad g \notin V\,m,\\
& \qquad\qquad \exists B \in L\,m : B \cap S' = \varnothing,\\
& \qquad\qquad \exists S'' \in P\,g : S'' \cap w = \varnothing\}\\
& \textbf{in}\\
& \textbf{let } d_g =\\
& \quad \bigsqcup \{\eta\,[g,m,S',w] \mid m \in S, S' \subseteq \mathcal{M}, w \subseteq \mathcal{M},\\
& \qquad\qquad S \cap S' = \varnothing,\\
& \qquad\qquad \exists S'' \in P\,g : S'' \cap w = \varnothing,\\
& \qquad\qquad \exists S''' \in P\,g : m \notin S'''\}\\
& \textbf{in}\\
& (\varnothing, (W,P,V,L,\sigma[x \mapsto \sigma\,g \sqcup (d_m \sqcap d_g)])).
\end{aligned}
$$

Although the set-builders in d_m and d_g appear very similar, and their results are combined by meet, they cannot be directly combined into a single set-builder with

the conjunction of all conditions. That would be unsound because Lock-Centered and Write-Centered Reading account for the same abstract value via different unknowns $[g, m, S', w]$.

The resulting analysis is by construction more precise than both Lock-Centered Reading and Write-Centered Reading.

Theorem 7. *Combined Reading is sound w.r.t. the local trace semantics.*

Proof. Deferred to the PhD thesis of Schwarz [Sch25]. ∎

5.5. Relational Analysis

So far we have only considered non-relational thread-modular value analyses. However, tracking relationships between program variables is indispensable for proving certain properties of programs or verifying the absence of certain programming errors [CH78; Cou+09; LF08]. Inferring relational properties is particularly challenging for multi-threaded programs. We hypothesize that in such an environment, only relational properties between globals protected by common mutexes are likely to persist throughout program execution.

Based on this hypothesis, we first propose a surprisingly simple relational thread-modular analysis in Section 5.5.1 and then describe numerous improvements to this base analysis in Section 5.5.2. The analyses are generic in the relational domain $\mathbb{R}$ (Section 2.4.3).

5.5.1. Mutex-Meet

We propose a thread-modular analysis which can be fully relational on local variables, but only infers relations between global variables that are jointly protected by some mutex. Similarly to Protection-Based Reading (Section 5.4.1), we assume each global g is *write-protected* by mutexes $\mathcal{M}_{\mathrm{w}}[g] \subseteq \mathcal{M}$ and each mutex m *write-protects* globals $\mathcal{G}_{\mathrm{w}}[m] \subseteq \mathcal{G}$.

The local state of the analysis is an abstract relation between locals and write-protected globals in the domain $\mathbb{R}$. For each mutex m the analysis uses an unknown $[m]$ whose value is also an abstract relation in the same domain but *only* between globals $\mathcal{G}_{\mathrm{w}}[m]$. When m is locked, the abstraction at $[m]$ is incorporated into the local state by *meet* ($\sqcap$), refining the state using additional constraints. When m is unlocked, the relation between globals $\mathcal{G}_{\mathrm{w}}[m]$ is side-effected from the current local state to the unknown $[m]$. Simultaneously, relations involving globals which are no longer protected are *forgotten* from the local state.

We now give the right-hand sides for this analysis. Initially, local variables are uninitialized (i.e., may have any value) and no global variables are protected (because no mutexes are held), so no relations are known, indicated by $\top$. Additionally, every unknown $[m]$ ($m \in \mathcal{M}$) is initialized using a side effect with the initial value 0 for

every protected global:

$$\begin{aligned}\mathsf{init}^\sharp \stackrel{\text{def}}{=} \ & \textbf{let } r\, V = [\![g \leftarrow 0 \mid g \in V]\!]^\sharp_{\mathbb{R}}\, \top \textbf{ in} \\ & \textbf{let } \rho = \{[m] \mapsto r\, (\mathcal{G}_{\mathrm{w}}[m]) \mid m \in \mathcal{M}\} \textbf{ in} \\ & (\rho, \top),\end{aligned}$$

where $[\![g \leftarrow 0 \mid g \in V]\!]^\sharp_{\mathbb{R}}$ is the abstract transformer of $\mathbb{R}$ corresponding to the parallel assignment of 0 to all variables in V. In this case, the parallel assignment can be reduced to a sequence of individual assignments because they are independent.

As mentioned above, locking a mutex m simply refines the local state with the value of unknown $[m]$ by meet:

$$[\![[u, S], \mathsf{lock}(m)]\!]^\sharp\, \eta \stackrel{\text{def}}{=} (\varnothing, \eta\, [u, S] \sqcap \eta\, [m]).$$

This is sound because the value of $[m]$ only maintains relationships between variables write-protected by m, and its value soundly accounts for the program state at every $\mathsf{unlock}(m)$ and at initialization.

In this section, define the *restriction* of a relation $r \in \mathbb{R}$ to a subset $V \subseteq \mathcal{V}$ of variables as

$$r|_V \stackrel{\text{def}}{=} [\![\mathsf{havoc}(\mathcal{V} \setminus V)]\!]^\sharp_{\mathbb{R}}\, r,$$

which amounts to forgetting all information about variables not in V. Thus, unlocking a mutex m side-effects to $[m]$ the local state restricted to variables $\mathcal{G}_{\mathrm{w}}[m]$ protected by m, to make changes of globals visible to other threads. Moreover, the local state is restricted to local variables and only those globals which remain protected (by at least one protecting mutex):

$$\begin{aligned}[\![[u, S], \mathsf{unlock}(m)]\!]^\sharp\, \eta \stackrel{\text{def}}{=} \ & \textbf{let } r = \eta\, [u, S] \textbf{ in} \\ & \textbf{let } \rho = \left\{[m] \mapsto r|_{\mathcal{G}_{\mathrm{w}}[m]}\right\} \textbf{ in} \\ & \textbf{let } V = \mathcal{X} \cup \bigcup\nolimits_{m' \in S \setminus \{m\}} \mathcal{G}_{\mathrm{w}}[m'] \textbf{ in} \\ & (\rho, r|_V).\end{aligned}$$

With the atomicity mutex m_g surrounding accesses to g, non-relational information about g is associated with this mutex. Hence, all reads and writes refer to the local copy of g:

$$\begin{aligned}[\![[u, S], g \leftarrow x]\!]^\sharp\, \eta &\stackrel{\text{def}}{=} (\varnothing, [\![g \leftarrow x]\!]^\sharp_{\mathbb{R}}\, (\eta\, [u, S])), \\ [\![[u, S], x \leftarrow g]\!]^\sharp\, \eta &\stackrel{\text{def}}{=} (\varnothing, [\![x \leftarrow g]\!]^\sharp_{\mathbb{R}}\, (\eta\, [u, S]))\end{aligned}$$

by delegating to the abstract semantics of $\mathbb{R}$. Guards and assignments (which may only involve local variables) are defined analogously.

Example 39. Consider the following program, where $\mathcal{M}_w[g] = \{m1, m2, m_g\}$, $\mathcal{M}_w[h] = \{m1, m2, m_h\}$ and, thus, $\mathcal{G}_w[m1] = \mathcal{G}_w[m2] = \{g, h\}$:

```
1  int g = 0, h = 0;
2  mutex m1, m2;

3  void main() {
4    create(t1);
5    int x; havoc(x);
6    lock(m1);
7      lock(m2);
8        g = x; h = x + 9;
9      unlock(m2);
10     lock(m2);
11       h = x;
12       assert(g == x); // ①
13       assert(h == x); // ②
14     unlock(m2);
15   unlock(m1);
16   create(t2);
17 }

18 void t1() {
19   lock(m2);
20   unlock(m2);
21   lock(m1);
22     lock(m2);
23       assert(g == h); // ③
24       int y; havoc(y);
25       g = y; h = y;
26     unlock(m2);
27   unlock(m1);
28 }

29 void t2() {
30   lock(m2);
31     lock(m1);
32       assert(g == h); // ④
33     unlock(m1);
34   unlock(m2);
35 }
```

Using the octagon domain, our analysis succeeds in proving all assertions here. Thread `t2` is of particular interest: when locking `m2` only $g \leq h$ is known to hold, and locking the additional mutex `m1` means that the better information $g = h$ becomes available.

The analysis by Mukherjee et al. [Muk+17], on the other hand, only succeeds in proving assertion ② — even when all globals are put in the same region. It cannot establish assertion ①, because all correlations between locals and globals are forgotten when the *mix* operation is applied at the second lock(`m2`) in the `main` thread. Assertion ③ cannot be established because, at lock(`m2`) in `t1`, the mix operation also incorporates the state after the first unlock(`m2`) in the `main` thread, where $g = h$ does not hold. Similarly, for assertion ④.

The same applies for assertion ③ and the analysis using *lock invariants* proposed by Miné [Min14b]. This analysis also falls short of showing assertion ①, as at the lock(`m2`) in the `main` thread, the lock invariant associated with `m2` is joined into the local state. Assertion ④ is similarly out of reach.

Even after equipping them with thread IDs, the same reasoning also applies to the analyses of Mukherjee et al. [Muk+17] and Miné [Min14b], and the non-relational analyses from Section 5.4. □

Theorem 8. *Mutex-Meet is sound w.r.t. the local trace semantics.*

Proof. Deferred to the extended version of the original paper [Sch+23c, Appendix B]. ■

Remark 20 (Analysis of write-protected globals). Above we assumed the sets $\mathcal{G}_{\mathrm{w}}[m]$ as given for simplicity of presentation. Analogously to Remark 17 (p. 147), these can be computed by pre-analysis or jointly with the analysis itself. However, in the latter case, additional care has to be taken to remain sound.

Specifically, the set $\mathcal{G}_{\mathrm{w}}[m]$ for a mutex $m \in \mathcal{M}$ may decrease between a side effect to $[m]$ (in $\mathsf{unlock}(m)$) and a subsequent use of $[m]$ (in $\mathsf{lock}(m)$). In such a situation, the abstract value of $[m]$ may still include constraints on global variables which the mutex no longer protects and which would unsoundly refine the local state when locking. This issue is avoided by also restricting the abstract state to $\mathcal{G}_{\mathrm{w}}[m]$ when locking (highlighted):

$$[\![[u, S], \mathsf{lock}(m)]\!]^\sharp\, \eta \stackrel{\text{def}}{=} \left(\varnothing, \eta\,[u, S] \sqcap (\eta\,[m])|_{\mathcal{G}_{\mathrm{w}}[m]}\right).$$

This is a non-issue for Vojdani's analysis and Protection-Based Reading: both rely on join ($\sqcup$) instead of meet ($\sqcap$) and already include such checks to avoid joining in unnecessary information. □

5.5.2. Improvements

The analysis in the previous section has numerous deficiencies, because all writes to a global are accumulated regardless of the writing thread. Moreover, the analysis forgets values of globals when they become unprotected, in contrast to some other approaches [Min14b; Muk+17].

Precision of the analysis can be improved by soundly limiting which writes a thread may actually read from based on MHP information. We propose such improvements to the analysis:

- **I1.** Exclude earlier writes from the ego thread which have been overwritten, if the thread is unique [Sch+23b, Section 7], analogously to **RI2**.
- **I2.** Exclude writes from threads which have not yet been created according to the thread IDs from Section 3.5.2 [Sch+23b, Section 6], analogously to **RI3**.
- **I3.** Exclude earlier overwritten writes from ancestor threads before the ego thread has been created according to the sets C from Section 3.5.3 [Sch+23c, Appendix E], analogously to **RI4**.
- **I4.** Exclude earlier writes from threads which have already been joined according to the sets J from Section 3.5.3 [Sch+23c, Appendix F], analogously to **RI5**.

The improvements are illustrated on the following examples, while their technical details are deferred to the cited original paper.

Example 40 (Overwritten self-writes)**.** Consider the following program, where $\mathcal{M}_{\mathrm{w}}[\mathtt{g}] = \{m_{\mathtt{g}}\}$:

```
1 int g = 0;

2 void main() {
3   g = 3;
4   g = 2;
5   assert(g == 2);
6   g = 8;
7 }
```

The Mutex-Meet analysis from the previous section cannot prove the assertion, because each thread could read all its own past (and future!) writes.

With improvement **I1**, the assertion can be proven, as the past (and the future) write of the unique `main` thread are excluded. □

Example 41 (Writes from not-yet-created threads)**.** Consider the following program, where $\mathcal{M}_{\mathrm{w}}[\mathtt{g}] = \{\mathtt{m}, m_{\mathtt{g}}\}$, $\mathcal{M}_{\mathrm{w}}[\mathtt{h}] = \{\mathtt{m}, m_{\mathtt{h}}\}$ and, thus, $\mathcal{G}_{\mathrm{w}}[\mathtt{m}] = \{\mathtt{g}, \mathtt{h}\}$:

```
1  int g = 0, h = 0;
2  mutex m;

3  void main() {
4    lock(m);
5      int x; havoc(x);
6      g = x; h = x;
7    unlock(m);
8    lock(m);
9      assert(g == h);
10   unlock(m);
11   create(t);
12 }

13 void t() {
14   lock(m);
15     g = 19;
16   unlock(m);
17 }
```

The Mutex-Meet analysis, even with the previous improvement, cannot prove the assertion, because the thread `t` breaks the invariant $\mathtt{g} = \mathtt{h}$.

With improvement **I2**, the assertion can be proven, as the invariant-breaking thread has definitely not been started yet at that point. □

Example 42 (Overwritten writes from before thread creation)**.** Consider the following program, where $\mathcal{M}_w[g] = \{m, m_g\}$, $\mathcal{M}_w[h] = \{m, m_h\}$ and, thus, $\mathcal{G}_w[m] = \{g, h\}$:

```
1  int g = 0, h = 0;
2  mutex m;

3  void main() {
4    lock(m);
5      g = 5; h = 8;
6    unlock(m);
7    lock(m);
8      g = 10; h = 10;
9    unlock(m);
10   create(t);
11 }

12 void t() {
13   lock(m);
14     assert(g == h);
15   unlock(m);
16 }
```

The Mutex-Meet analysis, even with the previous improvements, cannot prove the assertion, because the thread `t` could read all of its ancestor's writes.

With improvement **I3**, the assertion can be proven, as invariant-breaking writes are overwritten before the thread `t` is created. □

Example 43 (Writes from joined threads)**.** Consider the following program, where $\mathcal{M}_w[g] = \{m, m_g\}$, $\mathcal{M}_w[h] = \{m, m_h\}$ and, thus, $\mathcal{G}_w[m] = \{g, h\}$:

```
1  int g = 0; h = 0;
2  mutex m;

3  void main() {
4    tid ttid = create(t);
5    lock(m);
6      g = 20; h = 20;
7    unlock(m);
8    join(ttid);
9    lock(m);
10     assert(g == h);
11   unlock(m);
12 }

13 void t() {
14   lock(m);
15     g = 4; h = 8;
16   unlock(m);
17   int x; havoc(x);
18   lock(m);
19     g = x; h = x;
20   unlock(m);
21 }
```

The Mutex-Meet analysis, even with the previous improvements, cannot prove the assertion, because the thread `t` breaks the invariant `g = h` associated with the mutex `m`.

With improvement **I4**, the assertion can be proven, as invariant-breaking writes are overwritten before the thread `t` is joined. □

5.6. Implementation

We have implemented the thread-modular analyses described in the previous sections within GOBLINT. Each analysis is implemented as a so-called *privatization* module.[57] The non-relational base analysis and the relational APRON analysis are parametrized by such a module, allowing the thread-modular aspects to be easily exchanged, while reusing the rest. GOBLINT implements the following (versions of) thread-modular analyses:

Baseline. Implemented as `none` privatization. Recently, a version which synchronized all globals at all nodes was replaced with the simpler and more efficient version which does not keep values of globals in local states at all.[58]

Miné's analysis. Implemented as `mine-W-noinit` privatization. For comparison purposes, GOBLINT also includes other versions (without W and with thread IDs).

Vojdani's analysis. Implemented as `vojdani` privatization. Recently, the version which avoids excessive synchronization was reintroduced.[58] Ever since its original introduction in 2011, the only non-baseline privatization in GOBLINT was a version of Vojdani's analysis with lazy reading, however, it turned out to be unsound and prompted the design of Protection-Based Reading.

Protection-Based Reading. Implemented as `protection` privatization. The implementation follows Remark 16 (p. 146) to minimize local states, but implicitly uses $\operatorname{dom}\sigma$ as W. Moreover, the `protection-read` privatization implements the version from Remark 18 (p. 149). The latter is also the default.

Lock-Centered Reading. Implemented as `lock` privatization. The implementation follows Remark 19 (p. 152) to not read initial values in some cases.

Write-Centered Reading. Implemented as `write` privatization.

Combined Reading. Implemented as `write+lock` privatization.

Mutex-Meet. Implemented as `mutex-meet` privatization. This is available both as a non-relational and a relational version for the base and the APRON analyses, respectively. It is the default for the latter. GOBLINT includes various `mutex-meet-tid*` privatizations which implement improvements from the previous section.

GOBLINT performs these thread-modular analyses jointly with the analysis of protecting mutexes (Remarks 15, 17 and 20, pp. 139, 147 and 161). The solvers in GOBLINT can handle the non-monotonicity in the side effects this entails.

[57] Historically, GOBLINT only implemented Vojdani's analysis aka privatization (Section 5.3.3). Hence, the term which has obtained a more general meaning in the implementation of this work.

[58] `https://github.com/goblint/analyzer/pull/1552`

5.6.1. Differences from Theory

Implicit Atomicity Mutexes. While the mutexes $\mathcal{M}_{\mathcal{G}}$ are convenient for formalization, the implementation does not make them explicit. Instead, the sequence of $\mathsf{lock}(m_g)$, global read/write and $\mathsf{unlock}(m_g)$ are handled together with

$$[\![[u, S], \mathsf{lock}(m_g); x \leftarrow g; \mathsf{unlock}(m_g)]\!]^\sharp$$

and

$$[\![[u, S], \mathsf{lock}(m_g); g \leftarrow x; \mathsf{unlock}(m_g)]\!]^\sharp.$$

These transfer functions are defined as compositions of the three individual transfer functions. In most cases, when the latter three are inlined, numerous simplifications are possible.

Expressions. The language described in Section 5.2.1 requires that globals are read from or written to locals only. In particular, this means that expression assignments involving globals are transformed accordingly. The implementation in GOBLINT can perform such a transformation implicitly on the fly:

- In the relational APRON analysis, each global g occurring in the right-hand side of an assignment is replaced with a temporary (local) variable g_{in} and preceded by a corresponding global read $g_{\text{in}} \leftarrow g$. If a global g occurs in the left-hand side of an assignment, then it is replaced with a temporary (local) variable g_{out} and followed by a corresponding global write $g \leftarrow g_{\text{out}}$.
- In the non-relational base analysis, the temporary local variables are avoided altogether by working with abstract values directly. Reading a global (in the middle of a larger expression) returns its non-relational abstract value without requiring a local variable to store it in. Writing a global takes such an abstract value as argument instead of a local variable to read it from.

Splitting by Lockset. In practice, the set of mutexes $\mathcal{M}$ is not known ahead of time as they can be created dynamically in C. This makes the constraint system from Section 5.2.4 with unknowns $[u, S]$ and domain $\mathbb{D}$ difficult to implement. Instead, the splitting by lockset is achieved using path-sensitivity, whereby the unknowns are $[u]$ and the domain is $2^{\mathcal{M}} \rightharpoonup \mathbb{D}$, only maintaining abstract values for those locksets which may reach $u \in \mathcal{N}$. Flow-insensitive unknowns for the various analyses are transformed similarly.

Using Must-Locksets. The splitting by lockset can be abandoned altogether with the baseline, Vojdani's analysis, Protection-Based Reading and Mutex-Meet. This can improve efficiency at the cost of losing some precision. To this end, it suffices to additionally track for each program location, as part of the local state, a set of must-held mutexes from the domain $2^{\mathcal{M}}$ (ordered by superset). Thus, the unknowns are $[u]$ and the domain is $2^{\mathcal{M}} \times \mathbb{D}$. This is supported by the GOBLINT implementation but not used for the evaluation.

Multi-threaded Mode. When the analyses above are applied to a single-threaded program, values of global variables would still be analyzed completely flow-insensitively (especially because a single-threaded program is unlikely to use any mutexes), yielding very imprecise results. The analyses have similar imprecision during the single-threaded initialization phase of a multi-threaded program, where no threads have been created yet and global variables can be safely manipulated without mutexes.

Therefore, GOBLINT distinguishes two phases of a program: when it is definitely single-threaded, and when it has possibly become multi-threaded. This is accomplished by the same thread flag domain which is used to exclude single-threaded accesses from races (Section 3.5.4). In the first phase, global variables are analyzed flow-sensitively (akin to local variables). In the second phase, thread-modular analyses are actually used. It is during the transition between these phases where $\mathsf{init}^\sharp$-like steps are performed:

1. current values of globals are side-effected from the local state as the initial values for the globals (instead of $0^\sharp$),
2. globals in the local state are set to $\bot/\top$ or removed (depending on the analysis).

5.6.2. Assuming Globals

Example 44. Section 5.2.1 requires that conditional $\mathsf{assume}(e)$ statements only contain local variables in the expression $e \in \mathcal{E}$. For example,

$$\mathsf{assume}(\mathsf{g} < 10);$$
$$\mathsf{assert}(\mathsf{g} < 10)$$

must be represented as

$$\mathsf{lock}(m_\mathrm{g}); \mathsf{x} \leftarrow \mathsf{g}; \mathsf{unlock}(m_\mathrm{g});$$
$$\mathsf{assume}(\mathsf{x} < 10);$$
$$\mathsf{lock}(m_\mathrm{g}); \mathsf{x} \leftarrow \mathsf{g}; \mathsf{unlock}(m_\mathrm{g});$$
$$\mathsf{assert}(\mathsf{x} < 10).$$

If this occurs in a critical section where g is protected (by a program mutex), then an analysis should be able to prove the assertion.

The relational Mutex-Meet (Section 5.5.1), using a sufficiently expressive domain (e.g., octagons), can do it as follows. The first assignment results in the state $\{\mathsf{x} = \mathsf{g}\}$. The subsequent assumption refines the state to $\{\mathsf{x} = \mathsf{g}, \mathsf{x} < 10, \mathsf{g} < 10\}$, where $\mathsf{g} < 10$ is implied by the other two. Hence, the second assignment yields the exact same abstract state. Therefore, the assertion can be proven directly.

However, all of the presented non-relational analyses fail to prove the assertion because $\mathsf{assume}(\mathsf{x} < 10)$ only refines the local x and, without the two-variable equality, the refinement does not propagate to g. □

Remark 21 (Non-relational analyses). The issue from the previous example can be alleviated by introducing an additional assignment from the refined auxiliary local variable back to the corresponding global variable. For the above example, this means the following representation:

$$\begin{aligned}
&\mathsf{lock}(m_{\mathrm{g}}); \mathsf{x} \leftarrow \mathsf{g}; \mathsf{unlock}(m_{\mathrm{g}});\\
&\mathsf{assume}(\mathsf{x} < 10);\\
&\mathsf{lock}(m_{\mathrm{g}}); \mathsf{g} \leftarrow \mathsf{x}; \mathsf{unlock}(m_{\mathrm{g}});\\
&\mathsf{lock}(m_{\mathrm{g}}); \mathsf{x} \leftarrow \mathsf{g}; \mathsf{unlock}(m_{\mathrm{g}});\\
&\mathsf{assert}(\mathsf{x} < 10).
\end{aligned}$$

If this occurs in a critical section where g is protected (by a program mutex), then a sufficiently precise analysis can deduce that the value of g cannot have changed between the additional write and the final read, hence, it must have the refined value. □

Remark 22 (Refining writes to globals). Note that the refined abstract value written back to the global is necessarily included in its flow-insensitive abstraction. With this in mind, these additional writes can be treated specially and optimized:

1. Do not emit unnecessary side effects to unprotected values/weak interferences, etc.
2. Do not add the global to the W set/map in Miné's analysis (Section 5.3.2), optimized Protection-Based Reading (Section 5.4.1, Remark 16), Write-Centered Reading (Section 5.4.3) and Combined Reading (Section 5.4.4). These data structures are just used at unlock to side-effect protected values/synchronized interferences, etc. where the abstract value is also already included.

Nevertheless, the global must be added to σ, P (Protection-Based Reading, Write-Centered Reading), L (Lock-Centered Reading), etc. in order for the refinement to have any effect and persist for subsequent reads protected by a program mutex. □

5.7. Evaluation

We have evaluated the proposed and implemented analyses in terms of precision and performance. The following subsections present relevant parts of the evaluations from the original papers [Sch+21a; Sch+23b]. These analyses have been recently re-evaluated by Schwarz [Sch25, Chapter 5].

5.7.1. Non-relational Analyses

Although SV-COMP has multi-threaded programs in the *ConcurrencySafety* category, most of them are small and require intricate analysis of interleavings, which is out of scope for our thread-modular analyses. Hence, we assemble a set of larger benchmarks to draw conclusions about scalability:

- six multi-threaded POSIX programs from the GOBLINT benchmark suite,

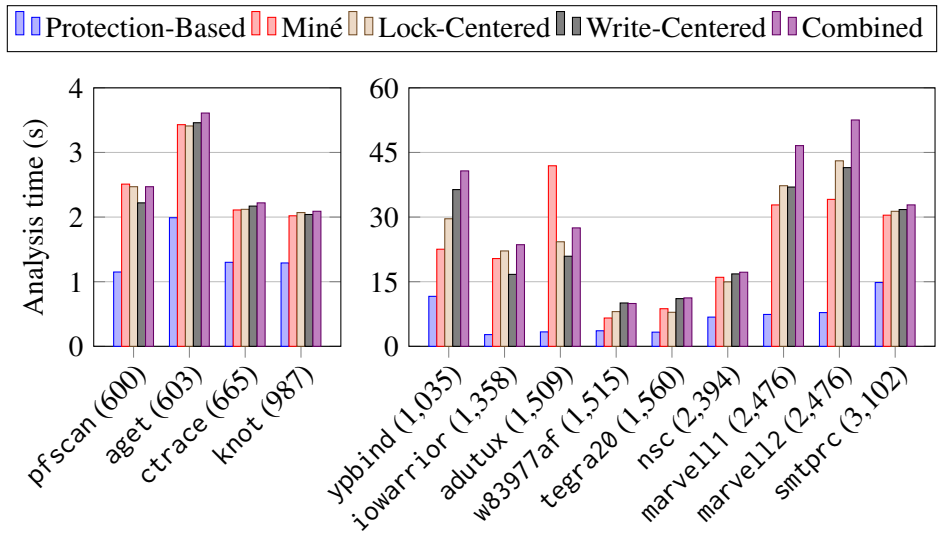


Figure 16. Non-relational analysis times for GOBLINT benchmarks (LLoC in parentheses)

- seven large `ldv-linux-3.14-races/` benchmarks from the *ConcurrencySafety* category of SV-COMP 2021 [Bey21].

The SV-COMP programs are subsequently included into the GOBLINT benchmark suite. Overall, the programs range from 1,280 to 12,778 LoC, with logical lines of code (LLoC) being between 600 and 3,102.

The analyses are performed context-sensitively with a standard points-to analysis for addresses and inclusion/exclusion sets as the domain for integer values. The evaluation was carried out on Ubuntu 20.04.1 and OCAML 4.11.1, running on an AMD EPYC 7742 processor.

Performance. We analyzed each of the programs with each of the non-relational analyses, and the analysis times are presented in Fig. 16. On smaller programs, Protection-Based Reading is almost twice as fast as the others, which have very similar running times. On larger programs, the differences are much larger: Protection-Based Reading is between 2 and 6 times faster than the others, whose running times tend to grow with their sophistication.

Precision. Since the analyses use different local and global domains, their precision cannot be compared directly via the constraint system solutions. Instead, we record and compare the observable behavior in the form of abstract values of global variables read at program locations. Our comparison reveals that for 11 out of 13 programs, all analyses are equally precise. For the remaining two programs, `pfscan` and `ypbind`, all but Miné's analysis are equally precise, while Miné's was less precise at 6% and 16% of global reads, respectively.

Thus, our experiments indicate that Protection-Based Reading offers similar precision at a significantly shorter analysis time, while the more involved Lock- and Write-Centered Reading do not pay off in practice. Moreover, the incomparability identified in the introduction can in fact be observed on at least some real-world programs.

5.7.2. Relational Analysis

We evaluate the relational Mutex-Meet analysis (Section 5.5.1) by comparing three different GOBLINT configurations:

1. Mutex-Meet with APRON analysis using the octagon domain, but also with non-relational base analysis for non-numeric values.
2. Mutex-Meet with APRON analysis using the box domain, but also with non-relational base analysis for non-numeric values.
3. Protection-Based Reading (Section 5.4.1) with base analysis using the interval domain.

A comparison with other tools is difficult; for details see the extended version of the original paper [Sch+23c, Section 9 and Appendix I]. All benchmarks were run in a virtual machine on an AMD EPYC 7742 processor running Ubuntu 20.04. The results of our evaluation are summarized in Table 17.

Our Tests. To capture particular challenges for multi-threaded relational analysis, we constructed a set of test benchmarks (including the examples from the original paper [Sch+23b]). The programs are equipped with assertions and grouped by features required to verify them. In particular, the group of relational tests distinguishes Mutex-Meet with the box and the octagon domain, as shown in the corresponding row of Table 17.

Table 17. Relational analysis precision summary in the number of assertions: all assertions proven (✓), some but not all assertions proven (number) or no assertions proven (✗)

				Mutex-Meet		Protection-
Set	Group	#	Asserts	Box	Octagon	Based
Our	Basic	3	4	✓	✓	✓
	Relational	10	35	✗	✓	4
GOBLINT	POSIX	5	1,679	1,146	1,490	1,582
	SV-COMP	7	360	✓	✓	✓
RATCOP		19	34	4	14	6
Total		44	2,112	1,514	1,903	1,956

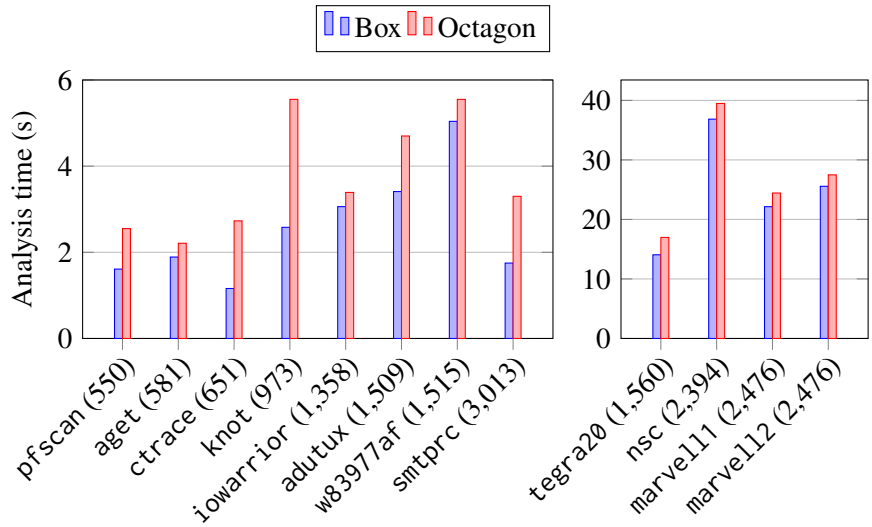


Figure 17. Relational analysis times for GOBLINT benchmarks (LLoC in parentheses)

GOBLINT Benchmarks (Section 5.7.1). Naturally, we evaluated these analyses on the benchmarks from the previous section. However, the `ypbind` benchmark was excluded due to abstractly creating a thread from an unknown ($\top$) function pointer, which was not supported by Mutex-Meet at the time.

Performance. Analysis times and program sizes are shown in Fig. 17. For 7 out of 12 benchmarks, the performance difference between the box and the octagon domain is relatively small. For the rest, analysis using the octagon domain takes up to twice the analysis time compared to using the box domain.

Precision. Unlike the other benchmark sets considered here, these benchmarks do not contain assertions. The previous section compared precision via abstract values of read global variables, however, these are non-relational. Thus, that precision comparison method does not apply well to the relational case. Instead, we compare precision via witnesses (as described in Section 2.5.4): we used the Mutex-Meet analysis with all improvements from Section 5.5.2 to infer invariants at each locking operation.

The results of validating these witnesses using the analyses considered here are shown in Table 18. For 9 out of 12 benchmarks, all three configurations could prove all witness invariants. For the three remaining benchmarks, Mutex-Meet using the octagon domain instead of the box domain resulted in more invariants proven. Notably, Protection-Based Reading could validate 95% of the invariants, despite being non-relational and not using thread IDs, unlike the analysis used to infer them. Perhaps surprisingly, Protection-Based Reading can validate invariants which these configurations of Mutex-Meet cannot, possibly because the two approaches are incomparable also in the non-relational case.

Table 18. Relational analysis precision in the number of invariants for GOBLINT benchmarks: all invariants proven (✓) or some but not all invariants proven (number)

Benchmark	Invariants	Mutex-Meet		Protection-Based
		Box	Octagon	
`pfscan`	221	14	32	165
`aget`	10	2	✓	✓
`ctrace`	1,448	1,130	✓	1,407
`iowarrior`	6	✓	✓	✓
`w83977af`	2	✓	✓	✓
`nsc`	24	✓	✓	✓
`marvell1`	200	✓	✓	✓
`marvell2`	128	✓	✓	✓
(other 4)	0	✓	✓	✓
Total	2,039	1,506	1,850	1,942

RATCOP Benchmarks [Muk+17]. The small JAVA programs were manually translated to C. Mutex-Meet with the octagon domain succeeded in proving most assertions that the best configuration of RATCOP could, while RATCOP required the polyhedra domain in one case. With the improvements from Section 5.5.2, it succeeded in proving *all* assertions that RATCOP could.

5.7.3. Threats to Validity

Construct Validity. Precision of thread-modular analyses is the focus of this entire chapter, from the motivation to our improvements. Therefore, we evaluate the relative precision of the analyses in practice. We also investigate the benefit of relational domains for thread-modular analysis.

Generally, there is a trade-off between precision and performance. As thread-modular abstract interpretation strives for fast analysis, we additionally evaluate the performance of the analyses.

Internal Validity. Unlike all other evaluations in this thesis, the BENCHEXEC framework (described in Section 3.4.6) was *not* used here. However, we believe that the performance measurements were still reliable. GOBLINT itself is single-threaded and does not spawn subprocesses for solvers, etc. Furthermore, only a single GOBLINT instance was running at a time on an otherwise idle machine. As such, there should have been no hidden unaccounted costs nor interference between benchmark runs.

Precision is more difficult to quantify than performance, especially on small benchmark suites consisting of large unlabeled programs. One way would be

to count the number of certain GOBLINT warnings (e.g., data races), with fewer meaning more precise. While it is reflective of precision from the user's perspective, it only indirectly depends on the value analysis: more precise abstract values may not be enough to reduce such warnings. Thus, we devised methods to compare the abstractions more directly, but independently of their incompatible representations.

Precision comparison is further complicated by the context-sensitivity of the analyses. Different analyses can give rise to different sets of contexts for a function, possibly making context-to-context comparison impossible. Hence, in the non-relational precision evaluation, abstract values were joined over all contexts before comparison. The same happens during witness generation, which is used in the relational precision evaluation on the GOBLINT benchmarks. These joins can introduce additional imprecision which may cover up and, thus, underestimate precision differences between analyses.

Precision comparison is also susceptible to *carry-over effects* [BS23a], meaning that improved precision at one program location trivially carries over to the next location because the intermediate statement only concerns unrelated variables. Such effects overestimate meaningful precision differences between analyses. We have not used mitigation techniques such as those more recently proposed by Ballou and Sherman [BS23a].

External Validity. The GOBLINT benchmark suite, as considered here, only consists of 13 medium-sized real-world programs. They are large enough to reveal some performance differences between the analyses, but scalability to larger programs is unknown. The benchmarks may be biased towards programs with simple protection patterns which do not benefit from the more advanced analyses. Schwarz [Sch25, Chapter 5] has recently re-evaluated these analyses on additional real-world programs.

Our tests in the relational analysis evaluation are specifically constructed to highlight the precision of our analyses. Thus, they imply nothing about generalizability and only serve as a baseline for future work.

5.8. Related Work

The idea of tracking the events possibly affecting a particular local thread configuration goes back to Lamport [Lam78] (see also Steen and Tanenbaum [ST17]). It has also been used extensively for the verification of concurrent systems via separation logic [Bro07; Bro14; Kre+17; LN13; Nan+14; SNB15; Nan+19].

In the context of data-flow analysis, a thread-local concrete semantics was also used in Mukherjee et al. [Muk+17] for proving the correctness of thread-modular analyses. The semantics there, however, is based on interleavings and permits stale values for unread globals. In contrast, we consider a partial order of past events and explicitly exclude the values of globals from local traces. These are instead recovered from the local trace by searching for the last preceding write at the point when the value of the global is accessed.

For abstract interpretation, Miné [Min12] presents a thread-modular abstract interpretation framework as layers of abstractions of the interleaving semantics, and in subsequent work [Min14b; MM17], the interleaving semantics is encoded into a local semantics by using auxiliary variables to track the control location of other threads. In contrast, we have shown equivalence for a purely thread-local trace semantics where threads only learn about the actions of other threads through synchronizing events.

The integration of relational domains into thread-modular abstract interpretation was pioneered by Miné [Min14b]. His analysis is based on lock invariants, determining for each mutex a relation which holds whenever the mutex is not held. Weak interferences are used to account for asynchronous variable accesses. For practical analyses, a relational abstraction only for lock invariants is proposed, while using a coarse, non-relational abstraction for the weak interferences. This framework closely follows the framework for non-relational analysis [Min12], while abandoning background locksets. Our relational analysis, on the other hand, maintains at each mutex m only relations between variables write-protected by m. For these relations, more precise results can be obtained since they are incorporated into the local state at locks by meet ($\sqcap$), while Miné [Min14b] uses join ($\sqcup$).

Monat and Miné [MM17] present an analysis framework which is orthogonal to our approach. It is tailored to the verification of algorithms that do not rely on explicit synchronization via mutexes, such as the Bakery algorithm. Suzanne and Miné [SM18] extend it to handle weak memory effects (PSO, TSO) by incorporating memory buffers into the thread-local semantics. The notion of interferences is also used by Sharma and Sharma [SS21] for the analysis of programs under the release/acquire memory model of C11 by additionally tracking abstractions of modification sequences for global variables. They consider fixed finite sets of threads only, and do not deal with thread creation or joining.

Earlier works on thread-modular relational analysis rely on DATALOG rules to model interferences in the sense of Miné [Min12] in combination with abstract interpretation applied to the data-flow graph [FK12] or the CFG [KW16] (later extended to weak memory [KW17]), respectively. The latter realize improvements **I2** and **I4** by reading thread IDs and joined threads off the CFGs [KW16]. In our approach, however, this information is computed *during* analysis.

In most of these approaches, clusters of variables (if there are any) are predefined and not treated specially by the analysis. Only the thread-modular analysis proposed by Mukherjee et al. [Muk+17] differs by propagating information from unlocks to locks. It is relational for the locals of each thread, and within disjoint subsets of globals, called *regions*. These regions must be determined beforehand and must satisfy *region race freedom*. In contrast, the only extra a priori information required by our analysis are the sets of (write-)protecting mutexes of globals, which can be computed during the analysis itself. The closest concept within our approach to a region is the set of globals jointly protected by a mutex. These sets may overlap — which the analysis explicitly exploits.

5.9. Conclusion and Future Work

We have provided enhanced versions of the analyses by Miné [Min12], as well as by Vojdani [Voj10; Voj+16]. To Miné's original analysis, we added lazy reading of globals and restricting local copies of globals to the values written by the ego thread. Vojdani's approach was purged of the assumption of common protecting mutexes, while, additionally, background locksets are taken into account to exclude certain written values from being read. Moreover, we constructed a novel relational analysis based on the concept of common protecting mutexes. For a better comparison, we relied on side-effecting constraint systems as a convenient framework within which all analyses could be formalized. That framework also enabled us to specify a concrete semantics of local traces, w.r.t. which all analyses could be proven correct. We also provided an implementation of all these analyses, and practically compared them for precision and efficiency. Interestingly, the simplest of all analyses still provided decent precision while outperforming the others.

Future Work. There are still plenty of practical and technical nuances that we have not explored here. For example, the given comparison and enhancements refer just to the first and the most fundamental analysis introduced by Miné.

Weak Memory. Since we have only considered sequential consistency, naturally one wonders about weak memory models. They have been considered in other works on abstract interpretation [Fer08; Alg+11b; SM16; SM18]. Our local trace semantics, in particular our causality order, and the standard axiomatic approach to weak memory [AMT14] share some similarities. However, there is also a crucial difference: the axiomatic approach defines and decides the validity of individual concrete executions, whereas our semantics captures and abstracts all valid executions reaching a location. A deeper understanding of this connection can be key to adapting our semantics and analyses to weak memory models.

Alternative Scheduling. Miné's analysis (Section 5.3.2) was originally designed for mono-processor real-time scheduling which also uses thread priorities. In order to apply our analyses in such alternative settings, it may be necessary to make additional adaptations. GOBLINT has been used as the testbed for such analyses in the past [Sch14].

Assuming Globals. Section 5.6.2 described a technique for performing assume on globals. Later evidence has revealed a more precise approach for Protection-Based Reading, whereby actually written and assumed values are tracked separately.[59] It is to be determined whether a similar technique could apply to the other analyses.

Initialization of Globals. Unlike all other analyses presented here, Miné's does not use side effects for initialization of globals. Instead, it propagates current values

[59] https://github.com/goblint/analyzer/pull/1693#pullrequestreview-2636985612

of globals (which may be initial) directly to ancestor threads via local states. As per Footnote 47 (p. 135), the different approaches to initialization may be subtly different in precision. Thus, it may be interesting to consider adapting our analyses to Miné's approach.

Non-relational Mutex-Meet. Although the relational Mutex-Meet can easily be restricted to the non-relational case, we have not compared that restriction to other non-relational analyses. For example, precision results in Section 5.7.2 indicate that Protection-Based Reading and non-relational Mutex-Meet may be incomparable. Thus, an in-depth theoretical and practical investigation is warranted.

6. THREAD-MODULAR CORRECTNESS WITNESSES

The two previous chapters focused on two independent topics: witnesses and thread-modular abstract interpretation. This ultimate chapter joins the two branches to propose *thread-modular correctness witnesses* and fill the research gap, addressing goal **G4**. Moreover, it validates the proposals by showing them in action, addressing goal **G5**.

6.1. Introduction

Recall from Table 1 on page 19 that there exists a violation witness format for multi-threaded programs [BF20]. It extends the automaton-based violation witness format for single-threaded programs (Section 2.5.1) to describe interleavings. A correctness witness for a multi-threaded program must cover all possible interleavings of threads. A naïve combination of single-threaded correctness and multi-threaded violation witnesses automata amounts to explicitly enumerating all such interleavings. However, there are far too many interleavings for all but the smallest programs, rendering the direct approach impractical. We are interested in scaling to real-world concurrent programs and, therefore, focus on a modular approach.

We approach the problem of thread-modular correctness witnesses from two different angles:

The local view (Section 6.2). The first approach avoids the need to modify the correctness witness formalism and format in any way. Thread-modular invariants computed by analyses are not directly exported, but rather *local views* from the perspective of certain program locations are used. This has the downside that the semantics of the original thread-modular invariant is completely forgotten: the resulting witness invariants provide no insight into how they were inferred.

The global view (Section 6.5). The second approach avoids slicing the thread-modular invariants into semantically-opaque local views. Instead, the invariants are enriched with *ghost variables*, which can correspond to semantic features used by the thread-modular analyses but are not expressible using ordinary programming language expressions. This allows the invariants to reflect their intended semantics. However, the downside is that the witness formalism and format need to be modified.

After illustrating both approaches by example, the subsequent sections describe generation and validation of the corresponding witnesses. Moreover, the approaches are implemented and separately evaluated.

6.2. The Local View

The method of unassume generalizes effortlessly from single-threaded to multi-threaded programs since the unassume operations are simply part of control flow. Any concurrency information is implicit in which threads and how reach those program locations. Witness invariants describe the *local view* of the global state for reaching thread executions. This view includes the values of global variables, the heap, concurrently running threads, state of synchronization primitives, etc. Hence, such witnesses are completely agnostic of the concurrency analysis and allow invariants to be exchanged between different thread-modular analyses regardless of the form of thread-modular invariants they use. For example, locksets which might be associated with such an invariant within the analysis are implied by locksets which threads can have when reaching the location. Since a witness validator may not assume the witness to be correct, it would still have to compute such information itself for checking purposes, so the validator should be able to infer that information on its own, without it being explicit in the witness.

This approach is supported by *rely-guarantee* reasoning of concurrent programs [Jon83], where relied invariants describe what a thread depends on from the surrounding environment, while guaranteed invariants describe what a thread ensures to the environment. If witness invariants are provided at mutex locking operations, then the invariants would be relied on in the following mutually excluded section. From these, an analysis implicitly derives what the thread guarantees at the end of such a section. If witness invariants are provided at mutex unlocking operations, then the invariants would be guaranteed by the mutually excluded section. From these, an analysis implicitly derives what other threads can rely on in their protected sections. However, the rely-guarantee model is only one possible intuition behind our approach — local traces (Section 5.2.3) provide another.

Unassuming. Single-threaded unassume, whether defined via assume or directly, can easily be lifted into the multi-threaded context to unassume invariants about global variables. While evaluating globals inside an invariant, they are read as specified by the thread-modular analysis. The resulting unassumed values for globals are written as specified by the thread-modular analysis. This ensures that the unassumed values are made visible to other threads.

Invariant Observability. The YAML witness format for sequential programs (Section 2.5.3) specifies where/when each location/loop invariant must hold. For concurrent programs, we must additionally define which global states the evaluation of an invariant may observe. Since witness invariants are not part of the program, they are also not governed by the C standard. However, the standard defines a set of *sequence points* where all side effects of the previous step are guaranteed to have been performed and no side effects of the following step have taken place [ISO11, Section 5.1.2.3]. Thus, we relax when witness invariants are required to hold: it suffices that an invariant holds whenever one thread is at the location of the invariant and every other thread is at a sequence point.

Listing 4. Program for illustrating the local view

```
1  int g = 0;
2  mutex m;

3  void main() {
4    create(inc);
5    create(dec);
6  }

7  void inc() {
8    while (1) {
9      lock(m);
10       if (g < 10)
11         g++;
12     unlock(m);
13   }
14 }

15 void dec() {
16   while (1) {
17     lock(m);
18       if (g > -10)
19         g--;
20     unlock(m);
21   }
22 }
```

Illustrative Example. We demonstrate our technique on the multi-threaded program depicted in Listing 4. The program creates two concurrently-executing threads (`inc` and `dec`), which modify the variable while holding the mutex `m`. Since the `main` thread does nothing else, we omit it in the examples below.

We analyze this program using Protection-Based Reading (Section 5.4.1), which maintains two flow-insensitive unknowns (protected and unprotected) for the variable `g`. Since all accesses of `g` are protected by `m`, the unprotected unknown will never be queried by the analysis. Thus, we will omit it in the examples below. Furthermore, we follow Remark 16 to minimize local states and Section 5.6.2 to branch on globals. Widening is employed to speed up the convergence of flow-insensitive unknowns, which are never narrowed (see Remark 1, p. 34).

Example 45. The protected unknown for `g` starts out with the initial interval $[0, 0]$. First, analyzing the `inc` thread, its first loop iteration publishes $[1, 1]$ to the protected unknown after line 12. Standard interval widening accelerates it to $[0, \infty]$. Iterating the loop in the `inc` thread to a fixpoint, all published abstract values are already subsumed by the widened abstract value.

Then, analyzing the `dec` thread, its first loop iteration publishes $[-1, \infty]$ to the protected unknown after line 20. Standard interval widening accelerates it to $\top$. Iterating the loop in the `dec` thread to a fixpoint, all published abstract values are already subsumed by the widened abstract value.

Since the `inc` thread depends on the changed protected unknown, it is reanalyzed, but all published abstract values are already subsumed by the protected abstract value, so the analysis reaches a fixpoint over all threads. Therefore, the analysis fails to establish bounds for `g`. Table 19 lists the abstract evaluation steps of this analysis in order. □

Before demonstrating our final approach, the following example illustrates the desired benefits using a simpler initialization-based approach (Section 4.3).

Table 19. Analysis progress and result (highlighted) for the program in Listing 4 from scratch

	g interval		
Line	Local	Protected	Comment
9	$\bot$	$[0, 0]$	
10	$\bot$	$[0, 0]$	
11	$[0, 0]$	$[0, 0]$	
12	$[1, 1]$	$[0, 0]$	Local: $\bot \sqcup [1, 1]$
13	$\bot$	$[0, \infty]$	Protected: $[0, 0] \nabla [1, 1]$
11	$[0, 9]$	$[0, \infty]$	
12	$[1, \infty]$	$[0, \infty]$	Local: $[10, \infty] \sqcup [1, 10]$
13	$\bot$	$[0, \infty]$	Protected: $[0, \infty] \nabla [1, \infty]$
9	$\bot$	$[0, \infty]$	Local: $\bot \nabla \bot$
9	$\bot$	$[0, \infty]$	Local: $\bot \Delta \bot$
17	$\bot$	$[0, \infty]$	
18	$\bot$	$[0, \infty]$	
19	$[0, \infty]$	$[0, \infty]$	
20	$[-1, \infty]$	$[0, \infty]$	Local: $\bot \sqcup [-1, \infty]$
21	$\bot$	$\top$	Protected: $[0, \infty] \nabla [-1, \infty]$
19	$[-9, \infty]$	$\top$	
20	$\top$	$\top$	Local: $[-\infty, -10] \sqcup [-10, \infty]$
21	$\bot$	$\top$	Protected: $\top \nabla \top$
17	$\bot$	$\top$	Local: $\bot \nabla \bot$
17	$\bot$	$\top$	Local: $\bot \Delta \bot$
11	$[-\infty, 9]$	$\top$	
12	$\top$	$\top$	Local: $[10, \infty] \sqcup [-\infty, 10]$
13	$\bot$	$\top$	Protected: $\top \nabla \top$
9	$\bot$	$\top$	Local: $\bot \nabla \bot$
9	$\bot$	$\top$	Local: $\bot \Delta \bot$

Table 20. Analysis progress and result (highlighted) for the program in Listing 4 starting with the flow-insensitive invariant $[-10, 10]$

	g interval		
Line	Local	Protected	Comment
9	$\bot$	$[-10, 10]$	
10	$\bot$	$[-10, 10]$	
11	$[-10, 9]$	$[-10, 10]$	
12	$[-9, 10]$	$[-10, 10]$	Local: $[10, 10] \sqcup [-9, 10]$
13	$\bot$	$[-10, 10]$	Protected: $[-10, 10] \nabla [-9, 10]$
9	$\bot$	$[-10, 10]$	Local: $\bot \nabla \bot$
9	$\bot$	$[-10, 10]$	Local: $\bot \Delta \bot$
17	$\bot$	$[-10, 10]$	
18	$\bot$	$[-10, 10]$	
19	$[-9, 10]$	$[-10, 10]$	
20	$[-10, 9]$	$[-10, 10]$	Local: $[-10, -10] \sqcup [-10, 9]$
21	$\bot$	$[-10, 10]$	Protected: $[-10, 10] \nabla [-10, 9]$
17	$\bot$	$[-10, 10]$	Local: $\bot \nabla \bot$
17	$\bot$	$[-10, 10]$	Local: $\bot \Delta \bot$

Example 46. Now, suppose that a witness provides the invariant $-10 \leq g \leq 10$ for the global variable. If the protected unknown for g starts out with the corresponding interval $[-10, 10]$, then the same analysis proceeds as follows.

First, analyzing the `inc` thread, its first loop iteration publishes $[-9, 10]$ to the protected unknown after line 12. The published abstract value is already subsumed by the protected abstract value and the loop in the `inc` is iterated to a fixpoint. Then, analyzing the `dec` thread, its first loop iteration publishes $[-10, 9]$ to the protected unknown after line 20. The published abstract value is already subsumed by the protected abstract value and the loop in the `dec` is iterated to a fixpoint.

Since the protected unknown did not change, the `inc` thread is not reanalyzed and the analysis reaches a fixpoint over all threads. Furthermore, it confirms the stronger invariant $[-10, 10]$ for g. Table 20 lists the analysis steps when initializing the protected unknown like this. □

This example shows two notable benefits. First, using a witness, the exact same analysis is able to verify a program which it could not on its own. Second, it can do so in many fewer steps, i.e., transfer function evaluations (cf. Tables 19 and 20). The problem is that the ability to initialize flow-insensitive unknowns like this is closely tied to a particular concurrency analysis.

Every thread-modular analysis has its own form of unknowns, for example, Miné's analysis (Section 5.3.2), Lock-Centered Reading (Section 5.4.2) and Write-

Listing 5. YAML witness for the program in Listing 4

```yaml
- entry_type: invariant_set
  metadata: # omitted
  content:
  - invariant:
      type: location_invariant
      location:
        file_name: listing-4.c
        line: 10
        function: inc
      value: -10 <= g && g <= 10
      format: c_expression
  - invariant:
      type: location_invariant
      location:
        file_name: listing-4.c
        line: 18
        function: dec
      value: -10 <= g && g <= 10
      format: c_expression
```

Centered Reading (Section 5.4.3) partition unknowns by background lockset. Hence, each analysis must provide its own means of incorporating witness invariants into initial abstract states. Therefore, we develop a more general technique, whereby thread-modular witness invariants are associated with program locations. The locations implicitly contain all the analysis-specific information, e.g., the current lockset.

Example 47. Now, suppose that a witness provides the invariant $-10 \leq g \leq 10$ for the `inc` thread on line 10 instead. For symmetry, it is also provided for the `dec` thread on line 18; otherwise, the result of the analysis would depend on the order of analyzing the threads. The corresponding YAML witness is shown in Listing 5.

This time, the protected unknown for `g` again starts out with the initial interval $[0, 0]$. However, we must take care to delay the application of widening operators using widening tokens (Section 4.6.1).

First, analyzing the `inc` thread, on line 10, the unassume operator incorporates the provided invariant as the abstract value $[-10, 10]_{\mathtt{i}}$, which has the widening token `i` attached. Continuing, the first loop iteration publishes $[-9, 10]_{\mathtt{i}}$ to the protected unknown after line 12 (where widening is delayed). Iterating the loop in the `inc` thread to a fixpoint, all published abstract values are already subsumed by the widened abstract value.

If widening tokens are not used, then the protected abstract value would be widened from the initial interval $[0, 0]$ to $\top$ already in the first loop iteration. Thus,

the analysis would immediately become as imprecise as in Example 45. Only difference being, this would take fewer steps to complete.[60]

Then, analyzing the dec thread, on line 18, the unassume operator incorporates the provided invariant as the abstract value $[-10, 10]_{\mathtt{d}}$, which has the widening token d attached. Continuing, the first loop iteration publishes $[-10, 9]_{\mathtt{d}}$ to the protected unknown after line 20 (where widening is delayed). Iterating the loop in the dec thread to a fixpoint, all published abstract values are already subsumed by the widened abstract value.

Since the inc thread depends on the changed protected unknown, it is reanalyzed, but all published abstract values are already subsumed by the protected abstract value, so the analysis reaches a fixpoint over all threads. Furthermore, it confirms the stronger invariant $[-10, 10]$ for g. Table 21 lists the analysis steps when using the unassume operator. Unlike Table 20, this witness-guided analysis took the same number of steps as the original. □

The incorporation of witness invariants at program locations is formalized as the unassume operation in Section 4.4. Compared to Example 46, this technique used the thread-modular analysis as a black box while guiding it with a witness. Therefore, we have achieved our goal of using the same thread-modular analysis, unmodified, to reverify the program.

Remark 23 (Performance improvement). Although in Example 47, witness-guided verification using the witness improved precision, it did not improve performance, unlike in the single-threaded Example 9 (p. 78). Nevertheless, it is also possible to obtain both improvements on the multi-threaded example. If the witness invariants are additionally provided before unlock(m) on lines 12 and 20, then the protected abstract value goes from $[0, 0]$ to $[-10, 10]_{\mathtt{i,d}}$ directly, avoiding the intermediate abstract value. In fact, since the program never reads the unprotected unknown, having the witness invariants provided *only* on those lines is just as good. If the program also read the global without protection, then the optimal witness provides the invariant in all four locations, which ensures that no unnecessary intermediate abstract values occur for either flow-insensitive unknown regardless of analysis order. □

Remark 24. The recent work of Stemmler et al. [Ste+25] adds narrowing of flow-insensitive unknowns (see Remark 1, p. 34). This allows the Protection-Based Reading analysis of the program in Listing 4 to also infer the precise invariant without guidance. In fact, they use a very similar program as their illustrative example. However, the additional narrowing requires more analysis steps than in Examples 45 and 47. Therefore, our witness-guided approach is still beneficial for speeding up the analysis. □

[60] "It doesn't work, but it's fast".

Table 21. Analysis progress and result (highlighted) for the program in Listing 4 unassuming invariant $-10 \leq g \leq 10$ on lines 10 and 18 with widening tokens (tagged i and d)

	g interval		
Line	Local	Protected	Comment
9	$\bot$	$[0,0]$	
10	$[-10,10]_{\mathtt{i}}$	$[0,0]$	Local: $[\![\mathsf{unassume}(-10 \leq g \leq 10)]\!]^\sharp$
11	$[-10,9]_{\mathtt{i}}$	$[0,0]$	
12	$[-9,10]_{\mathtt{i}}$	$[0,0]$	Local: $[10,10]_{\mathtt{i}} \sqcup [-9,10]_{\mathtt{i}}$
13	$\bot_{\mathtt{i}}$	$[-9,10]_{\mathtt{i}}$	Protected: $[0,0] \nabla [-9,10]_{\mathtt{i}}$
11	$[-10,9]_{\mathtt{i}}$	$[-9,10]_{\mathtt{i}}$	
12	$[-9,10]_{\mathtt{i}}$	$[-9,10]_{\mathtt{i}}$	Local: $[10,10]_{\mathtt{i}} \sqcup [-9,10]_{\mathtt{i}}$
13	$\bot_{\mathtt{i}}$	$[-9,10]_{\mathtt{i}}$	Protected: $[-9,10]_{\mathtt{i}} \nabla [-9,10]_{\mathtt{i}}$
9	$\bot_{\mathtt{i}}$	$[-9,10]_{\mathtt{i}}$	Local: $\bot \nabla \bot_{\mathtt{i}}$
9	$\bot_{\mathtt{i}}$	$[-9,10]_{\mathtt{i}}$	Local: $\bot_{\mathtt{i}} \Delta \bot_{\mathtt{i}}$
17	$\bot$	$[-9,10]_{\mathtt{i}}$	
18	$[-10,10]_{\mathtt{d}}$	$[-9,10]_{\mathtt{i}}$	Local: $[\![\mathsf{unassume}(-10 \leq g \leq 10)]\!]^\sharp$
19	$[-9,10]_{\mathtt{d}}$	$[-9,10]_{\mathtt{i}}$	
20	$[-10,9]_{\mathtt{d}}$	$[-9,10]_{\mathtt{i}}$	Local: $[-10,-10]_{\mathtt{d}} \sqcup [-10,9]_{\mathtt{d}}$
21	$\bot_{\mathtt{d}}$	$[-10,10]_{\mathtt{i},\mathtt{d}}$	Protected: $[-9,10]_{\mathtt{i}} \nabla [-10,9]_{\mathtt{d}}$
19	$[-9,10]_{\mathtt{d}}$	$[-10,10]_{\mathtt{i},\mathtt{d}}$	
20	$[-10,9]_{\mathtt{d}}$	$[-10,10]_{\mathtt{i},\mathtt{d}}$	Local: $[-10,-10]_{\mathtt{d}} \sqcup [-10,9]_{\mathtt{d}}$
21	$\bot_{\mathtt{d}}$	$[-10,10]_{\mathtt{i},\mathtt{d}}$	Protected: $[-10,10]_{\mathtt{i},\mathtt{d}} \nabla [-10,9]_{\mathtt{d}}$
17	$\bot_{\mathtt{d}}$	$[-10,10]_{\mathtt{i},\mathtt{d}}$	Local: $\bot \nabla \bot_{\mathtt{d}}$
17	$\bot_{\mathtt{d}}$	$[-10,10]_{\mathtt{i},\mathtt{d}}$	Local: $\bot_{\mathtt{d}} \Delta \bot_{\mathtt{d}}$
11	$[-10,9]_{\mathtt{i}}$	$[-10,10]_{\mathtt{i},\mathtt{d}}$	
12	$[-9,10]_{\mathtt{i}}$	$[-10,10]_{\mathtt{i},\mathtt{d}}$	Local: $[10,10]_{\mathtt{i}} \sqcup [-9,10]_{\mathtt{i}}$
13	$\bot_{\mathtt{i}}$	$[-10,10]_{\mathtt{i},\mathtt{d}}$	Protected: $[-10,10]_{\mathtt{i},\mathtt{d}} \nabla [-9,10]_{\mathtt{i}}$
9	$\bot_{\mathtt{i}}$	$[-10,10]_{\mathtt{i},\mathtt{d}}$	Local: $\bot_{\mathtt{i}} \nabla \bot_{\mathtt{i}}$
9	$\bot_{\mathtt{i}}$	$[-10,10]_{\mathtt{i},\mathtt{d}}$	Local: $\bot_{\mathtt{i}} \Delta \bot_{\mathtt{i}}$

6.3. Implementation of the Local View

Having just seen examples of witness invariants (together with their locations) and how an abstract interpreter could be guided by them, we now give a general description of their generation and witness-guided verification. Witness validation is essentially unchanged from the single-threaded case (Section 4.6.2): globals in witness invariants are evaluated in their locations as mandated by the analysis in use.

6.3.1. Witness Generation

All thread-modular analyses from Chapter 5 can be used to generate location-based invariants for all globals at all locations. The abstract value for global $g \in \mathcal{G}$ at location $u \in \mathcal{N}$ can be determined by simulating a read from g, and it can be converted into an invariant expression using asExpr from Section 4.6.3. However, doing so *en masse* for every global at every location leads to an unwieldy witness containing too much duplication. Therefore, to generate useful witnesses, two questions must be answered: (1) where to generate invariants and (2) for which globals?

Globals. At some (to-be-determined) location, interesting invariants would be those which are more precise than the completely unsynchronized invariants, due to some synchronization affecting the location. Depending on the thread-modular analysis, such potentially interesting globals are the following:

- For protection-based analyses (Vojdani's, Protection-Based Reading, Mutex-Meet), all globals (write-)protected by any currently held program mutex.
- For background-lockset–based analyses (Miné's, Lock-Centered Reading, Write-Centered Reading), all globals from all synchronized interferences on any currently held mutex. Note that this ignores the background locksets themselves, which encode thread history (i.e., locking order).

Locations. The above heuristics for the set of interesting globals also determine heuristics for their invariant locations. For example, within a simple critical section (which does not lock or unlock other mutexes), these sets of globals remain constant. Thus, to avoid duplication of invariants, the interesting locations for them are where the sets of globals change, i.e., around synchronization:

After locking. The added mutex provides additional mutual exclusion, which allows imposing stronger invariants, as seen in Example 47.

Before unlocking. The current invariant is published by the unlocking and becomes observable at corresponding locking operations, as seen in Remark 23.

After unlocking. The removed mutex causes less mutual exclusion, which weakens imposable invariants.

Currently, GOBLINT only implements the first of these. Hence, we have not studied their trade-offs, besides what is seen in Example 47 and Remark 23.

6.3.2. Witness-Guided Verification

We now want to apply the unassume operators from Chapter 4 to multi-threaded programs described in Section 5.2.1. Similarly to assume statements (see Section 5.6.2), the inserted $\mathsf{unassume}(e)$ statements can only contain local variables in the invariant expression $e \in \mathcal{I}$. Analogously, unassumes of invariants with globals must be transformed to act on temporary local copies. For example, $\mathsf{unassume}(-10 \leq \mathsf{g} \leq 10)$ must be represented as

$$\begin{aligned}&\mathsf{lock}(m_\mathsf{g}); \mathsf{x} \leftarrow \mathsf{g}; \mathsf{unlock}(m_\mathsf{g});\\ &\mathsf{unassume}(-10 \leq \mathsf{x} \leq 10).\end{aligned}$$

However, this unassume operation only relaxes the bounds of x and not necessarily g as intended. Worse yet, a sufficiently expressive relational domain does not necessarily help in this case, unlike in Example 44 (p. 166):

1. The naïve unassume (Section 4.4.2) fails to relax g:

$$\begin{aligned}[\![\mathsf{unassume}_{\{\mathsf{x}\}}(-10 \leq \mathsf{x} \leq 10)]\!]_1^\sharp \{\mathsf{g} = 0, \mathsf{x} = \mathsf{g}, \mathsf{x} = 0\}\\ = \{\mathsf{g} = 0, -10 \leq \mathsf{x} \leq 10\}.\end{aligned}$$

2. The strengthening-based dual-narrowing unassume (Section 4.4.3) yields:

$$\begin{aligned}[\![\mathsf{unassume}_{\{\mathsf{x}\}}(-10 \leq \mathsf{x} \leq 10)]\!]_2^\sharp \{\mathsf{g} = 0, \mathsf{x} = \mathsf{g}, \mathsf{x} = 0\}\\ = \{\mathsf{g} = 0, \mathsf{x} = \mathsf{g}, \mathsf{x} = 0\} \mathbin{⩲} \{-10 \leq \mathsf{x} \leq 10\},\end{aligned}$$

   which may either be

$$\{\mathsf{g} = 0, -10 \leq \mathsf{x} \leq 10\} \quad \text{or} \quad \{\mathsf{x} = \mathsf{g}, -10 \leq \mathsf{x} \leq 10, -10 \leq \mathsf{g} \leq 10\},$$

   depending on the order in which a strengthening operator adds back constraints.[61] Journault's $⩲_2$ [Jou19, Definition 6.12] enforces the result with $\mathsf{g} = 0$, which is less desirable of the two in this case.
3. Even if the dual-narrowing unassume from the previous point returns the latter result, the relaxed constraint $-10 \leq \mathsf{g} \leq 10$ only remains in the local state and is not side-effected to a flow-insensitive invariant.
4. The relaxed constraint on the global variable in the local state may still be used at a subsequent unlock. However, depending on the analysis, it may not be side-effected if the program itself does not write to g in the same critical section.

[61] With octagon domain constraints, this becomes

$$\{\mathsf{x} \leq 0, -\mathsf{x} \leq 0, \mathsf{g} \leq 0, -\mathsf{g} \leq 0, \mathsf{x}+\mathsf{g} \leq 0, \mathsf{x}-\mathsf{g} \leq 0, -\mathsf{x}+\mathsf{g} \leq 0, -\mathsf{x}-\mathsf{g} \leq 0\} \mathbin{⩲} \{\mathsf{x} \leq 10, -\mathsf{x} \leq 10\}.$$

With many more constraints to strengthen with and factorially many permutations to do so, there are five distinct possible results (exercise for the reader).

All in all, under specific conditions, the unassume may have the intended effect with a relational analysis, although it is not very reliable and predictable.

Therefore, we employ the trick from Remark 21 (p. 167) to introduce an additional assignment from the relaxed auxiliary local variable back to the corresponding global variable. For the above example, this means the following representation:

$$\begin{aligned}&\mathsf{lock}(m_\mathsf{g}); \mathsf{x} \leftarrow \mathsf{g}; \mathsf{unlock}(m_\mathsf{g});\\ &\mathsf{unassume}(-10 \leq \mathsf{x} \leq 10);\\ &\mathsf{lock}(m_\mathsf{g}); \mathsf{g} \leftarrow \mathsf{x}; \mathsf{unlock}(m_\mathsf{g}).\end{aligned}$$

Thus, unassuming a global variable appears as a write to the global variable with the relaxed abstract value.

This representation seamlessly avoids another issue: any intermediate overly imprecise abstract value during the unassume (e.g., a havoced $\top$ value with the naïve unassume) is not side-effected for other threads to observe. Although the analogy with assume statements is strong, Remark 22 does not apply to unassume statements, whose sole purpose is to introduce less precise abstract values than the flow-sensitive invariants currently have, and that can only happen with side effects.

6.4. Evaluation of the Local View

Our experimental goal is to check whether the three pillars for correctness witnesses (Section 1.2) hold for thread-modular witnesses. We use the set of larger benchmarks from Section 5.7.1 for our multi-threaded experiments. In the meantime, the `iowarrior` benchmark was disqualified from SV-COMP due to uninitialized memory access.[62]

We include two thread-modular value analyses in this evaluation: Protection-Based Reading (Section 5.4.1) and Miné's (Section 5.3.2). Both have been implemented within Goblint, thus, all other factors are excluded from the experiments. Besides the choice of thread-modular analysis, we configure Goblint the same as in Section 4.7.2, since we are using the same benchmarks. For the purposes of these experiments, we consider each of the thread-modular analyses as a separate framework: they are fundamentally different as value analyses and compute incompatible abstractions.

Since these benchmarks are unlabeled, we first use each analysis to analyze every benchmark program from scratch and produce a witness. Since Miné's analysis timed out after 900 s on the `nsc` benchmark and did not produce a witness, we exclude `nsc` entirely to avoid bias. On the remaining benchmarks, Protection-Based Reading produces witnesses with a total of 1,232 invariants and Miné's analysis produces witnesses with a total of 1,266 invariants. These witnesses serve as a baseline for what each analysis can prove about the programs. We can then use the validation of these witnesses as a proxy to measure the relative precision of witness-guided analysis.

[62] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1374`

6.4.1. Results

Same-Framework Consistency. We first use the same witnesses both for witness-guided analysis and validation. Protection-Based Reading reproves all but one of its invariants when guided by its own witnesses. Miné's analysis reproves all but 9 of its invariants when guided by its own witnesses. These results are presented on the main diagonal of Table 22. The success rates above 99% indicate that nearly all of the precision is preserved.

Table 22. Witness-guided verification precision in the number of invariants reproven for GOBLINT benchmarks using the local view

Analysis	Analysis invariants	Guidance from			
		Protection		Miné	
Protection	1,232	1,231	(99.92%)	1,230	(99.84%)
Miné	1,266	1,257	(99.29%)	1,257	(99.29%)

Cross-Framework Validation. Protection-Based Reading and Miné's analysis are known and expected to be incomparable, thus, it is unreasonable to expect one to validate all invariants of the other. The goal is to only evaluate the witnesses and not the inherent precision differences of the analyses themselves. Therefore, we substitute cross-framework validation with the novel notion of *cross-framework witness-guided verification*.

We again use the previously-generated witnesses, but this time pass them to the other thread-modular analysis for guidance. That is, the witness being validated (from the same analysis) is different from the witness being used for guidance (from the other analysis), leveraging GOBLINT's ability to handle them independently (Section 4.6.2). As such, we investigate whether guidance from a witness produced by a different approach to concurrency can interfere with the facts that the original analysis could establish about the program.

Protection-Based Reading reproves all but 2 out of its 1,232 invariants when guided by witnesses from Miné's analysis. Miné's analysis reproves all but 9 out of its 1,266 invariants when guided by witnesses from Protection-Based Reading. These results are presented on the antidiagonal of Table 22. The success rates above 99% indicate that nearly all of the precision is still preserved.

Content-Effort Dependence. Figure 18 illustrates the above results via transfer function evaluation counts (evals) as a measure for verification effort. Same-framework validation effort is shown in the subplots on the main diagonal, while the subplots on the antidiagonal show the effort for cross-framework witness-guided verification. Figure 19 shows the same data as a relative change per benchmark. On these benchmarks, witnesses reduce the evals of Protection-Based Reading by 8.5% on average and 25.5% at best, and the evals of Miné's analysis by 9.6% on average and 36.0% at best. The results indicate that witnesses can reduce reverification effort, even when exchanged between different thread-modular analyses.

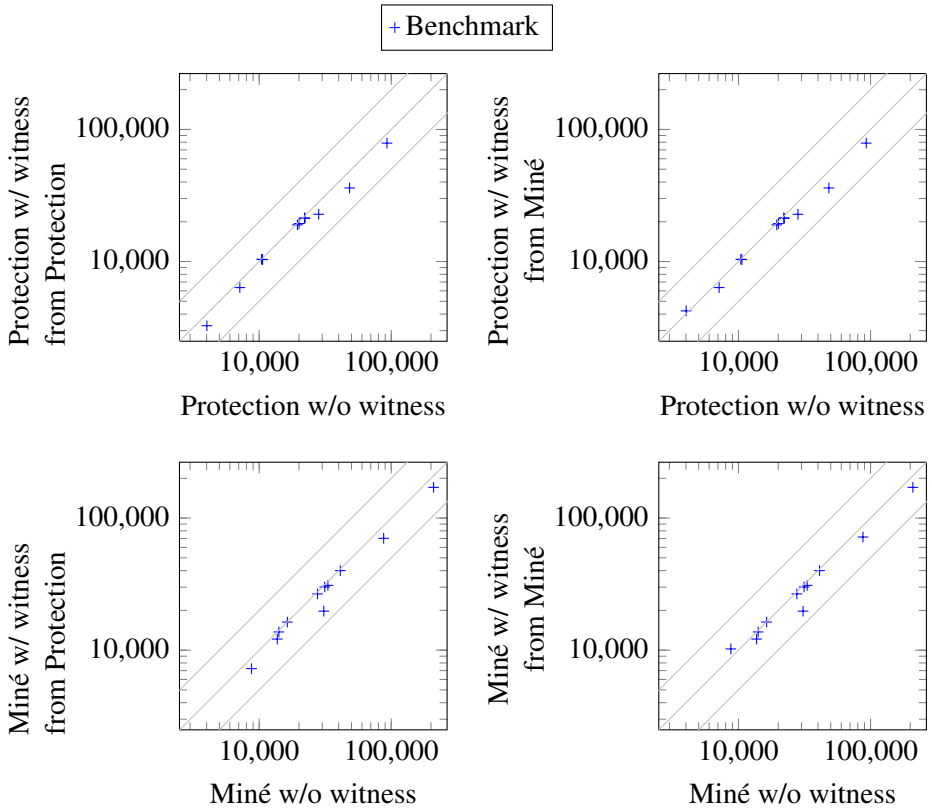


Figure 18. Witness-guided verification effort in evals for GOBLINT benchmarks using the local view; gray lines at powers of 2 from identity

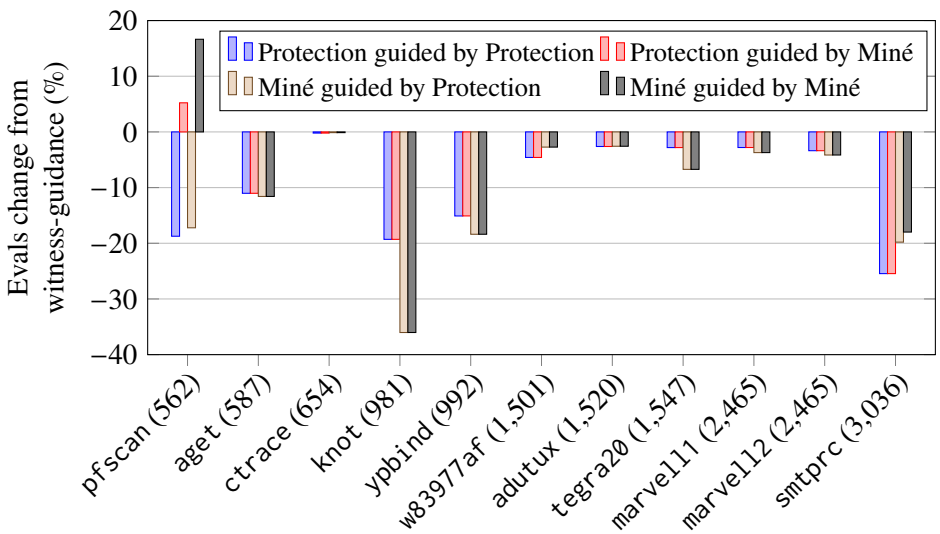


Figure 19. Witness-guided verification effort change in evals for GOBLINT benchmarks (LLoC in parentheses) using the local view

Table 23. Witness-guided verification effort for select GOBLINT benchmarks from Table 14 using the local view with Protection-Based Reading

Program	LLoC	Evals w/o witness	Evals w/ witness (change)	
			Generated	Handcrafted
`pfscan`	562	4,019	3,266 (−19%)	(−30%)
`aget`	587	7,138	6,351 (−11%)	(−41%)
`knot`	981	28,235	22,787 (−19%)	(−28%)
`smtprc`	3,036	48,380	36,067 (−25%)	(−50%)
Average			(−19%)	(−37%)

Comparison with Handcrafted Witnesses. Recall that the performance evaluation of the `unassume` operator in Section 4.7.2 used four of the same benchmarks, but with manually constructed witnesses. In fact, that evaluation relied on witness-guided verification as described in Section 6.3.2, also using the default Protection-Based Reading. Table 23 shows the change in evals due to the witnesses automatically generated here, and compares it with the change due to the handcrafted witnesses (from Table 14). On these benchmarks, the generated witnesses reduce evals by 19% on average and 25% at best, whereas the handcrafted witnesses reduce evals by 37% on average and 50% at best. This suggests that the automatic witness generation described in Section 6.3.1 has room for improvement.

6.4.2. Threats to Validity

Construct Validity. We have evaluated our approach according to the three pillars from Beyer et al. [Bey+16], except applied to witness-guided verification only. In particular, we replaced cross-framework validation with cross-framework witness-guided verification. This allows us to specifically evaluate our thread-modular witness generation and witness-guided verification approaches separately from the underlying thread-modular analyses (evaluated in Section 5.7).

Internal Validity. The evaluations were performed using the BENCHEXEC framework [BLW17] to ensure reliable benchmarking (as described in Section 3.4.6). The GOBLINT-internal evals metric is a deterministic alternative to CPU time (as described in Section 4.7.4).

Since Miné's analysis did not produce a witness for the `nsc` benchmark within the time limit, the benchmark was excluded completely to obtain unbiased results. Otherwise, same-framework consistency of Protection-Based Reading (which includes `nsc`) would have effectively been evaluated on different benchmarks than the rest (which exclude `nsc`), making the results incomparable.

Using witness validation as a substitute for the non-existent benchmark labels does not account for whether the analyses could validate their own witnesses without any witness-guidance. Thus, it may have been impossible to achieve

100% validation for reasons unrelated to witness-guided verification. Thanks to cross-framework witness-guided verification (not validation), the possible effect does not introduce unfairness.

External Validity. Threats regarding the GOBLINT benchmark suite are discussed in Section 5.7.3. The local view of Protection-Based Reading is also evaluated on more multi-threaded SV-COMP benchmarks below in Section 6.8, but only for comparison with the global view.

The cross-framework evaluation considered different analyses in GOBLINT as separate frameworks because unassume operators are not available elsewhere. The analyses have a lot in common though: they are forms of thread-modular abstract interpretation and their implementations share most of GOBLINT. Thus, cross-framework evaluation with a completely distinct verifier and validator may be less successful.

Even within GOBLINT, the evaluation considered only two non-relational approaches from Chapter 5. However, they were chosen to be as distinct as possible: other improved analyses bring the two incompatible approaches closer to each other. Protection-Based Reading was used instead of Vojdani's analysis because no implementation of the latter was available at the time (see Section 5.6). This does not threaten the distinctness of the used approaches because the improvements in Protection-Based Reading are not inspired by Miné's analysis.

We reuse the existing witness format for the multi-threaded case by following particular conventions for invariant locations and expressions. These conventions are specifically chosen to work reasonably well with our thread-modular analyses and unassume operators. As such, they may be less suitable for other verification and validation approaches. Note that the use and the benefits of *location* invariants in witnesses is already rather tool-specific in the single-threaded case.

6.5. The Global View

Invariants on local views of threads can reason about values of variables within the program, but not necessarily about the scheduling and synchronization of threads. The latter can be achieved by encoding relevant scheduling information into *ghost variables* (sometimes referred to as auxiliary variables), which are additional variables introduced to ease the specification and verification of intricate properties. Ghost variables have been long employed for verification of concurrent programs [OG76; Kel76; ABO09], but they can also be useful for sequential programs [Sch97].

We extend the correctness witnesses from Section 2.5.2 with ghost variables and ghost variable updates, inserted into the program to be executed atomically together with existing statements. Invariants of the witness can then refer to both program and ghost variables, and establish properties which are difficult (or impossible) to express without ghosts.

Listing 6. Program for illustrating the global view

(a) Program

```
1  int g = 0;
2  mutex m;

3  void main() {
4    create(t);

5    lock(m);
6      assert(g == 0);
7    unlock(m);
8  }

9  void t() {
10   lock(m);
11     g = 47;
12     g = 0;
13   unlock(m);
14 }
```

(b) Program with ghost instrumentation (highlighted)

```
1  int g = 0;
2  mutex m;
3  int m_locked = 0;

1  void main() {
4    create(t);
5    atomic { assert(m_locked || g == 0); }
1    atomic { lock(m); m_locked = 1; }
6      assert(g == 0);
7    atomic { unlock(m); m_locked = 0; }
8  }

9  void t() {
10   atomic { lock(m); m_locked = 1; }
11     g = 47;
12     g = 0;
13   atomic { unlock(m); m_locked = 0; }
14 }
```

Example 48. The program in Listing 6a has an example of a *resource invariant*: the variable g is always 0, except when thread t holds the mutex m. We cannot state this property with C assertions. Instead, in Listing 6b, we add a ghost variable m_locked and maintain it such that it indicates whether the mutex m is currently held by any thread. The assertion in main states that either some thread holds the mutex m or the global variable g is 0:

$$\texttt{m_locked} \vee \texttt{g} = 0.$$

This is equivalent to saying that if no thread holds m, then g is 0:

$$\neg\texttt{m_locked} \implies \texttt{g} = 0.$$

The latter may be more intuitive, whereas the former can be directly expressed as a C expression. □

Such ghost witnesses should facilitate the exchange of information between tools, which may employ different for concurrent programs. Some use interleaving semantics [Lam79], while others turn to thread-modular semantics [Lam78; Muk+17; Sch+21a]. We describe how a ghost witness for a program yields a ghost-witness–instrumented program, similar to the one in Listing 6b. The validity of a ghost witness, defined via the instrumented program, coincides w.r.t. the interleaving and the local trace semantics (as shown in Section 6.6.2).

Ghost witnesses are naturally suited to express thread-modular invariants computed by abstract interpretation (Section 6.7.1). For the experimental evaluation (Section 6.8), we generate ghost witnesses for concurrent programs in the SV-COMP benchmark suite using GOBLINT, and validate them using the model checker UGEMCUTTER.

6.6. Ghost Witnesses

The formal definition of a ghost witness extends that of a location-based correctness witnesses (Definition 10, p. 36) as follows.

Definition 15 (Ghost witness)**.** A *ghost witness* for a program P is a tuple $(\mathcal{V}_{👻}, U_{👻}, W, P, \Phi)$, where

1. *ghost variables* are given by a set $\mathcal{V}_{👻} \stackrel{\text{def}}{=} \mathcal{X}_{👻} \cup \mathcal{G}_{👻}$, consisting of disjoint subsets of local[63] and global ghost variables, disjoint from the ones in P;
2. *ghost updates* are given by a function $U_{👻} : \mathcal{A} \to \mathcal{S}^*_{👻}$ from arcs to (possibly empty) sequences of assignment statements to ghost variables with right-hand side expressions over variables in P and ghost variables;
3. invariants are given by a function $W : \mathcal{N} \to \mathcal{I}$ from nodes to (possibly trivial) invariant expressions over variables in P and ghost variables;
4. the safety property Φ is assertion safety.[64] ⌟

Before defining the semantics of such witnesses, we first extend our programming language with atomic blocks.

Atomic Blocks. An atomic block is a primitive statement $\mathsf{atomic}\{s_1; \ldots; s_n\}$, where s_1 is an arbitrary primitive statement, and $s_2, \ldots, s_n$ are assignments or assertions. An atomic block must be executed as a whole:

- It may only be executed if s_1 can be executed in the state before the atomic block, i.e., $s_1 \neq \mathsf{assume}(e)$, where e evaluates to false, and $s_1 \neq \mathsf{lock}(m)$, where m is held by another thread.
- The state after the atomic block is the state after the sequence $s_1; \ldots; s_n$.

By considering atomic blocks as primitives, the interleaving semantics in Section 5.2.2 is easily extended with valid transitions involving atomic blocks. Although C does not have atomic blocks as such, SV-COMP has a custom convention that statements between calls to `__VERIFIER_atomic_begin` and `__VERIFIER_atomic_end` are executed as an atomic block,[65] i.e., $\mathsf{atomic}\{s_1; \ldots; s_n\}$ is written as

```
1 __VERIFIER_atomic_begin();
2   s1; ...; sn;
3 __VERIFIER_atomic_end();
```

[63] Here, local ghost variables are only needed for our formalization with local traces, although, in general, they can be useful in their own right.

[64] This is not a major limitation as many properties can be reduced assertion safety (Section 2.6.1).

[65] `https://sv-comp.sosy-lab.org/2025/rules.php#programs`

We now define an instrumentation that inserts ghost variables ($\mathcal{V}_{👻}$), their updates ($U_{👻}$) and invariants (W) from the ghost witness into the program. Atomic blocks are used to force ghost updates to happen together with statements in the program, as seen in Listing 6b. In Example 48, this is crucial to have the ghost variable `m_locked` precisely reflect the locking state of the mutex `m` at all times.

Definition 16 (Ghost-witness–instrumented program)**.** Given a ghost witness $(\mathcal{V}_{👻}, U_{👻}, W, P, \Phi)$ for a program P, the *ghost-witness–instrumented program* $P_{👻}$ is obtained from P by augmenting it in three ways:

1. Ghost variables $\mathcal{V}_{👻}$ are added to the set of program variables of P.
2. For every node $n \in \mathcal{N}$ where $W\, n \neq e_{\mathsf{true}}$, the statement $\mathsf{atomic}\{\mathsf{assert}(W\, n)\}$ is inserted at that location to assert the invariant. In particular, if $W\, n$ contains global (ghost) variables, then the necessary temporary local (ghost) copies are introduced and the global reads are within the same atomic section.
3. For every arc $e = (u, s, v) \in \mathcal{A}$ where $U_{👻}\, e$ is non-empty, the arc is replaced with $(u, \mathsf{atomic}\{s; U_{👻}\, e\}, v)$ to perform the ghost updates atomically after the existing primitive s. ⌟

The semantics of a ghost witness for a program is defined via the corresponding ghost-witness–instrumented program.

6.6.1. Validity w.r.t. Interleaving Semantics

Definition 17 (Valid ghost witness w.r.t. interleaving semantics)**.** A ghost witness for a program P is *valid w.r.t. the interleaving semantics* if and only if the ghost-witness–instrumented program $P_{👻}$ is assertion-safe w.r.t. the interleaving semantics. ⌟

We show that if the original program is assertion-unsafe, then a ghost-witness–instrumented program is assertion-unsafe as well; i.e., assertion safety violations in the original program are preserved in the ghost-witness–instrumented program.

Theorem 9 (Preservation of violation)**.** *Let $(\mathcal{V}_{👻}, U_{👻}, W, P, \Phi)$ be a ghost witness. If there is an interleaving of P that violates an assertion a at arc e taken by thread t, then the corresponding interleaving of $P_{👻}$ violates the assertion a at the corresponding arc taken by thread t.*

Proof. A proof can be found in the original paper [Erh+25, Theorem 2]. To obtain from an interleaving of P an interleaving of $P_{👻}$, the main idea is to extend the states with ghost variables, add arcs introduced for asserting invariants, and replace each step taken on an arc with a step on the instrumented arc which includes the ghost updates. As the inserted invariant assertions and ghost updates cannot modify non-ghost variables, the initial and the extended interleaving correspond w.r.t. those variables. ■

Suppose a verifier outputs that a program P is assertion-safe and produces a ghost witness. By validating the witness, the validator should be able to conclude

that P is assertion-safe. This follows from the contrapositive of the previous theorem and Definition 17.

Corollary 2. *Let P be a program. If a ghost witness for P is valid, then P is assertion-safe.*

On the other hand, we show that if an assertion in the original program is safe, then the same assertion in the ghost-witness–instrumented program is safe as well; i.e., assertion safety of the original program is preserved by the witness instrumentation. The following theorem states this by contrapositive.

Theorem 10 (Preservation of correctness)**.** *Let $(\mathcal{V}_{👻}, U_{👻}, W, P, \Phi)$ be a ghost witness. If there is an interleaving of $P_{👻}$ that violates an assertion a at an arc corresponding to an arc e* in the original program *P, then there is an interleaving of P that violates the assertion a at the arc e.*

Proof. A proof sketch can be found in the extended version of the original paper [Erh+24b, Appendix B.1]. ■

By Definition 17, this carries over to witnesses.

Corollary 3. *Let P be a program. If a ghost witness for P is invalid because an assertion from P can be violated in $P_{👻}$, then P is assertion-unsafe.*

6.6.2. Validity w.r.t. Local Trace Semantics

We introduced atomic blocks into our language to instrument programs with ghost witnesses. While this notion of atomics is convenient for the interleaving semantics, it is less clear for the local trace semantics, where communication between threads only happens via observable and observing events (Section 5.2.3). Thus, we show how atomic blocks can be encoded as critical sections (using lock and unlock statements). This encoding is used to define the validity of ghost witnesses w.r.t. the local trace semantics, which turns out to be equivalent to the interleaving semantics case.

Atomic Blocks. Critical sections allow encoding atomic blocks in a way that preserves assertion safety, as has been noted in the literature [Lom77; Faj+16, Remark 3.52] and used for practical implementations of atomic blocks [McC+06].

We consider a procedure that takes programs P with atomic blocks and yields programs where these are encoded as critical sections using the atomicity mutexes $\mathcal{M}_\mathcal{G}$. In the resulting program, all statements accessing a global variable g have to be embedded into critical sections protected by the mutex m_g. Specifically, each atomic block is encoded as a critical section, which is protected by all mutexes $m_g \in \mathcal{M}_\mathcal{G}$ if g is read or written in the atomic block. Their lock operations are inserted following some total order on $\mathcal{M}_\mathcal{G}$. In Listing 7, one of the atomic blocks of the introductory example from Listing 6b is encoded via critical sections.

Theorem 11. *Let P be a program with atomics and P' the corresponding program with mutexes $\mathcal{M}_\mathcal{G}$ instead. The program P is assertion-safe w.r.t. the interleaving semantics if and only if P' is assertion-safe w.r.t. the interleaving semantics.*

Listing 7. Example of encoding of an atomic block as a critical section

```
1 atomic {                          1 lock(m_m_locked);
2   lock(m);                        2   lock(m);
3   m_locked = 1;        ⟼         3   m_locked = 1;
4 }                                 4 unlock(m_m_locked);
```

Proof. A proof-sketch for the direction that if P' is assertion-safe w.r.t. the interleaving semantics, also P is assertion-safe w.r.t. the interleaving semantics, can be found in the extended version of the original paper [Erh+24b]. ■

We remark that such an encoding of atomic blocks via mutexes may introduce deadlocks, which do not unduly restrict the set of reachable states.

Remark 25 (Assumptions on protecting mutexes). This encoding of atomic blocks into mutexes $\mathcal{M}_{\mathcal{G}}$ violates the assumption from Section 5.3.3 for Vojdani's analysis that $m_{g'} \notin \mathcal{M}_{\text{rw}}[g]$ for every $g' \neq g$. The same goes for the analogous assumption about $\mathcal{M}_{\text{w}}[g]$ from Section 5.4.1 for Protection-Based Reading. Since this only concerns the analyses, not the semantics, it will not be a problem for us because we will not apply the analyses to such encoded programs. □

Validity. A witness validator, therefore, may choose to take either view: analyzing the ghost-witness–instrumented program w.r.t. a semantics that considers atomic blocks or w.r.t. a semantics that encodes atomic blocks as critical sections. Using this encoding, we can define the validity of a ghost witness w.r.t. the local trace semantics.

Definition 18 (Valid ghost witness w.r.t. local trace semantics)**.** A ghost witness for a program P, resulting in the ghost-witness–instrumented program $P_{👻}$, is *valid w.r.t. the local trace semantics* if and only if the program $P_{👻}$ where atomics have been replaced with mutexes $\mathcal{M}_{\mathcal{G}}$ is assertion-safe w.r.t. the local trace semantics. ⌟

Finally, we can relate this notion back to the validity of ghost witnesses in the interleaving semantics. With the definition of a valid ghost witness w.r.t. the local trace semantics referring back to assertion safety, and by encoding atomicity in ghost-witness–instrumented programs via critical sections, we can lift the agreement of the two semantics on assertion safety to an agreement on the validity of ghost witnesses.

Theorem 12. *Let P be a program. A ghost witness for P is valid w.r.t. the interleaving semantics if and only if it is valid w.r.t. the local trace semantics.*

Proof. By Definition 17, the ghost witness is valid w.r.t. the interleaving semantics if and only if the ghost-witness–instrumented program $P_{👻}$ is assertion-safe. According to Theorem 11, the program $P_{👻}$ is assertion-safe w.r.t. the interleaving semantics if and only if the program $P_{👻}$ where atomics have been replaced with mutexes $\mathcal{M}_{\mathcal{G}}$ is assertion-safe w.r.t. the interleaving semantics. According to Corollary 1 (p. 130), the latter program is assertion-safe w.r.t. the interleaving semantics if and only if it is assertion-safe w.r.t. the local trace semantics. With Definition 18, the statement follows. ■

6.6.3. YAML Format

We propose here an extension of the YAML format for correctness witnesses (Section 2.5.3) with ghost variables. This defines a textual format for ghost witnesses $(\mathcal{V}_{\text{👻}}, U_{\text{👻}}, W, P, \Phi)$ from Definition 15.

Syntax. A YAML ghost correctness witness is a sequence of two entries. The first one is the `invariant_set` described in Section 2.5.3. The second one is an object with the following fields:

- **`entry_type`.** The constant string `ghost_instrumentation`.
- **`metadata`.** An object which contains the same metadata as described in Section 2.5.3 for `invariant_set`.
- **`content`.** An object with the following fields:
  - **`ghost_variables`.** A sequence of *ghost variable declaration* objects with the following fields:
    - **`name`.** A string indicating the name of the ghost variable.
    - **`type`.** A string indicating the type of the ghost variable.
    - **`scope`.** The constant string `global`.[66]
    - **`initial`.** An object with the following fields:
      - **`value`.** A string of the initial value of the ghost variable.
      - **`format`.** The constant string `c_expression`.
  - **`ghost_updates`.** A sequence of *ghost variable update* objects with the following fields:
    - **`location`.** An object with the same fields as described in Section 2.5.3 for invariants.
    - **`updates`.** A sequence of update objects for the location with the following fields:
      - **`variable`.** A string indicating the name of the ghost variable.
      - **`value`.** A string of the value assigned to the ghost variable.
      - **`format`.** The constant string `c_expression`.

Listing 8 shows the ghost witness for the program instrumentation in Listing 6.

Semantics. Ghost variables must have unique names, distinct from those appearing in the program. Both the initial values and the updated values of ghost variables must be side-effect–free C expressions. The initialization expression may refer to global variables of the program, whereas the update expression may also refer to ghost variables. Additionally, invariant expressions in the `invariant_set` may also refer to ghost variables.

[66] Since local ghost variables are only needed for our formalization with local traces, we currently omit them from the format. However, we include the `scope` field for easy future generalization.

Listing 8. YAML ghost witness for the program in Listing 6

```yaml
- entry_type: invariant_set
  metadata: # omitted
  content:
  - invariant:
      type: location_invariant
      location: { file_name: listing-6.c, line: 5 }
      value: m_locked || g == 0
      format: c_expression
- entry_type: ghost_instrumentation
  metadata: # omitted
  content:
    ghost_variables:
    - name: m_locked
      type: int
      scope: global
      initial:
        value: 0
        format: c_expression
    ghost_updates:
    - location: { file_name: listing-6.c, line: 5 }
      updates:
      - variable: m_locked
        value: 1
        format: c_expression
    - location: { file_name: listing-6.c, line: 7 }
      updates:
      - variable: m_locked
        value: 0
        format: c_expression
    - location: { file_name: listing-6.c, line: 10 }
      updates:
      - variable: m_locked
        value: 1
        format: c_expression
    - location: { file_name: listing-6.c, line: 13 }
      updates:
      - variable: m_locked
        value: 0
        format: c_expression
```

Ghost Update Locations. Definition 15 allows the $U_{\text{🔔}}$ component to attach ghost updates to any statement. The instrumentation of Definition 16 wraps these into an atomic block together with the program statement. This makes the statement atomic in the instrumented program, even when it was not originally; thus, undesirably changing the behavior of the program. Therefore, to avoid such issues, we only allow ghost updates to occur at the following statements:

- For calls to functions from the PTHREADS library, the ghost update is performed after evaluating the arguments of the function call, atomically together with:
  - the thread creation for `pthread_create`;
  - the locking/unlocking of the mutex, resp., for `pthread_mutex_lock`/ `pthread_mutex_unlock` (and analogously for read-write locks);
  - the locking of the mutex for `pthread_cond_wait`.
- For assignments, the ghost update is performed after evaluating the right-hand side of the assignment, atomically together with the actual write itself.[67]
- For calls to `__VERIFIER_atomic_begin`/`__VERIFIER_atomic_end` (used for atomic blocks in SV-COMP), the ghost update is performed after/before the function call, atomically together with the atomic block (i.e., all statements between these calls).

6.7. Implementation of the Global View

Having defined ghost witnesses and extended the YAML witness format, we now describe how specific analyses can generate and validate such witnesses.

6.7.1. Witness Generation

In this section, we describe how ghost witnesses, such as the one in Listing 8, can be generated using the thread-modular analyses from Chapter 5.

Ghost Variables. Generalizing the ghost instrumentation in Listing 6b, we introduce for each mutex $m \in \mathcal{M} \setminus \mathcal{M}_{\mathcal{G}}$ a boolean ghost variable m_{locked}, indicating whether any thread has locked m. The ghost variable is initialized to false and corresponding ghost updates are added to the witness: every $\mathsf{lock}(m)$ is instrumented with $m_{\text{locked}} \leftarrow \mathsf{true}$ and every $\mathsf{unlock}(m)$ with $m_{\text{locked}} \leftarrow \mathsf{false}$. Such ghost variables are well suited for describing the results of protection-based analyses.

Ghost Updates. In C, mutexes are not locked and unlocked directly, but rather through pointers, which poses additional hazards. Namely, the lock or unlock operation at a particular location may use a different mutex at different times.

[67] This may reduce the possible interleavings, but for race-free programs the behavior is equivalent.

Thus, the ghost instrumentation described above cannot be applied accurately and correctly. Additional analysis can be performed to avoid such issues:

1. For each lock/unlock operation, determine the set of mutexes that may be locked/unlocked.
2. Exclude a mutex from ghost instrumentation altogether if it is included in any of those sets which have more than one element.

Therefore, we restrict ourselves to cases where the instrumentation will be exact.

Analyses. We now describe the witness invariants corresponding to various thread-modular analyses. Let $\eta : \mathcal{X} \to \mathbb{D}$ be the constraint system solution, which corresponds to the analysis results. Recall the functions $\text{asExpr}_{\mathbb{R}}$ and $\text{asExpr}^g_{\mathbb{V}}$ from Section 4.6.3 for converting relational and non-relational abstractions into invariant expressions. The generated witness invariants vary depending on the thread-modular analysis because each one computes a different abstraction. For protection-based analyses, these are constructed as follows:

Baseline. These non-relational analyses compute for each global $g \in \mathcal{G}$ an invariant $[g]$ abstracting all of its possible values, ignoring mutual exclusion. The corresponding witness invariant is $\text{asExpr}^g_{\mathbb{V}}(\eta\,[g])$.

Mutex-Meet. This relational analysis computes for each mutex $m \in \mathcal{M}$ a mutex invariant $[m]$ between jointly protected globals $\mathcal{G}_{\text{w}}[m]$. In particular, for each global $g \in \mathcal{G}$, the mutex invariant $[m_g]$ directly yields an analogous unprotected witness invariant $\text{asExpr}_{\mathbb{R}}(\eta\,[m_g])$ on that global only. For each mutex $m \in \mathcal{M} \setminus \mathcal{M}_{\mathcal{G}}$, the mutex invariant $[m]$ holds whenever no thread holds the mutex. The corresponding witness invariant is

$$\neg m_{\text{locked}} \implies \text{asExpr}_{\mathbb{R}}(\eta\,[m]) \quad \iff \quad m_{\text{locked}} \vee \text{asExpr}_{\mathbb{R}}(\eta\,[m]),$$

where $\implies$ is the implication operator in the invariant language $\mathcal{I}$ (if available). For example, the ghost assertion in Listing 6b is of this kind.

Vojdani's analysis. This non-relational analysis computes for each global $g \in \mathcal{G}$ an invariant $[g]$ which holds whenever no thread holds a program mutex from $\mathcal{M}_{\text{rw}}[g]$. The corresponding witness invariant is

$$\left(\bigwedge_{m \in \mathcal{M}_{\text{rw}}[g] \setminus \{m_g\}} \neg m_{\text{locked}}\right) \implies \text{asExpr}^g_{\mathbb{V}}(\eta\,[g]) \iff \left(\bigvee_{m \in \mathcal{M}_{\text{rw}}[g] \setminus \{m_g\}} m_{\text{locked}}\right) \vee \text{asExpr}^g_{\mathbb{V}}(\eta\,[g]).$$

This naturally covers the case when g is unprotected by program mutexes, i.e., $\mathcal{M}_{\text{rw}}[g] = \{m_g\}$: the witness invariant degrades to $\text{asExpr}^g_{\mathbb{V}}(\eta\,[g])$, as for the baseline.

Listing 9. Program with a multiply-protected global variable

```
1  int g = 0;
2  mutex m1, m2;

3  void main() {
4    create(t);
5    // assert(g == 0);
6  }

7  void t() {
8    lock(m1);
9      lock(m2);
10       g = 1;
11     unlock(m2);
12     lock(m2);
13       g = 0;
14     unlock(m2);
15   unlock(m1);
16 }
```

This witness invariant may appear unnecessarily weak when there are multiple protecting program mutexes, all of which are held at every access of g. However, consider the program in Listing 9, where $\mathcal{M}_{\mathrm{rw}}[\mathtt{g}] = \{\mathtt{m1}, \mathtt{m2}, m_{\mathtt{g}}\}$, with the flow-insensitive invariant $(\mathtt{m1}_{\mathrm{locked}} \wedge \mathtt{m2}_{\mathrm{locked}}) \vee \mathtt{g} = 0$. This would be violated at the commented assertion, while thread `t` has temporarily unlocked `m2`, since `g = 1`. That is because $\mathcal{M}_{\mathrm{rw}}[\mathtt{g}]$ does not account for hypothetical reads of `g` by the witness invariant at locations with fewer mutexes, like the commented assertion.

Protection-Based Reading. This non-relational analysis computes for each global $g \in \mathcal{G}$ an unprotected invariant $[g]'$ and a protected invariant $[g]$ which holds whenever no thread holds a program mutex from $\mathcal{M}_{\mathrm{w}}[g]$. For each global $g \in \mathcal{G}$, the unprotected invariant directly yields a witness invariant $\mathrm{asExpr}^{g}_{\mathbb{V}}(\eta\,[g]')$, as for the baseline. For each global $g \in \mathcal{G}$, if g is write-protected by a program mutex, i.e., $\mathcal{M}_{\mathrm{w}}[g] \neq \{m_g\}$, then the corresponding protected witness invariant is

$$\left(\bigwedge_{m \in \mathcal{M}_{\mathrm{w}}[g] \setminus \{m_g\}} \neg m_{\mathrm{locked}}\right) \Rightarrow \mathrm{asExpr}^{g}_{\mathbb{V}}(\eta\,[g]) \iff \left(\bigvee_{m \in \mathcal{M}_{\mathrm{w}}[g] \setminus \{m_g\}} m_{\mathrm{locked}}\right) \vee \mathrm{asExpr}^{g}_{\mathbb{V}}(\eta\,[g]).$$

Unlike the witness invariants for Vojdani's analysis, the side condition $\mathcal{M}_{\mathrm{w}}[g] \neq \{m_g\}$ is necessary for soundness, as $[g]$ may be stale for such globals (cf. Footnote 52, p. 144). On the other hand, if $[g] = [g]'$, then the protected witness invariant may be omitted as it is subsumed by the unprotected one.

In C, if the ghost variable m_{locked} is not available, because it may be used ambiguously, then the entire witness invariant including it is omitted. We do not provide

witness invariant constructions for background-lockset–based analyses (Miné's, Lock-Centered Reading, Write-Centered Reading), because background locksets encode thread history (i.e., locking order) which cannot be described with such intuitive ghost variables.

Multi-threaded Mode. As described in Section 5.6.1, GOBLINT distinguishes the single-threaded phase of the program from the multi-threaded phase. Since thread-modular analyses are only used for the latter, their flow-insensitive invariants (in the respective unknowns) only apply to that phase. To soundly encode the invariants into a witness, we introduce another boolean ghost variable `multithreaded`, indicating whether the program has become multi-threaded. It is initialized to false and all thread creation statements that may potentially create the first non-main thread are instrumented with `multithreaded` $\leftarrow$ true. Thus, every witness invariant e from above is actually added as

$$\texttt{multithreaded} \Rightarrow e \quad \Longleftrightarrow \quad \neg\texttt{multithreaded} \vee e.$$

Locations. The analyses and their computed invariants are flow-insensitive on global variables, so the invariants should be valid at all program locations. The (ghost) witness formalism and format, however, do not support location-less invariants. Furthermore, repeating the flow-insensitive invariants at every location is costly for validators; instead, we add such flow-insensitive invariants after each thread create in the program.

6.7.2. Witness Validation

Abstract Interpretation. Although a ghost witness constitutes program instrumentation, validating the witness by verifying the transformed program using thread-modular abstract interpretation poses some challenges:

- Section 6.6.2 outlines how the atomic sections in ghost-witness–instrumented programs can be handled in the local traces formalism, used by the analyses in Chapter 5. Although sufficient in theory, it does not necessarily work in practice, where variables accessed by an atomic section are not statically known (e.g., due to pointers).
- Ghost variables may encode non–thread-modular features which would be out of reach for the thread-modular abstractions used by the analysis. In general, ghost variables can encode interleavings, while thread-modular analyses aim to give up interleaving precision.

We do not tackle these problems here but leave them as future work.

Model Checking. We implemented ghost witness validation in the software model checking tool ULTIMATE GEMCUTTER (UGEMCUTTER for short) [Klu+22; FKP22], which is part of the ULTIMATE tool family.

UGEMCUTTER is based on an interleaving semantics (Section 5.2.2). It uses a *commutativity relation* to identify pairs of actions (of different threads) whose

relative order does not affect program correctness. In its analysis, UGemCutter groups interleavings into equivalence classes based on the induced *Mazurkiewicz equivalence* [Maz86], i.e., it considers interleavings as equivalent if they only differ in the order of commuting actions. UGemCutter proves correctness for one representative per equivalence class, using the *trace abstraction* algorithm [HHP09], a scheme for counterexample-guided abstraction refinement (CEGAR). If the proof for the representatives succeeds, then UGemCutter soundly concludes the correctness of the entire program.

To validate ghost witnesses, UGemCutter instruments the given program with the ghost code and the invariants from the witness (a la Definition 16), and applies its usual verification algorithm to the instrumented program. In the witness-only validation mode (Section 2.5.2), UGemCutter only checks if the assertions of the instrumented program that correspond to witness invariants can be violated, while ignoring the assertions already present in the original program.

6.8. Evaluation of the Global View

Ghost witnesses, expressed in the format described in the Section 6.6.3, can be exchanged with and validated by other verification tools to confirm the results of the verification using tools with a significant technological and/or conceptual difference; and, thereby, increase trust in the verification tool or potentially uncover bugs in the verification. Complementary to the theoretical equivalence results of Theorem 12, we empirically evaluate the impact of ghost witness validation regarding the pillars from Section 1.2, using the following questions:

Cross-framework validation. Can ghost witnesses generated by a verifier be (witness-only) validated by a different tool based on different semantics, analysis approaches and technological foundations?

Content-effort dependence. What is the overhead of ghost witness validation? Can witnesses help to accelerate the analysis, compared to verification without a witness?

Additionally, we compare the local and global views in practice. Finally, we report on how the evaluation led to improvement of the tools involved (Section 6.8.2).

Benchmarks. For our evaluation, we used Goblint to analyze the programs in the *ConcurrencySafety-Main* category of SV-COMP 2024 [Bey24] and generate correctness witnesses. We configured Goblint the same as in SV-COMP 2025 (Section 3.4.5), except witness invariants are only generated for global variables. For two of the analyses, Protection-Based Reading and Mutex-Meet, we generated witnesses with and without ghost variables (as described in Sections 6.7.1 and 6.3.1, respectively), yielding four witness benchmark sets. For Mutex-Meet, relational analysis using octagons was unconditionally enabled. We excluded witnesses that do not contain any (non-trivial) invariants.

Note that, although Goblint cannot prove the incorrectness of programs, it is still able to generate valid invariants for incorrect programs (though, they

are, naturally, insufficient to prove correctness). Hence, the generated witness benchmark sets include witnesses for incorrect programs, where the invariants reflect information that GOBLINT has derived about the program, and which can be checked by witness-only validation.

Experimental Setup. We (witness-only) validated these witnesses using UGEMCUTTER. The validation was executed on an AMD RYZEN THREADRIPPER 3970X processor at 3.7 GHz with a time limit of 900 s, a memory limit of 16 GB and a CPU core limit of 2.

6.8.1. Results

Cross-Framework Validation. Table 24 shows the number of witnesses in each witness benchmark set, and how many of them are confirmed (resp. rejected) by UGEMCUTTER in the witness-only validation mode. Despite the significant differences between GOBLINT and UGEMCUTTER, the communication via the ghost witness format is successful. The analyzed witnesses contain non-trivial information: on average, ghost witnesses in the set Protection$_{\text{👻}}$ (resp. Mutex-Meet$_{\text{👻}}$) contain 44.1 (resp. 43.9) invariants and 3.8 (resp. 3.6) ghost updates. For witnesses that are not confirmed, UGEMCUTTER either runs out of resources (time or memory), or errors due to unsupported features in the program or the witness.

Table 24. Cross-framework witness-only validation in the number of witnesses using the global and the local view

(a) Correct programs

	Witnesses from							
UGEMCUTTER	Protection$_{\text{👻}}$		Mutex-Meet$_{\text{👻}}$		Protection		Mutex-Meet	
Confirmed	171	(94%)	193	(89%)	122	(77%)	167	(73%)
Rejected	0	(0%)	0	(0%)	0	(0%)	0	(0%)
Out of resources	5	(3%)	19	(9%)	34	(21%)	56	(24%)
Error	5	(3%)	5	(2%)	3	(2%)	7	(3%)
Total	181		217		159		230	

(b) Incorrect programs

	Witnesses from							
UGEMCUTTER	Protection$_{\text{👻}}$		Mutex-Meet$_{\text{👻}}$		Protection		Mutex-Meet	
Confirmed	192	(68%)	164	(57%)	224	(76%)	130	(43%)
Rejected	0	(0%)	0	(0%)	0	(0%)	0	(0%)
Out of resources	85	(30%)	117	(41%)	68	(23%)	163	(54%)
Error	5	(2%)	7	(2%)	3	(1%)	7	(2%)
Total	282		288		295		300	

Table 25. Cross-framework witness validation in the number of witnesses and total effort in CPU time (h:min:s) using the global and the local view

	Witnesses from							
	Protection$_{👻}$		Mutex-Meet$_{👻}$		Protection		Mutex-Meet	
Process	#	Time	#	Time	#	Time	#	Time
Verification[a]	120	1:07:27	131	1:18:29	101	44:34	118	1:07:43
Validation	119	1:31:33	123	1:29:27	100	2:01:29	110	2:37:12
Change		+35.7%		+14.0%		+172.6%		+132.1%

[a] Limited to tasks with a witness in the corresponding set.

Content-Effort Dependence. Table 25 shows a summary of the results of running UGemCutter in full witness validation mode (the mode used in SV-COMP), where the witness invariants must be confirmed and correctness of the program must be proven. As we are working with correctness witnesses, we only consider correct programs for the validation. Figure 20 shows the results in more detail. For Goblint witnesses validated by UGemCutter, witness validation generally incurs an overhead, rather than accelerating the verification. However, for most programs verified by UGemCutter, the corresponding witnesses can also be validated with a moderate slowdown, but a speedup can be observed on some larger benchmarks. In one case for Mutex-Meet$_{👻}$ and one case for Protection$_{👻}$,[68] validation of a ghost witness set succeeds (in < 250 s), while the verification of the corresponding program times out.

Comparison with the Local View. Witness generation for the local and the global view are very different. Thus, the number of invariants, their locations and the invariant expressions themselves are incomparable between the two kinds of witnesses. Non-ghost witnesses in the set Protection (resp. Mutex-Meet) contain on average 179 (resp. 749.9) invariant entries, which is significantly higher than for the ghost witnesses. This is expected because the non-ghost witnesses repeat an invariant at every critical section, whereas the ghost witnesses can express (roughly) the same using a flow-insensitive invariant (at one location).

Table 24a shows that the confirmation rate for non-ghost witnesses is lower than for ghost witnesses, largely due to more out of resources validation results. The greater overhead of non-ghost–witness validation is even more apparent in Table 25 and Fig. 20. The slowdown compared to ghost witness validation is explained by the difference in invariant counts.

Both the local and the global witness generation approaches try to express as much of their findings as possible using witness invariants. Therefore, the evaluation provides practical evidence that ghost witnesses are better suited for expressing thread-modular abstractions.

[68] Tasks `weaver/popl20-two-queue.wvr` and `weaver/popl20-simple-array-sum.wvr`

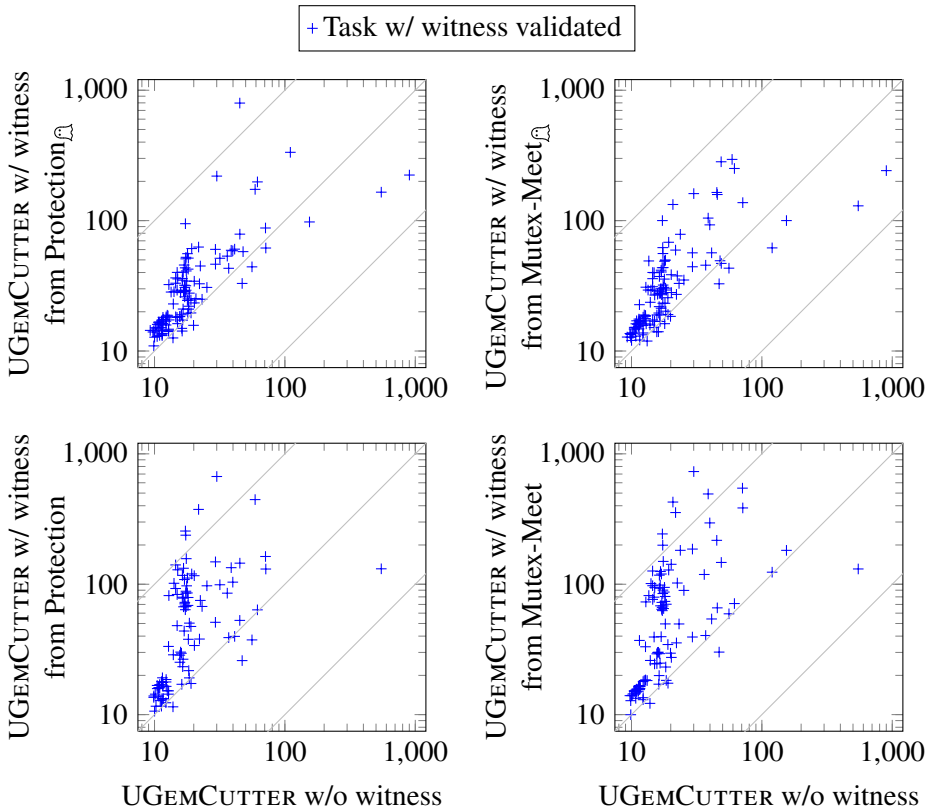


Figure 20. Cross-framework witness validation effort in CPU time (s) using the global and the local view; gray lines at powers of 10 from identity

6.8.2. Bugs

Over the duration of this work, its implementation and (re-)evaluation revealed 29 bugs in GOBLINT and UGEMCUTTER, and led to their fixing in most cases. These bugs already existed prior to and independently of this work. Crucially, they were revealed on a subset of the very same SV-COMP benchmarks that have been used to test and evaluate these tools for years. A full overview of these bugs is provided in the extended version of the original paper [Erh+24b, Appendix E].

In particular, four soundness-critical bugs were discovered in the verification performed by GOBLINT, and 13 bugs in GOBLINT's witness production:

- Three soundness issues in the relational value analysis, e.g., regarding atomic blocks with `__VERIFIER__atomic_*` (Section 6.6).
- Two issues in syntactic loop unrolling, e.g., sound handling of `continue`.
- Four issues in witness generation concerning locations of invariants related to complex loops, e.g., for `do-while` and `for` loops.
- Two issues in witness generation concerning locations of invariants related to short-circuiting logical expressions and multiple/compound initializers.

- Three issues in witness generation concerning variables in invariant expressions, e.g., scope of local variables and not outputting GOBLINT-internal variables.
- Two issues in witness generation concerning trivial invariants, e.g., for booleans (e.g., `x == 0 || x == 1` when `x` is of type `_Bool`) and for the congruence domain (e.g., `x % 1 == 0`).

Their fixes were included in the version of GOBLINT participating in SV-COMP 2025 (Section 3.4.5).

In ULTIMATE, five issues concerned ACSL specification support (which is reused as part of the ghost witness instrumentation), two bugs concerned the translation of C code to ULTIMATE's internal representation, one bug concerned the internal representation itself, and one bug was specific to witness validation. Notably, the bugs regarding ACSL support and translation of C code are soundness-critical, and cannot be caught by correctness checks inside ULTIMATE, which operate on the internal representation.

Thus, the ghost witness exchange and validation significantly improved both tools, a benefit also observed by Beyer et al. [Bey+16, Footnote 8, p. 333].

6.8.3. Threats to Validity

Construct Validity. We have evaluated our approach according to two of the three pillars from Beyer et al. [Bey+16]. Same-framework evaluation was not possible since neither tool implemented both ghost witness generation and validation. In recent follow-up work, Schüssele et al. [Sch+26] have developed and evaluated a method for turning interleaving-based proofs from UAUTOMIZER into thread-modular proofs with one ghost variable.

We replaced cross-framework validation with *cross-framework witness-only validation* because the thread-modular analyses of GOBLINT are insufficient to completely verify many of the programs. This allows us to specifically evaluate the ghost witnesses separately from the precision of GOBLINT. It also yields larger benchmark sets of witnesses by including correctness witnesses for incorrect programs, which cannot be fully valid.

Internal Validity. The evaluations were performed using the BENCHEXEC framework [BLW17] to ensure reliable benchmarking (as described in Section 3.4.6).

Witnesses without any non-trivial invariants are trivially confirmed by witness-only validation. If not removed, they would have artificially inflated the confirmation rates in Table 24. Full validation of such witnesses is equivalent to from-scratch verification of the corresponding program. Thus, they would have also biased the results for content-effort dependence.

External Validity. Threats regarding the SV-COMP benchmark suite are discussed in Section 3.4.6. By excluding witnesses containing only trivial invariants, the eval-

uation may be biased towards programs which rely on mutexes for synchronization and which are well suited for the thread-modular analyses in GOBLINT.

The evaluations only considered one tool for ghost witness generation and one for validation. For additional diversity, we considered two thread-modular analyses for witness generation: one non-relational and one relational. The recent work of Schüssele et al. [Sch+26] proposes and evaluates ghost witness generation using UAUTOMIZER.

The ghost witness validator UGEMCUTTER is generic: it supports arbitrary use of ghost variables (within the rules of Section 6.6.3). However, the witnesses generated by GOBLINT only use two kinds of ghost variables: for mutex locking states and for multi-threadedness. Different kinds of ghost variables used in different ways may be more or less beneficial for the validator. Furthermore, the benefits may be different for validators based on other verification approaches.

6.9. Related Work

Ghost variables are crucial for the (relative) completeness of the proof systems of Owicki and Gries [OG76], and Lamport [Lam77]. The usefulness of these and similar approaches has been shown in case studies [Gri77; Dij78; Lub84; SD23]. Nipkow and Nieto [NN99] formalize the Owicki-Gries method in ISABELLE/HOL and extend the method to parametric programs [Nie01; Nie02]. Recent works extend the Owicki-Gries method to weak memory [LV15; Dal+20; Dal+22; Wri+23] and persistent memory [RLV20; Bil+22].

We showed that our ghost instrumentation preserves the assertion safety of the original program. Filliâtre, Gondelman, and Paskevich [FGP16] present an ML-style programming language and ensure non-interference via the type system, allowing a bisimulation proof of ghost code erasure. For concurrent code, Zhang et al. [Zha+12] present a structural approach to establish erasure, and Schmaltz [Sch12] gives bisimulation proofs for a significant portion of low-level C with ghost updates.

Our approach to generating ghost witnesses is reminiscent of a form of resource invariant synthesis [Cla79]. There are other approaches worth exploring, such as generating ghosts for counting proofs [FKP14] or inferring conditional history variables from counterexamples [VM23]. Hoenicke, Majumdar, and Podelski [HMP17] explore how thread-modular reasoning can avoid ghosts by introducing proof systems that consider interleavings of up to k threads. Farzan, Klumpp, and Podelski [FKP24] present an approach to simplifying proofs, reducing the complexity of the ghost state.

6.10. Conclusion and Future Work

We proposed two approaches to thread-modular correctness witnesses and evaluated each one. In the local view approach, we used the existing witness format to describe the shared global state from the perspective of individual threads. We showed that our witness-guided abstract interpretation seamlessly works together with our thread-modular analyses, yielding improvements in both precision and performance.

In the global view approach, we have extended the existing witness format with ghost variables, which are well established for reasoning about the correctness of concurrent programs. For two types of thread-modular analyses implemented in the GOBLINT abstract interpreter, we have shown how the computed invariants can naturally be expressed in the new format, highlighting that its expressiveness is useful for existing analyses. A validation of the ghost witnesses that GOBLINT generated for a set of concurrent SV-COMP tasks showed that 75% of them could be validated by the model checker UGEMCUTTER — indicating that ghost witnesses can bridge the semantic divide between thread-modular and interleaving approaches. Subsequently, the SV-COMP community has accepted our proposed ghost witness format and it will be demoed in SV-COMP 2026.

Future Work. Although we have shown the feasibility of two flavors of thread-modular correctness witnesses, this is certainly not the end of the road.

Robustness. Example 47 and Remark 23 (pp. 181 and 182) noted that the precision and performance of witness-guided verification using non-ghost witnesses can be quite sensitive to the location of witness invariants. This poses a challenge to witness-guided verification across frameworks which might not agree on the best placement. Section 6.3.1 made a particular choice for generating witnesses for the local view in order to demonstrate their feasibility, but it did not evaluate other possibilities. Hence, future work could study which witness generation strategies are best suited for guidance for which analyses. A further goal would be to design approaches, both for witness generation and witness-guided verification, which are less sensitive to such invariant locations and, thus, more robust overall.

Background-Lockset–Based Witnesses. Witness generation may be improved for analyses using background locksets (Miné's, Lock-Centered Reading, Write-Centered Reading). The construction proposed for the local view in Section 6.3.1 completely ignores background locksets and does not make use of the extra precision from their disjointness. No construction was proposed for the global view in Section 6.7.1, as this likely requires additional families of ghost variables.

Ghost Witness Generation. Section 6.7.1 only considered the generation of ghost witnesses from thread-modular abstract interpretation. Future work may study how the results of other approaches and static analyzers for concurrent programs are best encoded via ghost code.

Ghost Witness Validation. Section 6.7.2 left it as future work to validate ghost witnesses using thread-modular abstract interpretation, particularly in GOBLINT VALIDATOR. Possible directions include the following:

- Design local-trace–based thread-modular analyses (or extend the ones from Chapter 5) to directly handle atomic blocks from ghost-witness–instrumented programs, as opposed to relying on their transformation described in Section 6.6.2. In particular, for the practical case where variables accessed by an atomic section are not statically known due to pointers. This would also bypass the issue described in Remark 25 (p. 195).
- Develop digests [SE24] to refine thread-modular analyses such that they become sufficiently precise for validating ghost witnesses. These digests could be general for some class of ghost variables or specific to a single ghost witness.
- Instead of treating ghost variables opaquely in a sufficiently powerful analysis, another approach would be to infer semantic meanings of ghost variables and treat them specially. For example, from the program in Listing 6b, infer that `m_locked` exactly encodes the locking state of `m`. Knowing this, ghost invariants in specific forms, like the ones in Section 6.7.1, could be used and validated in analysis-specific manner.

  This approach is suitable for validating ghost witnesses generated by the same approach (or a reasonably similar one), but clearly does not generalize to arbitrary use of ghosts.

Non-value Witnesses. In this chapter, both for the local and the global view, we only considered witnesses and invariants about the values of variables. However, thread-modular reasoning also applies to non-value information, e.g., synchronization. For example, Saan and Erhard [SE23] propose YAML witness entry types for the following:

Current lockset at a location, which is flow-sensitive like the local view.

Protecting mutexes for a global variable, which is flow-insensitive like the global view.

Both kinds of information can be used for basic reasoning about race freedom. Since GOBLINT internally computes both, it would be straightforward to generate and validate such witnesses.

More intriguingly, it is also possible to unassume such non-value information. In Example 32 (p. 148), it could guide the analysis of write-protecting mutexes (Remark 17, p. 147) with Footnote 55 (p. 148). This offers a way to semantically control non-monotonicity (see Remarks 15, 17 and 20, pp. 139, 147 and 161). Therefore, it would be interesting to explore both the precision and the performance implications of unassuming non-value information.

Other Combinations. Additionally, the following combinations and applications of witnesses can be considered and evaluated:

Combined local and global views. In principle, nothing prevents the local and the global view witnesses from being combined. GOBLINT can generate both together and UGEMCUTTER can validate them. It remains as future work to determine if such a combination has advantages over the individual approaches.

Single-threaded ghost witnesses. Although we motivated ghost witnesses for the correctness of multi-threaded programs, the proposed formalism, YAML format and UGEMCUTTER validator apply to single-threaded programs without modification. It is to be determined what various approaches could express using ghosts in that case.

Ghost violation witnesses. Ghosts may also be useful when describing violations, both of single- and multi-threaded programs. For example, ghost variables allow assumption expressions in violation witnesses to indirectly refer to data which is lexically out of scope.

7. CONCLUSION

The overall goal of the thesis was to obtain "performant, precise and trustworthy analysis of concurrent programs" by using witnesses. Let us consider each of the subgoals we defined Section 1.1.

In Chapter 4, we introduced a novel unassume operation for guiding abstract interpretation with invariants from witnesses. Through extensive examples and experimental evaluations, we have demonstrated that this operator can successfully increase the precision and performance of an abstract interpreter, fulfilling goals **G1** and **G3**. Having implemented the operation in GOBLINT, we have obtained the first correctness witness validator using abstract interpretation.

In Chapter 5, we compared two approaches to thread-modular abstract interpretation and found them to be incomparable in terms of precision. In true scientific spirit, this was just the beginning of a journey to design and implement new thread-modular analyses, which are more precise than prior work, fulfilling goal **G2**. This fixed a long-standing soundness issue from 2011 in GOBLINT.

In Chapter 6, we brought the research directions from the previous two chapters together and proposed thread-modular correctness witnesses, fulfilling goal **G4**. Through examples and experimental evaluations, we validated our proposals, fulfilling goal **G5**. The proposed ghost witness format has been accepted by the SV-COMP community and will be demoed in SV-COMP 2026.

On Tools. In parallel with all of the above, we meticulously fixed and maintained the soundness of GOBLINT — our preferred abstract interpretation framework. Moreover, we extended its data race analysis with temporal reasoning, using our insights from thread-modular abstract interpretation. All of this has paid off: GOBLINT has been sound all 5 years that it has participated in SV-COMP and won the data race category 4 times.

Participating in SV-COMP has been immensely valuable to GOBLINT. First, the one time work of integrating GOBLINT with SV-COMP infrastructure has allowed us to easily use its over 33,000 benchmarks, the largest suite of its kind. Second, it has motivated us to continuously fix and improve our tool in terms of soundness, precision and performance. Third, it has increased the visibility, awareness and use of GOBLINT in the community.

Implementing witness generation and validation has far-reaching benefits for tools. Witness support exercises other components of the tools in new ways and is surprisingly effective at revealing existing bugs in them. This was already noted by Beyer et al. [Bey+16, Footnote 8, p. 333] in the context of same-framework consistency of CPACHECKER. In Section 6.8.2, we described similar benefits in the context of cross-framework validation between GOBLINT and ULTIMATE.

On Witnesses. The saga of verification witnesses was kicked off 10 years ago by Beyer et al. [Bey+15; Bey+16]. After years of experience, the community realized the limitations of the original approach, and proposed a new, simpler and

extensible format [Aya+24]. This has spawned a spectrum of proposals for new kinds of witnesses, including our ghost witnesses (Section 6.6). Thus, the research on witnesses is just getting started, as there are many gaps to fill and questions to answer.

A witness may be valid, but that does not necessarily make it good. For a correct program, the trivial correctness witness, which provides no (non-trivial) invariants, is valid but completely useless. In contrast, providing redundant or irrelevant invariants is also undesirable and detrimental. SV-COMP takes a very pragmatic approach: witness validators are given less resources than verifiers, in the hope that this pushes verifiers to produce good witnesses which are relatively easy to validate. Unfortunately, this does not account for the large variation in program sizes and tool efficiency. Therefore, the question of correctness witness quality and its measurement remains open.

The pragmatic approach of SV-COMP is further undermined by the fact that many approaches to correctness witness validation do not actually benefit from witness invariants, but only treat them as additional proof burden. In this regard, our witness-guided verification technique is a positive outlier and hopefully serves as motivation for others. Importantly, witness-guided verification implicitly defines what is a good witness (for a particular validator) that actually helps, not hinders. Moreover, witness-guided verification is also crucial for other under- or unexplored applications of witnesses: cooperative verification and incremental verification. If a verifier cannot benefit from a witness produced by another tool or itself, then it might as well verify everything alone and from scratch.

The question remains:

> Have we achieved the overall goal of performant, precise and trustworthy analysis of concurrent programs?

With this thesis, we have certainly made advances in all three aspects, thanks to witnesses. But are we there yet? Possibly not, because even more performant, more precise and more trustworthy analysis is always desirable. It is unlikely that we have unlocked the full potential of witnesses for achieving this noble goal.

BIBLIOGRAPHY

[ABG22] Flavio Ascari, Roberto Bruni, and Roberta Gori. "Limits and difficulties in the design of under-approximation abstract domains". In: *Foundations of Software Science and Computation Structures*. LNCS 13242. Springer, 2022, pp. 21–39. DOI: 10.1007/978-3-030-99253-8_2.

[ABO09] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 2009. DOI: 10.1007/978-1-84882-745-5.

[ACS22] Paulína Ayaziová, Marek Chalupa, and Jan Strejček. "SYMBIOTIC-WITCH: A KLEE-Based Violation Witness Checker (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13244. Springer, 2022, pp. 468–473. DOI: 10.1007/978-3-030-99527-0_33.

[Ádá+25] Zsófia Ádám, Paulína Ayaziová, Levente Bajczi, Dirk Beyer, Marek Jankola, Marian Lingsch-Rosenfeld, and Jan Strejček. "Non-termination Witnesses and Their Validation". In: *Automated Software Engineering*. IEEE, 2025, pp. 1969–1981. DOI: 10.1109/ASE63991.2025.00164.

[AFG12] Elvira Albert, Antonio Flores-Montoya, and Samir Genaim. "Analysis of May-Happen-in-Parallel in Concurrent Objects". In: *Formal Techniques for Distributed Systems*. LNCS 7273. Springer, 2012, pp. 35–51. DOI: 10.1007/978-3-642-30793-5_3.

[Aga+07] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. "May-happen-in-parallel analysis of X10 programs". In: *Principles and Practice of Parallel Programming*. ACM, 2007, pp. 183–193. DOI: 10.1145/1229428.1229471.

[AGG15] Elvira Albert, Samir Genaim, and Pablo Gordillo. "May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization". In: *Static Analysis*. LNCS 9291. Springer, 2015, pp. 72–89. DOI: 10.1007/978-3-662-48288-9_5.

[AGG17] Elvira Albert, Samir Genaim, and Pablo Gordillo. "May-Happen-in-Parallel Analysis with Returned Futures". In: *Automated Technology for Verification and Analysis*. LNCS 10482. Springer, 2017, pp. 42–58. DOI: 10.1007/978-3-319-68167-2_3.

[Alb+06] Elvira Albert, Puri Arenas, Germán Puebla, and Manuel Hermenegildo. "Reduced Certificates for Abstraction-Carrying Code". In: *Logic Programming*. LNCS 4079. Springer, 2006, pp. 163–178. DOI: 10.1007/11799573_14.

[Alg+11a] Jade Alglave, Alastair F. Donaldson, Daniel Kroening, and Michael Tautschnig. “Making Software Verification Tools Really Work”. In: *Automated Technology for Verification and Analysis*. LNCS 6996. Springer, 2011, pp. 28–42. DOI: 10.1007/978-3-642-24372-1_3.

[Alg+11b] Jade Alglave, Daniel Kroening, John Lugton, Vincent Nimal, and Michael Tautschnig. “Soundness of Data Flow Analyses for Weak Memory Models”. In: *Programming Languages and Systems*. LNCS 7078. Springer, 2011, pp. 272–288. DOI: 10.1007/978-3-642-25318-8_21.

[Ama+16] Gianluca Amato, Francesca Scozzari, Helmut Seidl, Kalmer Apinis, and Vesal Vojdani. “Efficiently intertwining widening and narrowing”. In: *Science of Computer Programming* 120 (2016), pp. 1–24. DOI: 10.1016/j.scico.2015.12.005.

[Ama+18] Gianluca Amato, Simone Di Nardo Di Maio, Maria Chiara Meo, and Francesca Scozzari. “Descending chains and narrowing on template abstract domains”. In: *Acta Informatica* 55.6 (2018), pp. 521–545. DOI: 10.1007/s00236-016-0291-0.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. “Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory”. In: *ACM Transactions on Programming Languages and Systems* 36.2 (2014), pp. 7:1–7:74. DOI: 10.1145/2627752.

[AMZ22] Vincenzo Arceri, Isabella Mastroeni, and Enea Zaffanella. “Decoupling the Ascending and Descending Phases in Abstract Interpretation”. In: *Programming Languages and Systems*. LNCS 13658. Springer, 2022, pp. 25–44. DOI: 10.1007/978-3-031-21037-2_2.

[APH05] Elvira Albert, Germán Puebla, and Manuel Hermenegildo. “Abstraction-Carrying Code”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. LNCS 3452. Springer, 2005, pp. 380–397. DOI: 10.1007/978-3-540-32275-7_25.

[Api14] Kalmer Apinis. “Frameworks for analyzing multi-threaded C”. PhD thesis. Technische Universität München, 2014. URN: urn:nbn:de:bvb:91-diss-20140606-1189191-0-9.

[Arm+25] Lukas Armborst, Dirk Beyer, Marieke Huisman, and Marian Lingsch-Rosenfeld. “AutoSV-Annotator: Integrating Deductive and Automatic Software Verification”. In: *Formal Methods for Industrial Critical Systems*. LNCS 16040. Springer, 2025, pp. 59–77. DOI: 10.1007/978-3-032-00942-5_4.

[AS13] Gianluca Amato and Francesca Scozzari. “Localizing Widening and Narrowing”. In: *Static Analysis*. LNCS 7935. Springer, 2013, pp. 25–42. DOI: 10.1007/978-3-642-38856-9_4.

[ASV12] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. “Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis”. In: *Programming Languages and Systems*. LNCS 7705. Springer, 2012, pp. 157–172. DOI: 10.1007/978-3-642-35182-2_12.

[Aya+24] Paulína Ayaziová, Dirk Beyer, Marian Lingsch-Rosenfeld, Martin Spiessl, and Jan Strejček. “Software Verification Witnesses 2.0”. In: *Model Checking Software*. LNCS 14624. Springer, 2024, pp. 184–203. DOI: 10.1007/978-3-031-66149-5_11.

[Bag+05] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. “Precise widening operators for convex polyhedra”. In: *Science of Computer Programming* 58.1–2 (2005), pp. 28–56. DOI: 10.1016/j.scico.2005.02.003.

[Bai+24] Daniel Baier, Dirk Beyer, Po-Chun Chien, Marek Jankola, Matthias Kettl, Nian-Ze Lee, Thomas Lemberger, Marian Lingsch-Rosenfeld, Martin Spiessl, Henrik Wachowitz, and Philipp Wendler. “CPAchecker 2.3 with Strategy Selection (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 14572. Springer, 2024, pp. 359–364. DOI: 10.1007/978-3-031-57256-2_21.

[Bar06] Rajkishore Barik. “Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs”. In: *Languages and Compilers for Parallel Computing*. LNCS 4339. Springer, 2006, pp. 152–169. DOI: 10.1007/978-3-540-69330-7_11.

[Bau+21] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. “The dogged pursuit of bug-free C programs: The Frama-C software analysis platform”. In: *Communications of the ACM* 64.8 (2021), pp. 56–68. DOI: 10.1145/3470569.

[Bau+23] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ANSI/ISO C Specification Language. Version 1.19*. 2023. URL: https://frama-c.com/download/acsl-1.19.pdf (visited on 2026-02-12).

[Ben+99] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. “Revising Hull and Box Consistency”. In: *Logic Programming*. MIT, 1999, pp. 230–244. DOI: 10.7551/mitpress/4304.003.0024.

[Bey+15] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. “Witness Validation and Stepwise Testification across Software Verifiers”. In: *Foundations of Software Engineering*. ACM, 2015, pp. 721–733. DOI: 10.1145/2786805.2786867.

[Bey+16] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. "Correctness Witnesses: Exchanging Verification Results Between Verifiers". In: *Foundations of Software Engineering*. ACM, 2016, pp. 326–337. DOI: 10.1145/2950290.2950351.

[Bey+18] Dirk Beyer, Matthias Dangl, Thomas Lemberger, and Michael Tautschnig. "Tests from Witnesses: Execution-Based Validation of Verification Results". In: *Tests and Proofs*. LNCS 10889. Springer, 2018, pp. 3–23. DOI: 10.1007/978-3-319-92994-1_1.

[Bey+22] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. "Verification Witnesses". In: *ACM Transactions on Software Engineering and Methodology* 31.4 (2022), pp. 57:1–57:69. DOI: 10.1145/3477579.

[Bey+25] Dirk Beyer, Marek Jankola, Marian Lingsch-Rosenfeld, Tian Xia, and Xiyue Zheng. "TransVer: A Modular Program-Transformation Framework for Reduction to Reachability". In: *Model Checking Software*. LNCS 15945. Springer, 2025, pp. 1–24. DOI: 10.1007/978-3-032-06847-7_1.

[Bey20] Dirk Beyer. "Advances in Automatic Software Verification: SV-COMP 2020". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 12079. Springer, 2020, pp. 347–367. DOI: 10.1007/978-3-030-45237-7_21.

[Bey21] Dirk Beyer. "Software Verification: 10th Comparative Evaluation (SV-COMP 2021)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 12652. Springer, 2021, pp. 401–422. DOI: 10.1007/978-3-030-72013-1_24.

[Bey22a] Dirk Beyer. "Progress on Software Verification: SV-COMP 2022". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13244. Springer, 2022, pp. 375–402. DOI: 10.1007/978-3-030-99527-0_20.

[Bey22b] Dirk Beyer. *Verifiers and Validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022)*. Artifact. 2022. DOI: 10.5281/zenodo.5959149.

[Bey23] Dirk Beyer. "Competition on Software Verification and Witness Validation: SV-COMP 2023". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13994. Springer, 2023, pp. 495–522. DOI: 10.1007/978-3-031-30820-8_29.

[Bey24] Dirk Beyer. "State of the Art in Software Verification and Witness Validation: SV-COMP 2024". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 14572. Springer, 2024, pp. 299–329. DOI: 10.1007/978-3-031-57256-2_15.

[BF20] Dirk Beyer and Karlheinz Friedberger. “Violation Witnesses and Result Validation for Multi-Threaded Programs”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*. Vol. 1. LNCS 12476. Springer, 2020, pp. 449–470. DOI: 10.1007/978-3-030-61362-4_26.

[BH17] Rémy Boutonnet and Nicolas Halbwachs. “Improving the results of program analysis by abstract interpretation beyond the decreasing sequence”. In: *Formal Methods in System Design* 53.3 (2017), pp. 384–406. DOI: 10.1007/s10703-017-0310-y.

[Bil+22] Eleni Vafeiadi Bila, Brijesh Dongol, Ori Lahav, Azalea Raad, and John Wickerson. “View-Based Owicki-Gries Reasoning for Persistent x86-TSO”. In: *Programming Languages and Systems*. LNCS 13240. Springer, 2022, pp. 234–261. DOI: 10.1007/978-3-030-99336-8_9.

[BJL25] Dirk Beyer, Marek Jankola, and Marian Lingsch-Rosenfeld. “Transition Invariants as Building Blocks for Termination Witnesses”. Talk. Workshop on Verification Witnesses and Their Validation. 2025. URL: https://vewit.sosy-lab.org/2025/slides/termination-witnesses.pdf (visited on 2025-07-31).

[BJP06] Frédéric Besson, Thomas Jensen, and David Pichardie. “Proof-carrying code from certified abstract interpretation and fixpoint compression”. In: *Theoretical Computer Science* 364.3 (2006), pp. 273–291. DOI: 10.1016/j.tcs.2006.08.012.

[BJT07] Frédéric Besson, Thomas Jensen, and Tiphaine Turpin. “Small Witnesses for Abstract Interpretation-Based Proofs”. In: *Programming Languages and Systems*. LNCS 4421. Springer, 2007, pp. 268–283. DOI: 10.1007/978-3-540-71316-6_19.

[BK11] Dirk Beyer and M. Erkan Keremoglu. “CPAchecker: A Tool for Configurable Software Verification”. In: *Computer Aided Verification*. LNCS 6806. Springer, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16.

[BK22] Dirk Beyer and Sudeep Kanav. “CoVeriTeam: On-Demand Composition of Cooperative Verification Systems”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 1. LNCS 13243. Springer, 2022, pp. 561–579. DOI: 10.1007/978-3-030-99524-9_31.

[BKR22] Dirk Beyer, Sudeep Kanav, and Cedric Richter. “Construction of Verifier Combinations Based on Off-the-Shelf Verifiers”. In: *Fundamental Approaches to Software Engineering*. LNCS 13241. Springer, 2022, pp. 49–70. DOI: 10.1007/978-3-030-99429-7_3.

[BL24] Levente Bajczi and Marian Lingsch-Rosenfeld. “Software Verification Witnesses for Weak Memory”. Report. Formal Methods in Computer-Aided Design. Student Forum. 2024. URL: https://leventebajczi.

com/publications/fmcad_studentforum24_wm_witness.pdf (visited on 2025-07-31).

[BL25] Levente Bajczi and Marian Lingsch-Rosenfeld. "Violation Witnesses for Weak Memory Software". Talk. Workshop on Verification Witnesses and Their Validation. 2025. URL: https://vewit.sosy-lab.org/2025/slides/wmm-witnesses.pdf (visited on 2025-07-31).

[Bla+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "A Static Analyzer for Large Safety-Critical Software". In: *Programming Language Design and Implementation*. ACM, 2003, pp. 196–207. DOI: 10.1145/781131.781153.

[BLW17] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Reliable benchmarking: requirements and solutions". In: *International Journal on Software Tools for Technology Transfer* 21.1 (2017), pp. 1–29. DOI: 10.1007/s10009-017-0469-y.

[Bra+14] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. "IKOS: A Framework for Static Analysis Based on Abstract Interpretation". In: *Software Engineering and Formal Methods*. LNCS 8702. Springer, 2014, pp. 271–277. DOI: 10.1007/978-3-319-10431-7_20.

[Bro07] Stephen Brookes. "A semantics for concurrent separation logic". In: *Theoretical Computer Science* 375.1–3 (2007), pp. 227–270. DOI: 10.1016/j.tcs.2006.12.034.

[Bro14] Stephen Brookes. "On Grainless Footprint Semantics for Shared-memory Programs". In: *Mathematical Foundations of Programming Semantics*. ENTCS 308. Elsevier, 2014, pp. 65–86. DOI: 10.1016/j.entcs.2014.10.005.

[BS20] Dirk Beyer and Martin Spiessl. "MetaVal: Witness Validation via Verification". In: *Computer Aided Verification*. Vol. 2. LNCS 12225. Springer, 2020, pp. 165–177. DOI: 10.1007/978-3-030-53291-8_10.

[BS22a] Dirk Beyer and Martin Spiessl. "The Static Analyzer Frama-C in SV-COMP (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13244. Springer, 2022, pp. 429–434. DOI: 10.1007/978-3-030-99527-0_26.

[BS22b] Dirk Beyer and Jan Strejček. "Case Study on Verification-Witness Validators: Where We Are and Where We Go". In: *Static Analysis*. LNCS 13790. Springer, 2022, pp. 160–174. DOI: 10.1007/978-3-031-22308-2_8.

[BS23a] Kenny Ballou and Elena Sherman. "Minimally Comparing Relational Abstract Domains". In: *Automated Technology for Verification and*

Analysis. Vol. 2. LNCS 14216. Springer, 2023, pp. 159–175. DOI: 10.1007/978-3-031-45332-8_8.

[BS23b] Dirk Beyer and Martin Spiessl. "LIV: Loop-Invariant Validation Using Straight-Line Programs". In: *Automated Software Engineering*. IEEE, 2023, pp. 2074–2077. DOI: 10.1109/ASE56229.2023.00214.

[BS25] Dirk Beyer and Jan Strejček. "Improvements in Software Verification and Witness Validation: SV-COMP 2025". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 15698. Springer, 2025, pp. 151–186. DOI: 10.1007/978-3-031-90660-2_9.

[BSU22] Dirk Beyer, Martin Spiessl, and Sven Umbricht. "Cooperation Between Automatic and Interactive Software Verifiers". In: *Software Engineering and Formal Methods*. LNCS 13550. Springer, 2022, pp. 111–128. DOI: 10.1007/978-3-031-17108-6_7.

[BW20] Dirk Beyer and Heike Wehrheim. "Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework". In: *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*. Vol. 1. LNCS 12476. Springer, 2020, pp. 143–167. DOI: 10.1007/978-3-030-61362-4_8.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Principles of Programming Languages*. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[CC92] Patrick Cousot and Radhia Cousot. "Abstract Interpretation Frameworks". In: *Journal of Logic and Computation* 2.4 (1992), pp. 511–547. DOI: 10.1093/logcom/2.4.511.

[CH78] Patrick Cousot and Nicolas Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program". In: *Principles of Programming Languages*. ACM, 1978, pp. 84–96. DOI: 10.1145/512760.512770.

[Cla79] Edmund M. Clarke. "Synthesis of Resource Invariants for Concurrent Programs". In: *Principles of Programming Languages*. ACM, 1979, pp. 211–221. DOI: 10.1145/567752.567772.

[Cot+22] Alex Coto, Omar Inverso, Emerson Sales, and Emilio Tuosto. "A Prototype for Data Race Detection in CSeq 3 (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13244. Springer, 2022, pp. 413–417. DOI: 10.1007/978-3-030-99527-0_23.

[Cou+05] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The ASTRÉE Analyzer". In: *Programming Languages and Systems*. LNCS 3444. Springer, 2005, pp. 21–30. DOI: 10.1007/978-3-540-31987-0_3.

[Cou+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. "Why does Astrée scale up?" In: *Formal Methods in System Design* 35.3 (2009), pp. 229–264. DOI: 10.1007/s10703-009-0089-6.

[Cou15] Patrick Cousot. "Abstracting Induction by Extrapolation and Interpolation". In: *Verification, Model Checking, and Abstract Interpretation*. LNCS 8931. Springer, 2015, pp. 19–42. DOI: 10.1007/978-3-662-46081-8_2.

[Cou21] Patrick Cousot. *Principles of Abstract Interpretation*. MIT, 2021. ISBN: 978-0-262-04490-5.

[Cou99] Patrick Cousot. "The Calculational Design of a Generic Abstract Interpreter". In: *Calculational System Design*. NATO ASI Series F 173. IOS, 1999, pp. 421–505. URL: https://pcousot.github.io/publications/Cousot-Marktoberdorf98.pdf (visited on 2026-02-03).

[Dal+20] Sadegh Dalvandi, Simon Doherty, Brijesh Dongol, and Heike Wehrheim. "Owicki-Gries Reasoning for C11 RAR". In: *European Conference on Object-Oriented Programming*. LIPIcs 166. Schloss Dagstuhl, 2020, pp. 11:1–11:26. DOI: 10.4230/LIPIcs.ECOOP.2020.11.

[Dal+22] Sadegh Dalvandi, Brijesh Dongol, Simon Doherty, and Heike Wehrheim. "Integrating Owicki-Gries for C11-Style Memory Models into Isabelle/HOL". In: *Journal of Automated Reasoning* 66.1 (2022), pp. 141–171. DOI: 10.1007/S10817-021-09610-2.

[DDN11] Arnab De, Deepak D'Souza, and Rupesh Nasre. "Dataflow Analysis for Datarace-Free Programs". In: *Programming Languages and Systems*. LNCS 6602. Springer, 2011, pp. 196–215. DOI: 10.1007/978-3-642-19718-5_11.

[Deu03] Alain Deutsch. *Static Verification of Dynamic Properties*. White paper. PolySpace Technologies, 2003. URL: https://www.sigada.org/conf/sigada2003/SIGAda2003-CDROM/SIGAda2003-Logos/PolySpace/Marketing/White_Papers/Static_Verification_of_Dynamic_Properties.pdf (visited on 2026-02-03).

[Di+15] Peng Di, Yulei Sui, Ding Ye, and Jingling Xue. "Region-Based May-Happen-in-Parallel Analysis for C Programs". In: *International Conference on Parallel Processing*. IEEE, 2015, pp. 889–898. DOI: 10.1109/ICPP.2015.98.

[Die+23] Daniel Dietsch, Matthias Heizmann, Dominik Klumpp, Frank Schüssele, and Andreas Podelski. "Ultimate Taipan and Race Detection in Ultimate (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13994. Springer, 2023, pp. 582–587. DOI: 10.1007/978-3-031-30820-8_40.

[Dij78] Edsger W. Dijkstra. “Finding the correctness proof of a concurrent program”. In: *Program Construction*. LNCS 69. Springer, 1978, pp. 24–34. DOI: 10.1007/BFb0014652.

[Dis+19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O’Hearn. “Scaling static analyses at Facebook”. In: *Communications of the ACM* 62.8 (2019), pp. 62–70. DOI: 10.1145/3338112.

[DLW15] Matthias Dangl, Stefan Löwe, and Philipp Wendler. “CPAchecker with Support for Recursive Programs and Floating-Point Arithmetic (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. LNCS 9035. Springer, 2015, pp. 423–425. DOI: 10.1007/978-3-662-46681-0_34.

[DV25a] Tomáš Dacík and Tomáš Vojnar. “RacerF: Data Race Detection with Frama-C (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 15698. Springer, 2025, pp. 248–253. DOI: 10.1007/978-3-031-90660-2_20.

[DV25b] Tomáš Dacík and Tomáš Vojnar. “RacerF: Lightweight Static Data Race Detection for C Code”. In: *European Conference on Object-Oriented Programming*. LIPIcs 333. Schloss Dagstuhl, 2025, pp. 37:1–37:19. DOI: 10.4230/LIPIcs.ECOOP.2025.37.

[Erh+24a] Julian Erhard, Manuel Bentele, Matthias Heizmann, Dominik Klumpp, **Simmo Saan**, Frank Schüssele, Michael Schwarz, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. *Artifact for the VMCAI’2025 Paper “Correctness Witnesses for Concurrent Programs: Bridging the Semantic Divide with Ghosts”*. Oct. 2024. DOI: 10.5281/zenodo.13863579.

[Erh+24b] Julian Erhard, Manuel Bentele, Matthias Heizmann, Dominik Klumpp, **Simmo Saan**, Frank Schüssele, Michael Schwarz, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. “Correctness Witnesses for Concurrent Programs: Bridging the Semantic Divide with Ghosts (Extended Version)”. Nov. 2024. DOI: 10.48550/arXiv.2411.16612.

[Erh+24c] Julian Erhard, **Simmo Saan**, Sarah Tilscher, Michael Schwarz, Karoline Holter, Vesal Vojdani, and Helmut Seidl. “Interactive abstract interpretation: reanalyzing multithreaded C programs for cheap”. In: *International Journal on Software Tools for Technology Transfer* 26.6 (Dec. 2024), pp. 647–667. DOI: 10.1007/s10009-024-00768-9.

[Erh+24d] Julian Erhard, Johanna Franziska Schinabeck, Michael Schwarz, and Helmut Seidl. “When to Stop Going Down the Rabbit Hole: Taming Context-Sensitivity on the Fly”. In: *State Of the Art in Program Analysis*. ACM, 2024, pp. 35–44. DOI: 10.1145/3652588.3663321.

[Erh+24e] Julian Erhard, Michael Schwarz, Vesal Vojdani, **Simmo Saan**, and Helmut Seidl. "When long jumps fall short: control-flow tracking and misuse detection for nonlocal jumps in C. Extended version". In: *International Journal on Software Tools for Technology Transfer* 26.5 (Oct. 2024), pp. 589–605. DOI: 10.1007/s10009-024-00764-z.

[Erh+25] Julian Erhard, Manuel Bentele, Matthias Heizmann, Dominik Klumpp, **Simmo Saan**, Frank Schüssele, Michael Schwarz, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. "Correctness Witnesses for Concurrent Programs: Bridging the Semantic Divide with Ghosts". In: *Verification, Model Checking, and Abstract Interpretation*. Vol. 1. LNCS 15529. Springer, Jan. 2025, pp. 74–100. DOI: 10.1007/978-3-031-82700-6_4.

[Faj+16] Lisbeth Fajstrup, Eric Goubault, Emmanuel Haucourt, Samuel Mimram, and Martin Raussen. *Directed Algebraic Topology and Concurrency*. Springer, 2016. DOI: 10.1007/978-3-319-15398-8.

[Fer05] Jérôme Feret. "Abstract interpretation of mobile systems". In: *The Journal of Logic and Algebraic Programming* 63.1 (2005), pp. 59–130. DOI: 10.1016/j.jlap.2004.01.005.

[Fer08] Pietro Ferrara. "Static Analysis Via Abstract Interpretation of the Happens-Before Memory Model". In: *Tests and Proofs*. LNCS 4966. Springer, 2008, pp. 116–133. DOI: 10.1007/978-3-540-79124-9_9.

[FGP16] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. "The spirit of ghost code". In: *Formal Methods in System Design* 48.3 (2016), pp. 152–174. DOI: 10.1007/s10703-016-0243-x.

[FK12] Azadeh Farzan and Zachary Kincaid. "Verification of Parameterized Concurrent Programs by Modular Reasoning about Data and Control". In: *Principles of Programming Languages*. ACM, 2012, pp. 297–308. DOI: 10.1145/2103656.2103693.

[FKP14] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. "Proofs That Count". In: *Principles of Programming Languages*. ACM, 2014, pp. 151–164. DOI: 10.1145/2535838.2535885.

[FKP22] Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. "Sound Sequentialization for Concurrent Program Verification". In: *Programming Language Design and Implementation*. ACM, 2022, pp. 506–521. DOI: 10.1145/3519939.3523727.

[FKP24] Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. "Commutativity Simplifies Proofs of Parameterized Programs". In: *Proceedings of the ACM on Programming Languages* 8.POPL (2024), pp. 83:1–83:29. DOI: 10.1145/3632925.

[FS01] Cormac Flanagan and James B. Saxe. "Avoiding Exponential Explosion: Generating Compact Verification Conditions". In: *Princi-*

ples of Programming Languages. ACM, 2001, pp. 193–205. DOI: 10.1145/360204.360220.

[GC03] Arie Gurfinkel and Marsha Chechik. "Proof-Like Counter-Examples". In: *Tools and Algorithms for the Construction and Analysis of Systems*. LNCS 2619. Springer, 2003, pp. 160–175. DOI: 10.1007/3-540-36577-X_12.

[GG05] Frédéric Goualard and Laurent Granvilliers. "Controlled Propagation in Continuous Numerical Constraint Networks". In: *Symposium on Applied Computing*. ACM, 2005, pp. 377–382. DOI: 10.1145/1066677.1066765.

[Got+07] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. "Thread-Modular Shape Analysis". In: *Programming Language Design and Implementation*. ACM, 2007, pp. 266–277. DOI: 10.1145/1250734.1250765.

[GR06] Denis Gopan and Thomas Reps. "Lookahead Widening". In: *Computer Aided Verification*. LNCS 4144. Springer, 2006, pp. 452–466. DOI: 10.1007/11817963_41.

[Gra89] Philippe Granger. "Static analysis of arithmetical congruences". In: *International Journal of Computer Mathematics* 30.3–4 (1989), pp. 165–190. DOI: 10.1080/00207168908803778.

[Gra92] Philippe Granger. "Improving the results of static analyses of programs by local decreasing iterations". In: *Foundations of Software Technology and Theoretical Computer Science*. LNCS 652. Springer, 1992, pp. 68–79. DOI: 10.1007/3-540-56287-7_95.

[Gri77] David Gries. "An Exercise in Proving Parallel Programs Correct". In: *Communications of the ACM* 20.12 (1977), pp. 921–930. DOI: 10.1145/359897.359903.

[Hal79] Nicolas Halbwachs. "Détermination automatique de relations linéaires vérifiées par les variables d'un programme". French. PhD thesis. Institut National Polytechnique de Grenoble and Université Joseph-Fourier, 1979. URL: https://theses.hal.science/tel-00288805 (visited on 2026-02-08).

[Hei+23] Matthias Heizmann, Max Barth, Daniel Dietsch, Leonard Fichtner, Jochen Hoenicke, Dominik Klumpp, Mehdi Naouar, Tanja Schindler, Frank Schüssele, and Andreas Podelski. "ULTIMATE AUTOMIZER and the CommuHash Normal Form (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13994. Springer, 2023, pp. 577–581. DOI: 10.1007/978-3-031-30820-8_39.

[Hei+24] Matthias Heizmann, Dominik Klumpp, Lars Nitzke, and Frank Schüssele. "Petrification: Software Model Checking for Programs with Dynamic Thread Management". In: *Verification, Model Checking, and*

Abstract Interpretation. Vol. 2. LNCS 14500. Springer, 2024, pp. 3–25. DOI: 10.1007/978-3-031-50521-8_1.

[Hei+25] Matthias Heizmann, Dominik Klumpp, Marian Lingsch-Rosenfeld, and Frank Schüssele. "Correctness Witnesses with Function Contracts". 2025. DOI: 10.48550/arXiv.2501.12313.

[HH12] Nicolas Halbwachs and Julien Henry. "When the Decreasing Sequence Fails". In: *Static Analysis*. LNCS 7460. Springer, 2012, pp. 198–213. DOI: 10.1007/978-3-642-33125-1_15.

[HHP09] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Refinement of Trace Abstraction". In: *Static Analysis*. LNCS 5673. Springer, 2009, pp. 69–85. DOI: 10.1007/978-3-642-03237-0_7.

[HHP13] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Software Model Checking for People Who Love Automata". In: *Computer Aided Verification*. LNCS 8044. Springer, 2013, pp. 36–52. DOI: 10.1007/978-3-642-39799-8_2.

[HMP17] Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. "Thread Modularity at Many Levels: A Pearl in Compositional Verification". In: *Principles of Programming Languages*. ACM, 2017, pp. 473–485. DOI: 10.1145/3009837.3009893.

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[Hol+24a] Karoliine Holter, Juhan Oskar Hennoste, Patrick Lam, **Simmo Saan**, and Vesal Vojdani. "Abstract Debuggers: Exploring Program Behaviors using Static Analysis Results". In: *New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, Oct. 2024, pp. 130–146. DOI: 10.1145/3689492.3690053.

[Hol+24b] Karoliine Holter, Juhan Oskar Hennoste, **Simmo Saan**, Patrick Lam, and Vesal Vojdani. "Abstract Debugging with GobPie". In: *Future Debugging Techniques*. ACM, Sept. 2024, pp. 32–33. DOI: 10.1145/3678720.3685320.

[Hol+25] Karoliine Holter, **Simmo Saan**, Patrick Lam, and Vesal Vojdani. "Sound Static Data Race Verification for C: Is the Race Lost?" In: *ACM Transactions on Programming Languages and Systems* 47.3 (Sept. 2025), pp. 11:1–11:40. DOI: 10.1145/3732933.

[HR23] Jaemin Hong and Sukyoung Ryu. "Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs". In: *International Conference on Software Engineering*. IEEE, 2023, pp. 716–728. DOI: 10.1109/ICSE48619.2023.00069.

[HSF22] Fei He, Zhihang Sun, and Hongyu Fan. "DEAGLE: An SMT-based Verifier for Multi-threaded Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13244. Springer, 2022, pp. 424–428. DOI: 10.1007/978-3-030-99527-0_25.

[HW22] Jan Haltermann and Heike Wehrheim. "Information Exchange Between Over- and Underapproximating Software Analyses". In: *Software Engineering and Formal Methods*. LNCS 13550. Springer, 2022, pp. 37–54. DOI: 10.1007/978-3-031-17108-6_3.

[ISO11] International Organization for Standardization and International Electrotechnical Commission. *ISO/IEC 9899:2011. Information technology – Programming languages – C*. International standard. Committee draft N1570. 2011. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf (visited on 2026-02-11).

[JM09] Bertrand Jeannet and Antoine Miné. "APRON: A Library of Numerical Abstract Domains for Static Analysis". In: *Computer Aided Verification*. LNCS 5643. Springer, 2009, pp. 661–667. DOI: 10.1007/978-3-642-02658-4_52.

[JMO19] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. "An Abstract Domain for Trees with Numeric Relations". In: *Programming Languages and Systems*. LNCS 11423. Springer, 2019, pp. 724–751. DOI: 10.1007/978-3-030-17184-1_26.

[Jon83] Cliff B. Jones. "Tentative Steps Toward a Development Method for Interfering Programs". In: *ACM Transactions on Programming Languages and Systems* 5.4 (1983), pp. 596–619. DOI: 10.1145/69575.69577.

[Jou19] Matthieu Journault. "Precise and modular static analysis by abstract interpretation for the automatic proof of program soundness and contracts inference". PhD thesis. Sorbonne Université, 2019. URL: https://theses.hal.science/tel-02947214 (visited on 2026-02-08).

[Kah+07] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. "Fast and Accurate Static Data-Race Detection for Concurrent Programs". In: *Computer Aided Verification*. LNCS 4590. Springer, 2007, pp. 226–239. DOI: 10.1007/978-3-540-73368-3_26.

[Kar76] Michael Karr. "Affine Relationships Among Variables of a Program". In: *Acta Informatica* 6.2 (1976), pp. 133–151. DOI: 10.1007/BF00268497.

[Kel76] Robert M. Keller. "Formal Verification of Parallel Programs". In: *Communications of the ACM* 19.7 (1976), pp. 371–384. DOI: 10.1145/360248.360251.

[KIG05] Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. “Reasoning About Threads Communicating via Locks”. In: *Computer Aided Verification*. LNCS 3576. Springer, 2005, pp. 505–518. DOI: 10.1007/11513988_49.

[KL22] Matthias Kettl and Thomas Lemberger. “The Static Analyzer INFER in SV-COMP (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13244. Springer, 2022, pp. 451–456. DOI: 10.1007/978-3-030-99527-0_30.

[Klu+22] Dominik Klumpp, Daniel Dietsch, Matthias Heizmann, Frank Schüssele, Marcel Ebbinghaus, Azadeh Farzan, and Andreas Podelski. “ULTIMATE GEMCUTTER and the Axes of Generalization (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13244. Springer, 2022, pp. 479–483. DOI: 10.1007/978-3-030-99527-0_35.

[KPS24] Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles. *Guide to Software Verification with FRAMA-C. Core Components, Usages, and Applications*. Computer Science Foundations and Applied Logic. Springer, 2024. DOI: 10.1007/978-3-031-55608-1.

[Kre+17] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. “The Essence of Higher-Order Concurrent Separation Logic”. In: *Programming Languages and Systems*. LNCS 10201. Springer, 2017, pp. 696–723. DOI: 10.1007/978-3-662-54434-1_26.

[KW16] Markus Kusano and Chao Wang. “Flow-Sensitive Composition of Thread-Modular Abstract Interpretation”. In: *Foundations of Software Engineering*. ACM, 2016, pp. 799–809. DOI: 10.1145/2950290.2950291.

[KW17] Markus Kusano and Chao Wang. “Thread-Modular Static Analysis for Relaxed Memory Models”. In: *Foundations of Software Engineering*. ACM, 2017, pp. 337–348. DOI: 10.1145/3106237.3106243.

[Lam77] Leslie Lamport. “Proving the Correctness of Multiprocess Programs”. In: *IEEE Transactions on Software Engineering* 3.2 (1977), pp. 125–143. DOI: 10.1109/TSE.1977.229904.

[Lam78] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563.

[Lam79] Leslie Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers* 28.9 (1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439.

[LF08] Francesco Logozzo and Manuel Fähndrich. “Pentagons: A Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses”. In: *Symposium on Applied Computing*. ACM, 2008, pp. 184–188. DOI: 10.1145/1363686.1363736.

[Liv+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. “In Defense of Soundiness: A Manifesto”. In: *Communications of the ACM* 58.2 (2015), pp. 44–46. DOI: 10.1145/2644805.

[LN13] Ruy Ley-Wild and Aleksandar Nanevski. “Subjective Auxiliary State for Coarse-Grained Concurrency”. In: *Principles of Programming Languages*. ACM, 2013, pp. 561–574. DOI: 10.1145/2429069.2429134.

[Lom77] David B. Lomet. “Process Structuring, Synchronization, and Recovery Using Atomic Actions”. In: *Language Design for Reliable Software*. ACM, 1977, pp. 128–137. DOI: 10.1145/800022.808319.

[Lub84] Boris D. Lubachevsky. “An Approach to Automating the Verification of Compact Parallel Coordination Programs. I”. In: *Acta Informatica* 21.2 (1984), pp. 125–169. DOI: 10.1007/BF00289237.

[LV15] Ori Lahav and Viktor Vafeiadis. “Owicki-Gries Reasoning for Weak Memory Models”. In: *Automata, Languages, and Programming*. Vol. 2. LNCS 9135. Springer, 2015, pp. 311–323. DOI: 10.1007/978-3-662-47666-6_25.

[Maz86] Antoni Mazurkiewicz. “Trace Theory”. In: *Advances in Petri Nets*. Vol. 2. LNCS 255. Springer, 1986, pp. 279–324. DOI: 10.1007/3-540-17906-2_30.

[McC+06] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. “AUTOLOCKER: Synchronization Inference for Atomic Sections”. In: *Principles of Programming Languages*. ACM, 2006, pp. 346–358. DOI: 10.1145/1111037.1111068.

[Min01] Antoine Miné. “The Octagon Abstract Domain”. In: *Working Conference on Reverse Engineering*. IEEE, 2001, pp. 310–319. DOI: 10.1109/WCRE.2001.957836.

[Min04] Antoine Miné. “Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors”. In: *Programming Languages and Systems*. LNCS 2986. Springer, 2004, pp. 3–17. DOI: 10.1007/978-3-540-24725-8_2.

[Min06] Antoine Miné. “The octagon abstract domain”. In: *Higher-Order and Symbolic Computation* 19.1 (2006), pp. 31–100. DOI: 10.1007/s10990-006-8609-1.

[Min12] Antoine Miné. “Static Analysis of Run-Time Errors in Embedded Real-Time Parallel C Programs”. In: *Logical Methods in Computer Science* 8.1 (2012), pp. 26:1–26:63. DOI: 10.2168/lmcs-8(1:26)2012.

[Min14a] Antoine Miné. “Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions”. In: *Science of Computer Programming* 93.B (2014), pp. 154–182. DOI: 10.1016/j.scico.2013.09.014.

[Min14b] Antoine Miné. “Relational Thread-Modular Static Value Analysis by Abstract Interpretation”. In: *Verification, Model Checking, and Abstract Interpretation*. LNCS 8318. Springer, 2014, pp. 39–58. DOI: 10.1007/978-3-642-54013-4_3.

[Min17] Antoine Miné. “Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation”. In: *Foundations and Trends in Programming Languages* 4.3–4 (2017), pp. 120–372. DOI: 10.1561/2500000034. URL: https://hal.science/hal-01657536v1 (visited on 2026-02-12).

[MM17] Raphaël Monat and Antoine Miné. “Precise Thread-Modular Abstract Interpretation of Concurrent Programs Using Relational Interference Abstractions”. In: *Verification, Model Checking, and Abstract Interpretation*. LNCS 10145. Springer, 2017, pp. 386–404. DOI: 10.1007/978-3-319-52234-0_21.

[MM24] Marco Milanese and Antoine Miné. “Generation of Violation Witnesses by Under-Approximating Abstract Interpretation”. In: *Verification, Model Checking, and Abstract Interpretation*. Vol. 1. LNCS 14499. Springer, 2024, pp. 50–73. DOI: 10.1007/978-3-031-50524-9_3.

[MNY14] Ravi Mangal, Mayur Naik, and Hongseok Yang. “A Correspondence between Two Approaches to Interprocedural Analysis in the Presence of Join”. In: *Programming Languages and Systems*. LNCS 8410. Springer, 2014, pp. 513–533. DOI: 10.1007/978-3-642-54833-8_27.

[MOM23] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. “Mopsa-C: Modular Domains and Relational Abstract Interpretation for C Programs (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13994. Springer, 2023, pp. 565–570. DOI: 10.1007/978-3-031-30820-8_37.

[MOM25] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. “Mopsa-C with Trace Partitioning and Autosuggestions (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 15698. Springer, 2025, pp. 229–235. DOI: 10.1007/978-3-031-90660-2_17.

[Mon+24] Raphaël Monat, Marco Milanese, Francesco Parolini, Jérôme Boillot, Abdelraouf Ouadjaout, and Antoine Miné. "MOPSA-C: Improved Verification for C Programs, Simple Validation of Correctness Witnesses (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 14572. Springer, 2024, pp. 387–392. DOI: 10.1007/978-3-031-57256-2_26.

[MSS13] Bogdan Mihaila, Alexander Sepp, and Axel Simon. "Widening as Abstract Domain". In: *NASA Formal Methods*. LNCS 7871. Springer, 2013, pp. 170–184. DOI: 10.1007/978-3-642-38088-4_12.

[Muk+17] Suvam Mukherjee, Oded Padon, Sharon Shoham, Deepak D'Souza, and Noam Rinetzky. "Thread-Local Semantics and Its Efficient Sequential Abstractions for Race-Free Programs". In: *Static Analysis*. LNCS 10422. Springer, 2017, pp. 253–276. DOI: 10.1007/978-3-319-66706-5_13.

[NAC99] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. "An Efficient Algorithm for Computing MHP Information for Concurrent JAVA Programs". In: *Foundations of Software Engineering*. LNCS 1687. Springer, 1999, pp. 338–354. DOI: 10.1007/3-540-48166-4_21.

[Nan+14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. "Communicating State Transition Systems for Fine-Grained Concurrent Resources". In: *Programming Languages and Systems*. LNCS 8410. Springer, 2014, pp. 290–310. DOI: 10.1007/978-3-642-54833-8_16.

[Nan+19] Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. "Specifying Concurrent Programs in Separation Logic: Morphisms and Simulations". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 161:1–161:30. DOI: 10.1145/3360587.

[Nec+02] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. "CIL: Intermediate language and tools for analysis and transformation of C programs". In: *Compiler Construction*. LNCS 2304. Springer, 2002, pp. 213–228. DOI: 10.1007/3-540-45937-5_16.

[Nie01] Leonor Prensa Nieto. "Completeness of the Owicki-Gries System for Parameterized Parallel Programs". In: *International Parallel and Distributed Processing Symposium*. IEEE, 2001, pp. 1534–1541. DOI: 10.1109/IPDPS.2001.925138.

[Nie02] Leonor Prensa Nieto. "Verification of parallel programs with the Owicki-Gries and Rely-Guarantee methods in ISABELLE/HOL". PhD thesis. Technische Universität München, 2002. URN: urn:nbn:de:bvb:91-diss2002021117027.

[NN99] Tobias Nipkow and Leonor Prensa Nieto. "Owicki/Gries in Isabelle/HOL". In: *Fundamental Approaches to Software Engineering*. LNCS 1577. Springer, 1999, pp. 188–203. DOI: 10.1007/978-3-540-49020-3_13.

[OAS20] Organization for the Advancement of Structured Information Standards. *Static Analysis Results Interchange Format (SARIF) Version 2.1.0*. OASIS standard. 2020. URL: https://docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html (visited on 2024-04-16).

[OG76] Susan Owicki and David Gries. "Verifying Properties of Parallel Programs: An Axiomatic Approach". In: *Communications of the ACM* 19.5 (1976), pp. 279–285. DOI: 10.1145/360051.360224.

[OHe07] Peter W. O'Hearn. "Resources, concurrency, and local reasoning". In: *Theoretical Computer Science* 375.1–3 (2007), pp. 271–307. DOI: 10.1016/j.tcs.2006.12.035.

[PFH11] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. "Locksmith: Practical Static Race Detection for C". In: *ACM Transactions on Programming Languages and Systems* 33.1 (2011), pp. 3:1–3:55. DOI: 10.1145/1889997.1890000.

[PHM22] Hernán Ponce-de-León, Thomas Haas, and Roland Meyer. "Dartagnan: SMT-based Violation Witness Validation (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13244. Springer, 2022, pp. 418–423. DOI: 10.1007/978-3-030-99527-0_24.

[Pon+20] Hernán Ponce-de-León, Florian Furbach, Keijo Heljanko, and Roland Meyer. "Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 12079. Springer, 2020, pp. 378–382. DOI: 10.1007/978-3-030-45237-7_24.

[RED16] Eric F. Rizzi, Sebastian Elbaum, and Matthew B. Dwyer. "On the Techniques We Create, the Tools We Build, and Their Misalignments: a Study of KLEE". In: *International Conference on Software Engineering*. ACM, 2016, pp. 132–143. DOI: 10.1145/2884781.2884835.

[Reg10] John Regehr. *A Guide to Undefined Behavior in C and C++. Part 1*. 2010. URL: https://blog.regehr.org/archives/213 (visited on 2025-07-02).

[RLV20] Azalea Raad, Ori Lahav, and Viktor Vafeiadis. "Persistent Owicki-Gries Reasoning: A Program Logic for Reasoning about Persistent Programs on Intel-x86". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 151:1–151:28. DOI: 10.1145/3428219.

[Saa+20] **Simmo Saan**, Michael Schwarz, Kalmer Apinis, Julian Erhard, Helmut Seidl, Ralf Vogler, and Vesal Vojdani. *GOBLINT at SV-COMP 2021*. Version svcomp21-0-g82e03b87. Artifact. Dec. 2020. DOI: 10.5281/zenodo.4485853.

[Saa+21] **Simmo Saan**, Michael Schwarz, Kalmer Apinis, Julian Erhard, Helmut Seidl, Ralf Vogler, and Vesal Vojdani. "GOBLINT: Thread-Modular Abstract Interpretation Using Side-Effecting Constraints (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 12652. Springer, Mar. 2021, pp. 438–442. DOI: 10.1007/978-3-030-72013-1_28.

[Saa+22] **Simmo Saan**, Michael Schwarz, Julian Erhard, Manuel Pietsch, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. *GOBLINT at SV-COMP 2023*. Version svcomp23-0-g4f5dcf38f. Artifact. Nov. 2022. DOI: 10.5281/zenodo.7467093.

[Saa+23a] **Simmo Saan**, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl. *GOBLINT at SV-COMP 2024*. Version svcomp24-0-gc2e9465a7. Artifact. Nov. 2023. DOI: 10.5281/zenodo.10202867.

[Saa+23b] **Simmo Saan**, Michael Schwarz, Julian Erhard, Manuel Pietsch, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. "GOBLINT: Autotuning Thread-Modular Abstract Interpretation (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13994. Springer, Apr. 2023, pp. 547–552. DOI: 10.1007/978-3-031-30820-8_34.

[Saa+23c] **Simmo Saan**, Michael Schwarz, Julian Erhard, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. "Correctness Witness Validation by Abstract Interpretation". Extended version. Oct. 2023. DOI: 10.48550/arXiv.2310.16572.

[Saa+23d] **Simmo Saan**, Michael Schwarz, Julian Erhard, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. *Correctness Witness Validation by Abstract Interpretation*. Artifact. 2023. DOI: 10.5281/zenodo.8253000.

[Saa+24a] **Simmo Saan**, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl. "GOBLINT VALIDATOR: Correctness Witness Validation by Abstract Interpretation (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 14572. Springer, Apr. 2024, pp. 335–340. DOI: 10.1007/978-3-031-57256-2_17.

[Saa+24b] **Simmo Saan**, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl. "GOBLINT: Abstract Interpretation for Memory Safety and Termination (Competition Contribution)". In: *Tools and Algorithms for the*

Construction and Analysis of Systems. Vol. 3. LNCS 14572. Springer, Apr. 2024, pp. 381–386. DOI: 10.1007/978-3-031-57256-2_25.

[Saa+24c] **Simmo Saan**, Karoliine Holter, Julian Erhard, Michael Schwarz, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl. *GOBLINT at SV-COMP 2025*. Version svcomp25-0-g5512d834b. Artifact. Nov. 2024. DOI: 10.5281/zenodo.14054652.

[Saa+24d] **Simmo Saan**, Michael Schwarz, Julian Erhard, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. "Correctness Witness Validation by Abstract Interpretation". In: *Verification, Model Checking, and Abstract Interpretation*. Vol. 1. LNCS 14499. Springer, Jan. 2024, pp. 74–97. DOI: 10.1007/978-3-031-50524-9_4.

[Saa20] **Simmo Saan**. "Witness Generation for Data-flow Analysis". MSc thesis. University of Tartu, 2020. HDL: 10062/94066.

[Saa23] **Simmo Saan**. *Thread-Modular Correctness Witnesses and their Validation: An Abstract Interpretation Perspective*. Artifact. Feb. 2023. DOI: 10.5281/zenodo.7585192.

[ŠBK20] Jan Švejda, Philipp Berger, and Joost-Pieter Katoen. "Interpretation-Based Violation Witness Validation for C: NITWIT". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 1. LNCS 12078. Springer, 2020, pp. 40–57. DOI: 10.1007/978-3-030-45190-5_3.

[Sch+21a] Michael Schwarz, **Simmo Saan**, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. "Improving Thread-Modular Abstract Interpretation". In: *Static Analysis*. LNCS 12913. Springer, Oct. 2021, pp. 359–383. DOI: 10.1007/978-3-030-88806-0_18.

[Sch+21b] Michael Schwarz, **Simmo Saan**, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. "Improving Thread-Modular Abstract Interpretation". Extended version. Aug. 2021. DOI: 10.48550/arXiv.2108.07613.

[Sch+21c] Michael Schwarz, **Simmo Saan**, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. *Improving Thread-Modular Abstract Interpretation*. Artifact. July 2021. DOI: 10.5281/zenodo.5148608.

[Sch+23a] Michael Schwarz, Julian Erhard, Vesal Vojdani, **Simmo Saan**, and Helmut Seidl. "When Long Jumps Fall Short: Control-Flow Tracking and Misuse Detection for Non-local Jumps in C". In: *State Of the Art in Program Analysis*. ACM, June 2023, pp. 20–26. DOI: 10.1145/3589250.3596140.

[Sch+23b] Michael Schwarz, **Simmo Saan**, Helmut Seidl, Julian Erhard, and Vesal Vojdani. "Clustered Relational Thread-Modular Abstract Interpretation with Local Traces". In: *Programming Languages and Systems*. LNCS 13990. Springer, Apr. 2023, pp. 28–58. DOI: 10.1007/978-3-031-30044-8_2.

[Sch+23c] Michael Schwarz, **Simmo Saan**, Helmut Seidl, Julian Erhard, and Vesal Vojdani. “Clustered Relational Thread-Modular Abstract Interpretation with Local Traces”. Extended version. Jan. 2023. DOI: 10.48550/arXiv.2301.06439.

[Sch+23d] Michael Schwarz, **Simmo Saan**, Helmut Seidl, Julian Erhard, and Vesal Vojdani. *Clustered Relational Thread-Modular Abstract Interpretation with Local Traces*. Artifact. Jan. 2023. DOI: 10.5281/zenodo.7767778.

[Sch+26] Frank Schüssele, Matthias Zumkeller, Miriam Lagunes-Rochin, and Dominik Klumpp. “The Ghosts of Empires: Extracting Modularity from Interleaving-Based Proofs”. In: *Proceedings of the ACM on Programming Languages* 10.POPL (2026), pp. 42:1–42:29. DOI: 10.1145/3776684.

[Sch12] Sabine Bettina Schmaltz. “Towards the Pervasive Formal Verification of Multi-Core Operating Systems and and Hypervisors Implemented in C”. PhD thesis. Universität des Saarlandes, 2012. DOI: 10.22028/D291-26525.

[Sch14] Martin Schwarz. “Static Analysis of Embedded Software with Priority Scheduling and Interrupts”. PhD thesis. Technische Universität München, 2014. URN: urn:nbn:de:101:1-2015022721853. URL: https://www2.in.tum.de/bib/files/diss-schwarz.pdf (visited on 2026-02-08).

[Sch25] Michael Schwarz. “Thread-Modular Abstract Interpretation: The Local Perspective”. PhD thesis. Technische Universität München, 2025. URN: urn:nbn:de:bvb:91-diss-20250611-1765098-0-4.

[Sch97] Thomas Schreiber. “Auxiliary Variables and Recursive Procedures”. In: *Theory and Practice of Software Development*. LNCS 1214. Springer, 1997, pp. 697–711. DOI: 10.1007/BFb0030635.

[SD23] Mikhail Semenyuk and Brijesh Dongol. “Ownership-Based Owicki-Gries Reasoning”. In: *Symposium on Applied Computing*. ACM, 2023, pp. 1685–1694. DOI: 10.1145/3555776.3577636.

[SE23] **Simmo Saan** and Julian Erhard. “Beyond Automaton-Based Witnesses and Location Invariants”. Talk. Workshop on Cooperative Software Verification. 2023. URL: https://coop.sosy-lab.org/2023/slides/coop2023-yaml-witnesses.pdf (visited on 2025-06-03).

[SE24] Michael Schwarz and Julian Erhard. “The digest framework: concurrency-sensitivity for abstract interpretation”. In: *International Journal on Software Tools for Technology Transfer* 26.6 (2024). DOI: 10.1007/s10009-024-00773-y.

[SE26] Michael Schwarz and Julian Erhard. “Data Race Detection by Digest-Driven Abstract Interpretation”. In: *Verification, Model Checking, and*

Abstract Interpretation. LNCS 16417. Springer, 2026, pp. 309–334. DOI: 10.1007/978-3-032-15700-3_15.

[SEV20] Helmut Seidl, Julian Erhard, and Ralf Vogler. "Incremental Abstract Interpretation". In: *From Lambda Calculus to Cybersecurity Through Program Analysis*. LNCS 12065. Springer, 2020, pp. 132–148. DOI: 10.1007/978-3-030-41103-9_5.

[SM16] Thibault Suzanne and Antoine Miné. "From Array Domains to Abstract Interpretation Under Store-buffer-based Memory Models". In: *Static Analysis*. LNCS 9837. Springer, 2016, pp. 469–488. DOI: 10.1007/978-3-662-53413-7_23.

[SM18] Thibault Suzanne and Antoine Miné. "Relational Thread-Modular Abstract Interpretation Under Relaxed Memory Models". In: *Programming Languages and Systems*. LNCS 11275. Springer, 2018, pp. 109–128. DOI: 10.1007/978-3-030-02768-1_6.

[SNB15] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. "Mechanized Verification of Fine-Grained Concurrent Programs". In: *Programming Language Design and Implementation*. ACM, 2015, pp. 77–87. DOI: 10.1145/2737924.2737964.

[SS21] Divyanjali Sharma and Subodh Sharma. "Thread-Modular Analysis of Release-Acquire Concurrency". In: *Static Analysis*. LNCS 12913. Springer, 2021, pp. 384–404. DOI: 10.1007/978-3-030-88806-0_19.

[ST17] Marten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. 3rd ed. 2017. URL: https://www.distributed-systems.net (visited on 2026-02-12).

[Ste+25] Fabian Stemmler, Michael Schwarz, Julian Erhard, Sarah Tilscher, and Helmut Seidl. "Taking Out the Toxic Trash: Recovering Precision in Mixed Flow-Sensitive Static Analyses". In: *Proceedings of the ACM on Programming Languages* 9.PLDI (2025), pp. 194:1–194:27. DOI: 10.1145/3729297.

[Str22] Jan Strejček. *Issues related to the fact that the semantics of witnesses are defined over CFAs and the translation from C programs to CFAs is undefined*. 2022. URL: https://gitlab.com/sosy-lab/benchmarking/sv-witnesses/-/blob/svcomp23/GraphML_witness_format_issues.pdf (visited on 2026-01-21).

[SV09] Helmut Seidl and Vesal Vojdani. "Region Analysis for Race Detection". In: *Static Analysis*. LNCS 5673. Springer, 2009, pp. 171–187. DOI: 10.1007/978-3-642-03237-0_13.

[SV17] Helmut Seidl and Ralf Vogler. "Proving Absence of Starvation by Means of Abstract Interpretation and Model Checking". In: *Automated Technology for Verification and Analysis*. LNCS 10482. Springer, 2017, pp. 3–22. DOI: 10.1007/978-3-319-68167-2_1.

[SV21] Helmut Seidl and Ralf Vogler. “Three improvements to the top-down solver”. In: *Mathematical Structures in Computer Science* 31.9 (2021), pp. 1090–1134. DOI: 10.1017/S0960129521000499.

[SVM03] Helmut Seidl, Varmo Vene, and Markus Müller-Olm. “Global invariants for analyzing multithreaded applications”. In: *Proceedings of the Estonian Academy of Sciences. Physics, Mathematics* 52.4 (2003), pp. 413–436. DOI: 10.3176/phys.math.2003.4.05.

[SVV09] Helmut Seidl, Vesal Vojdani, and Varmo Vene. “A Smooth Combination of Linear and Herbrand Equalities for Polynomial Time Must-Alias Analysis”. In: *Formal Methods*. LNCS 5850. Springer, 2009, pp. 644–659. DOI: 10.1007/978-3-642-05089-3_41.

[Tel+25] Csanád Telbisz, Levente Bajczi, Dániel Szekeres, and András Vörös. “THETA: Various Approaches for Concurrent Program Verification (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 15698. Springer, 2025, pp. 260–265. DOI: 10.1007/978-3-031-90660-2_22.

[Tör23] Edwin Török. “Targeted Static Analysis for OCAML C Stubs: eliminating gremlins from the code”. 2023. DOI: 10.48550/arXiv.2307.14909.

[VM23] Cole Vick and Kenneth L. McMillan. “Synthesizing History and Prophecy Variables for Symbolic Model Checking”. In: *Verification, Model Checking, and Abstract Interpretation*. LNCS 13881. Springer, 2023, pp. 320–340. DOI: 10.1007/978-3-031-24950-1_15.

[Voj+16] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. “Static Race Detection for Device Drivers: The GOBLINT Approach”. In: *Automated Software Engineering*. ACM, 2016, pp. 391–402. DOI: 10.1145/2970276.2970337.

[Voj06] Vesal Vojdani. “Mitmelõimeliste C-programmide kraasimine analüsaatoriga GOBLIN”. Estonian. MSc thesis. University of Tartu, 2006. HDL: 10062/1253.

[Voj10] Vesal Vojdani. “Static Data Race Analysis of Heap-Manipulating C Programs”. PhD thesis. University of Tartu, 2010. HDL: 10062/15866.

[VV07] Vesal Vojdani and Varmo Vene. “GOBLINT: Path-Sensitive Data Race analysis”. In: *Symposium on Programming Languages and Software Tools*. Eötvös Loránd University, 2007, pp. 130–141. ISBN: 978-963-463-925-1.

[VV09] Vesal Vojdani and Varmo Vene. “GOBLINT: Path-sensitive data race analysis”. In: *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica* 30 (2009), pp. 141–155. URL: https://ac.inf.elte.hu/Vol_030_2009/doi/141_30.pdf (visited on 2026-02-07).

[WC24] Zhen Wang and Zhenbang Chen. “AISE: A Symbolic Verifier by Synergizing Abstract Interpretation and Symbolic Execution (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 14572. Springer, 2024, pp. 347–352. DOI: 10.1007/978-3-031-57256-2_19.

[Wen13] Philipp Wendler. “CPAchecker with Sequential Combination of Explicit-State Analysis and Predicate Analysis (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. LNCS 7795. Springer, 2013, pp. 613–615. DOI: 10.1007/978-3-642-36742-7_45.

[Wri+23] Daniel Wright, Sadegh Dalvandi, Mark Batty, and Brijesh Dongol. “Mechanised Operational Reasoning for C11 Programs with Relaxed Dependencies”. In: *Formal Aspects of Computing* 35.2 (2023), pp. 10:1–10:27. DOI: 10.1145/3580285.

[Wu+25] Tong Wu, Xianzhiyu Li, Edoardo Manino, Rafael Sá Menezes, Mikhail R. Gadelha, Shale Xiong, Norbert Tihanyi, Pavlos Petoumenos, and Lucas C. Cordeiro. “ESBMC v7.7: Efficient Concurrent Software Verification with Scheduling, Incremental SMT and Partial Order Reduction (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 15698. Springer, 2025, pp. 223–228. DOI: 10.1007/978-3-031-90660-2_16.

[Yak+25] Boris Yakobowski, Richard Bonichon, David Bühler, and Basile Desloges. *The Mthread plug-in. For Frama-C 32.0 (Germanium)*. 2025. URL: https://frama-c.com/download/mthread-manual-32.0-Germanium.pdf (visited on 2026-02-08).

[Zha+12] Zipeng Zhang, Xinyu Feng, Ming Fu, Zhong Shao, and Yong Li. “A Structural Approach to Prophecy Variables”. In: *Theory and Applications of Models of Computation*. LNCS 7287. Springer, 2012, pp. 61–71. DOI: 10.1007/978-3-642-29952-0_12.

[Zho+18] Qing Zhou, Lian Li, Lei Wang, Jingling Xue, and Xiaobing Feng. “May-Happen-in-Parallel Analysis with Static Vector Clocks”. In: *Code Generation and Optimization*. ACM, 2018, pp. 228–240. DOI: 10.1145/3168813.

[Zia19] Ghiles Ziat. “A Combination of Abstract Interpretation and Constraint Programming”. PhD thesis. Sorbonne Université, 2019. URL: https://theses.hal.science/tel-03987752 (visited on 2026-02-08).

ACKNOWLEDGEMENTS

First, I am grateful to my supervisor **Vesal Vojdani**. I took my first steps into abstract interpretation and the GOBLINT tool for my BSc thesis, which he already co-supervised. During my MSc studies, Vesal introduced me to thread-modular abstract interpretation, in particular the two existing approaches compared in this thesis. I learned about SV-COMP and witnesses for my MSc thesis, which he also supervised. Thus, this PhD thesis would simply not exist without Vesal's vision.

The gratitude extends to Vesal's research group, especially **Kalmer Apinis** and **Karoliine Holter**. Kalmer's internal review suggested improvements to the clarity of my thesis. Karoliine took over some side quests and completed them with publications not included in this thesis.

Second, I am grateful to my academic grandfather **Helmut Seidl**. He essentially served as my unofficial co-supervisor and co-authored all the publications included in this thesis. Helmut made sure that the art of abstract nonsense would not be forgotten.

The gratitude extends to Helmut's research group at TUM, especially his PhD students **Michael Schwarz**, **Julian Erhard** and **Sarah Tilscher**. Our weekly online GobCons and yearly in-person GobCons maintained the close international collaboration between the two groups. This took me on various detours which often ended with publications not included in this thesis. All in all, the benefits of this collaboration flow in both directions.

Third, I am grateful to **Patrick Cousot**, the co-inventor of abstract interpretation. The groundbreaking papers on abstract interpretation are among the earliest works cited in this thesis. Thus, I was very surprised but also unbelievably honored to have Patrick review my thesis.

Fourth, I am grateful to **Dirk Beyer**, the main organizer of SV-COMP and a leading proponent of software verification witnesses. These make him the most cited author in this thesis (excluding myself and my co-authors). Thus, I am incredibly honored to have Dirk review my thesis in the midst of organizing SV-COMP 2026. His careful review was worth the wait, as it revealed numerous small but important issues.

I am thankful to the VMCAI 2024 program committee who gave the Best Paper Award to our paper on witness-guided abstract interpretation. It was an unexpected but pleasant surprise to gain such a recognition for the central topic of my thesis.

I am also thankful to everyone who has contributed to GOBLINT over the years, both before I started my PhD and during it. The former includes Vesal and Kalmer, who developed GOBLINT during its early years. The latter includes the aforementioned PhD students of Helmut, who co-developed GOBLINT, and their own students, who contributed substantial features to GOBLINT. The increasing success of GOBLINT in SV-COMP would not have been possible without them.

This work was supported by the European Union and the Estonian Research Council via projects PSG61, TEM-TA119 and PRG2764.

DATA AVAILABILITY STATEMENT

Chapter 3. All data relating to SV-COMP overall and GOBLINT participations can be found via Table 2:

- The benchmarks, scripts, results and witnesses are available as described in the competition reports and on the competition website.[69] The latter also hosts the results directly for easy access.
- The GOBLINT binaries are available as Zenodo artifacts. The artifacts have been awarded the *Available*, *Functional* and *Reusable* badges via SV-COMP.
- The GOBLINT source codes are available at its GitHub repository under the GIT tags corresponding to the versions.

Chapter 4. The benchmarks, tools, scripts and raw results for Sections 4.7.1 and 4.7.2 are available as a Zenodo artifact [Saa+23d]. The artifact has been awarded the *Available* and *Functional* badges.

The benchmarks, scripts, results and witnesses for Section 4.7.3 are available via SV-COMP 2024, as described above. The GOBLINT VALIDATOR binary and source code are the same as for GOBLINT in SV-COMP 2024.

Chapter 5. The benchmarks, GOBLINT source code and scripts for Section 5.7.1 are available as a Zenodo artifact [Sch+21c]. The artifact has been awarded the *Available*, *Validated* and *Extensible* badges.

The benchmarks, GOBLINT source code and scripts for Section 5.7.2 are available as a Zenodo artifact [Sch+23d]. The artifact has been awarded the *Available* and *Functional* badges.

Chapter 6. The benchmarks, tools, scripts, raw results and witnesses for Section 6.4 are available as part of a Zenodo artifact [Saa23]. This artifact corresponds to an earlier revision of Chapter 4 [Saa+23d]. The relevant parts are in the `mt` directory, as opposed to the `st-*` directories.

The benchmarks, tools, scripts, results and witnesses for Section 6.8 are available as a Zenodo artifact [Erh+24a]. The artifact has been awarded the *Available*, *Functional* and *Reusable* badges.

[69] `https://sv-comp.sosy-lab.org`

SISUKOKKUVÕTE

Lõimeviisilise programmianalüüsi õigsustõendid

Igaüks tahab, et tarkvara oleks kiire ja *lihtsalt töötaks* (õigesti), st ei sisaldaks vigu. Kiiruse saavutamiseks võib kasutada mitmelõimelisust, mis võimaldab programmil täita mitut operatsiooni korraga. Kuna lõimede ajaplaanimine (ingl *scheduling*) on mittedeterministlik, siis on mitmelõimeliste programmide kirjutamine eriti veaohtlik. Vigade leidmiseks võib kasutada testimist, kuid tihti ei saa sellega tagada täielikku veatust. Lisaks raskendab mitmelõimeliste programmide testimist ettearvamatu ajaplaanimine. Korrektsuse garanteerimiseks saab kasutada formaalset verifitseerimist, mis mitmelõimelise tarkvara arendamisel on eriti oluline.

Verifitseerimist on võimalik automatiseerida, kasutades staatilist analüsaatorit – teist programmi, mis üritab lähtekoodi põhjal automaatselt tõestada tarkvara korrektsust. Automaatsed tarkvara verifitseerijad on aga ka ise keerukad programmid, mis võivad samuti olla vigased ja seeläbi anda ebakorrektseid tulemusi. Suurendamaks oma järelduste usaldusväärsust, võivad verifitseerijad luua tõendeid (ingl *witnesses*), mis paljastavad nende arutluskäigu analüüsitud programmi kohta. Sellised tõendusobjektid võimaldavad inimesel analüüsi tulemusi inspekteerida ja kinnitada või tagasi lükata. Sedagi võib teha automaatselt, kasutades tõendite valideerijat – kolmandat programmi, mis kontrollib tõendi kehtivust. Kahjuks puudusid varem verifitseerimistõendid mitmelõimeliste programmide korrektsuse põhjendamiseks. Selle doktoritöö eesmärk ongi täita see uurimislünk ning kasutada tõendeid, et muuta mitmelõimeliste programmide analüüs usaldusväärseks, kuid ka kiireks ja täpseks.

Samaaegsed lõimede täitmised võivad omavahel ajaliselt vahelduda (ingl *interleave*) eksponentsiaalselt paljudel viisidel. Seega pole pärismaailma programmide puhul nende ammendav ükshaaval analüüs võimalik. Paremini skaleeruvaks alternatiiviks on lõimeviisiline ehk lõimlõimeline (ingl *thread-modular*) lähenemine, mille korral lõimi analüüsitakse eraldi ja nendevahelisi mõjusid rakendatakse abstraheeritult. Üks sedalaadi meetodeid on abstraktne interpretatsioon, mis on edukas nii pärismaailma tarkvara kui ka mitmelõimeliste programmide verifitseerimises. Doktoritöös uuritaksegi kõigepealt täpsemalt lõimeviisilist abstraktset interpretatsiooni. Kahte olemas olnud lähenemist võrreldes leiti, et need on täpsuse poolest omavahel võrreldamatud, st leidub juhte, kus kumbki on teisest täpsem. Sellest tulenevalt kavandati uued, varasematest täpsemad lõimlõimelised analüüsid.

Verifitseerimistõendid peavad mitte ainult põhjendama verifitseerija tulemust analüüsitud programmi kohta, vaid võiks ka lihtsustada valideerijal sama järelduseni jõudmist. Mõningaid programmianalüüsi meetodeid, näiteks mudelkontrolli, saab tõendite abil muuta kiiremaks ja täpsemaks, ent sama pole teada abstraktse interpretatsiooni kohta. Teise aspektina uuritaksegi doktoritöös, kuidas abstraktne interpretatsioon võib kasu saada tõenditest, nii jõudluse kui ka täpsuse seisukohalt. Selleks esitatakse uudne ebaeeldamise (ingl *unassume*) operaator, millega saab

kontrollitult ebatäpsust sisestades analüüsi suunata. Uue operaatori abil arendatakse esimene abstraktsel interpretatsioonil põhinev tõendite valideerija.

Lõpuks ühendatakse doktoritöös mainitud kaks suunda ja pakutakse välja õigsustõendid mitmelõimelistele programmidele, täites seega uurimislünga. Esitatakse kahte sorti lõimeviisilised tõendid, mis vastavad lokaalsele ja globaalsele vaatele mitmelõimelisusest, ning näidatakse eksperimentaalselt, et nende abil saab suurendada mitmelõimeliste programmide analüüsi usaldusväärsust, kiirust ja täpsust.

Kõik ideed on realiseeritud staatilise analüüsi raamistikus GOBLINT, mis põhineb abstraktsel interpretatsioonil. Seda on töö käigus järjekindlalt parandatud ja täiendatud, mistõttu pakub doktoritöö ka uusima ülevaate sellest raamistikust ja selle arengust. Muu hulgas tõi kogu töö tõenditega välja GOBLINTis olnud vigu ja nende kõrvaldamine on analüsaatorit üleüldiselt parandanud. Viimase viie aasta jooksul on GOBLINT edukalt esinenud tarkvara verifitseerimise võistlusel SV-COMP, demonstreerides järjekindlalt oma korrektsust.

Kokkuvõttes edendatakse doktoritöö raames tipptasemel abstraktset interpretatsiooni ja verifitseerimistõendeid. Tõendid võimaldavad välja töötada paremaid staatilisi analüsaatoreid, mis omakorda aitavad tarkvaraarendajatel kirjutada veatuid mitmelõimelisi programme.

CURRICULUM VITAE

Personal Data

Name:	Simmo Saan
Date of Birth:	1996-03-03
Citizenship:	Estonia
Languages:	Estonian, English, German
Homepage:	`https://sim642.eu`
Email:	`simmo.saan@ut.ee / simmo.saan@gmail.com`
ORCID:	`0000-0003-4553-1350`

Education

2020–2026	University of Tartu, PhD in Computer Science
2018–2020	University of Tartu, MSc in Computer Science (*cum laude*)
2015–2018	University of Tartu, BSc in Computer Science (*cum laude*)
2012–2015	Hugo Treffner Gymnasium
2003–2012	Tartu Kivilinna Gymnasium

Employment

2022–2026	University of Tartu, Junior Research Fellow of Programming Languages
2019–2022	University of Tartu, Programmer
2017–2019	University of Tartu, Teaching Assistant

Scientific Work

Main fields of interest:

- Abstract interpretation
- Concurrency
- Verification witnesses

Awards:

- VMCAI 2024 Best Paper Award
- ECOOP 2024 Distinguished Artifact Reviewer Award

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Simmo Saan
Sünniaeg:	1996-03-03
Kodakondsus:	Eesti
Keelteoskus:	Eesti, inglise, saksa
Koduleht:	`https://sim642.eu`
E-post:	`simmo.saan@ut.ee / simmo.saan@gmail.com`
ORCID:	`0000-0003-4553-1350`

Haridus

2020–2026	Tartu Ülikool, doktorikraad informaatikas
2018–2020	Tartu Ülikool, magistrikraad informaatikas (*cum laude*)
2015–2018	Tartu Ülikool, bakalaureusekraad informaatikas (*cum laude*)
2012–2015	Hugo Treffneri Gümnaasium
2003–2012	Tartu Kivilinna Gümnaasium

Teenistuskäik

2022–2026	Tartu Ülikool, programmeerimiskeelte nooremteadur
2019–2022	Tartu Ülikool, programmeerija
2017–2019	Tartu Ülikool, õppeassistent

Teadustegevus

Peamised uurimisvaldkonnad:

- Abstraktne interpretatsioon
- Konkurrentsus
- Verifitseerimistõendid

Auhinnad:

- VMCAI 2024 parima artikli auhind
- ECOOP 2024 silmapaistva artefaktide retsensendi auhind

LIST OF ORIGINAL PUBLICATIONS

Publications Included in the Thesis

I **Simmo Saan**, Michael Schwarz, Kalmer Apinis, Julian Erhard, Helmut Seidl, Ralf Vogler, and Vesal Vojdani. "GOBLINT: Thread-Modular Abstract Interpretation Using Side-Effecting Constraints (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 12652. Springer, Mar. 2021, pp. 438–442. DOI: 10.1007/978-3-030-72013-1_28.

II Michael Schwarz, **Simmo Saan**, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. "Improving Thread-Modular Abstract Interpretation". In: *Static Analysis*. LNCS 12913. Springer, Oct. 2021, pp. 359–383. DOI: 10.1007/978-3-030-88806-0_18.

III **Simmo Saan**, Michael Schwarz, Julian Erhard, Manuel Pietsch, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. "GOBLINT: Autotuning Thread-Modular Abstract Interpretation (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 2. LNCS 13994. Springer, Apr. 2023, pp. 547–552. DOI: 10.1007/978-3-031-30820-8_34.

IV Michael Schwarz, **Simmo Saan**, Helmut Seidl, Julian Erhard, and Vesal Vojdani. "Clustered Relational Thread-Modular Abstract Interpretation with Local Traces". In: *Programming Languages and Systems*. LNCS 13990. Springer, Apr. 2023, pp. 28–58. DOI: 10.1007/978-3-031-30044-8_2.

V **Simmo Saan**, Michael Schwarz, Julian Erhard, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. "Correctness Witness Validation by Abstract Interpretation". In: *Verification, Model Checking, and Abstract Interpretation*. Vol. 1. LNCS 14499. Springer, Jan. 2024, pp. 74–97. DOI: 10.1007/978-3-031-50524-9_4.

VI **Simmo Saan**, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl. "GOBLINT VALIDATOR: Correctness Witness Validation by Abstract Interpretation (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 14572. Springer, Apr. 2024, pp. 335–340. DOI: 10.1007/978-3-031-57256-2_17.

VII **Simmo Saan**, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl. "GOBLINT: Abstract Interpretation for Memory Safety and Termination (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3. LNCS 14572. Springer, Apr. 2024, pp. 381–386. DOI: 10.1007/978-3-031-57256-2_25.

VIII Julian Erhard, Manuel Bentele, Matthias Heizmann, Dominik Klumpp, **Simmo Saan**, Frank Schüssele, Michael Schwarz, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. "Correctness Witnesses for Concurrent Programs: Bridging the Semantic Divide with Ghosts". In: *Verification, Model Checking, and Abstract Interpretation*. Vol. 1. LNCS 15529. Springer, Jan. 2025, pp. 74–100. DOI: 10.1007/978-3-031-82700-6_4.

Publications Not Included in the Thesis

IX Michael Schwarz, Julian Erhard, Vesal Vojdani, **Simmo Saan**, and Helmut Seidl. "When Long Jumps Fall Short: Control-Flow Tracking and Misuse Detection for Non-local Jumps in C". In: *State Of the Art in Program Analysis*. ACM, June 2023, pp. 20–26. DOI: 10.1145/3589250.3596140.

X Karoliine Holter, Juhan Oskar Hennoste, **Simmo Saan**, Patrick Lam, and Vesal Vojdani. "Abstract Debugging with GobPie". In: *Future Debugging Techniques*. ACM, Sept. 2024, pp. 32–33. DOI: 10.1145/3678720.3685320.

XI Julian Erhard, Michael Schwarz, Vesal Vojdani, **Simmo Saan**, and Helmut Seidl. "When long jumps fall short: control-flow tracking and misuse detection for nonlocal jumps in C. Extended version". In: *International Journal on Software Tools for Technology Transfer* 26.5 (Oct. 2024), pp. 589–605. DOI: 10.1007/s10009-024-00764-z.

XII Karoliine Holter, Juhan Oskar Hennoste, Patrick Lam, **Simmo Saan**, and Vesal Vojdani. "Abstract Debuggers: Exploring Program Behaviors using Static Analysis Results". In: *New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, Oct. 2024, pp. 130–146. DOI: 10.1145/3689492.3690053.

XIII Julian Erhard, **Simmo Saan**, Sarah Tilscher, Michael Schwarz, Karoliine Holter, Vesal Vojdani, and Helmut Seidl. "Interactive abstract interpretation: reanalyzing multithreaded C programs for cheap". In: *International Journal on Software Tools for Technology Transfer* 26.6 (Dec. 2024), pp. 647–667. DOI: 10.1007/s10009-024-00768-9.

XIV Karoliine Holter, **Simmo Saan**, Patrick Lam, and Vesal Vojdani. "Sound Static Data Race Verification for C: Is the Race Lost?" In: *ACM Transactions on Programming Languages and Systems* 47.3 (Sept. 2025), pp. 11:1–11:40. DOI: 10.1145/3732933.

DISSERTATIONES INFORMATICAE PREVIOUSLY PUBLISHED IN DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

19. **Helger Lipmaa.** Secure and efficient time-stamping systems. Tartu, 1999, 56 p.
22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.
23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.
24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.
27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 lk.
29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.
45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.
49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.
53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007, 170 p.
55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.
56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.
59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.
61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.
62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.
64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.
66. **Mark Fišel.** Optimizing Statistical Machine Translation via Input Modification. Tartu 2011, 104 p.
67. **Margus Niitsoo**. Black-box Oracle Separation Techniques with Applications in Time-stamping. Tartu 2011, 174 p.
71. **Siim Karus.** Maintainability of XML Transformations. Tartu 2011, 142 p.
72. **Margus Treumuth.** A Framework for Asynchronous Dialogue Systems: Concepts, Issues and Design Aspects. Tartu 2011, 95 p.
73. **Dmitri Lepp.** Solving simplification problems in the domain of exponents, monomials and polynomials in interactive learning environment T-algebra. Tartu 2011, 202 p.

74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.
77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.
78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.
79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.
81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.
83. **Dan Bogdanov.** Sharemind: programmable secure computations with practical applications. Tartu 2013, 191 p.
84. **Jevgeni Kabanov.** Towards a more productive Java EE ecosystem. Tartu 2013, 151 p.
87. **Margus Freudenthal.** Simpl: A toolkit for Domain-Specific Language development in enterprise information systems. Tartu, 2013, 151 p.
90. **Raivo Kolde.** Methods for re-using public gene expression data. Tartu, 2014, 121 p.
91. **Vladimir Šor.** Statistical Approach for Memory Leak Detection in Java Applications. Tartu, 2014, 155 p.
92. **Naved Ahmed.** Deriving Security Requirements from Business Process Models. Tartu, 2014, 171 p.
94. **Liina Kamm.** Privacy-preserving statistical analysis using secure multi-party computation. Tartu, 2015, 201 p.
100. **Abel Armas Cervantes.** Diagnosing Behavioral Differences between Business Process Models. Tartu, 2015, 193 p.
101. **Fredrik Milani.** On Sub-Processes, Process Variation and their Interplay: An Integrated Divide-and-Conquer Method for Modeling Business Processes with Variation. Tartu, 2015, 164 p.
102. **Huber Raul Flores Macario.** Service-Oriented and Evidence-aware Mobile Cloud Computing. Tartu, 2015, 163 p.
103. **Tauno Metsalu.** Statistical analysis of multivariate data in bioinformatics. Tartu, 2016, 197 p.
104. **Riivo Talviste.** Applying Secure Multi-party Computation in Practice. Tartu, 2016, 144 p.
108. **Siim Orasmaa.** Explorations of the Problem of Broad-coverage and General Domain Event Analysis: The Estonian Experience. Tartu, 2016, 186 p.
109. **Prastudy Mungkas Fauzi.** Efficient Non-interactive Zero-knowledge Protocols in the CRS Model. Tartu, 2017, 193 p.
110. **Pelle Jakovits.** Adapting Scientific Computing Algorithms to Distributed Computing Frameworks. Tartu, 2017, 168 p.
111. **Anna Leontjeva.** Using Generative Models to Combine Static and Sequential Features for Classification. Tartu, 2017, 167 p.
112. **Mozhgan Pourmoradnasseri.** Some Problems Related to Extensions of Polytopes. Tartu, 2017, 168 p.

113. **Jaak Randmets.** Programming Languages for Secure Multi-party Computation Application Development. Tartu, 2017, 172 p.
114. **Alisa Pankova.** Efficient Multiparty Computation Secure against Covert and Active Adversaries. Tartu, 2017, 316 p.
116. **Toomas Saarsen.** On the Structure and Use of Process Models and Their Interplay. Tartu, 2017, 123 p.
121. **Kristjan Korjus.** Analyzing EEG Data and Improving Data Partitioning for Machine Learning Algorithms. Tartu, 2017, 106 p.
122. **Eno Tõnisson.** Differences between Expected Answers and the Answers Offered by Computer Algebra Systems to School Mathematics Equations. Tartu, 2017, 195 p.

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

1. **Abdullah Makkeh**. Applications of Optimization in Some Complex Systems. Tartu 2018, 179 p.
2. **Riivo Kikas**. Analysis of Issue and Dependency Management in Open-Source Software Projects. Tartu 2018, 115 p.
3. **Ehsan Ebrahimi**. Post-Quantum Security in the Presence of Superposition Queries. Tartu 2018, 200 p.
4. **Ilya Verenich**. Explainable Predictive Monitoring of Temporal Measures of Business Processes. Tartu 2019, 151 p.
5. **Yauhen Yakimenka**. Failure Structures of Message-Passing Algorithms in Erasure Decoding and Compressed Sensing. Tartu 2019, 134 p.
6. **Irene Teinemaa**. Predictive and Prescriptive Monitoring of Business Process Outcomes. Tartu 2019, 196 p.
7. **Mohan Liyanage.** A Framework for Mobile Web of Things. Tartu 2019, 131 p.
8. **Toomas Krips.** Improving performance of secure real-number operations. Tartu 2019, 146 p.
9. **Vijayachitra Modhukur.** Profiling of DNA methylation patterns as biomarkers of human disease. Tartu 2019, 134 p.
10. **Elena Sügis.** Integration Methods for Heterogeneous Biological Data. Tartu 2019, 250 p.
11. **Tõnis Tasa.** Bioinformatics Approaches in Personalised Pharmacotherapy. Tartu 2019, 150 p.
12. **Sulev Reisberg.** Developing Computational Solutions for Personalized Medicine. Tartu 2019, 126 p.
13. **Huishi Yin.** Using a Kano-like Model to Facilitate Open Innovation in Requirements Engineering. Tartu 2019, 129 p.
14. **Faiz Ali Shah.** Extracting Information from App Reviews to Facilitate Software Development Activities. Tartu 2020, 149 p.
15. **Adriano Augusto**. Accurate and Efficient Discovery of Process Models from Event Logs. Tartu 2020, 194 p.
16. **Karim Baghery.** Reducing Trust and Improving Security in zk-SNARKs and Commitments. Tartu 2020, 245 p.
17. **Behzad Abdolmaleki.** On Succinct Non-Interactive Zero-Knowledge Protocols Under Weaker Trust Assumptions. Tartu 2020, 209 p.
18. **Janno Siim.** Non-Interactive Shuffle Arguments. Tartu 2020, 154 p.
19. **Ilya Kuzovkin.** Understanding Information Processing in Human Brain by Interpreting Machine Learning Models. Tartu 2020, 149 p.
20. **Orlenys López Pintado.** Collaborative Business Process Execution on the Blockchain: The Caterpillar System. Tartu 2020, 170 p.
21. **Ardi Tampuu.** Neural Networks for Analyzing Biological Data. Tartu 2020, 152 p.

22. **Madis Vasser.** Testing a Computational Theory of Brain Functioning with Virtual Reality. Tartu 2020, 106 p.
23. **Ljubov Jaanuska.** Haar Wavelet Method for Vibration Analysis of Beams and Parameter Quantification. Tartu 2021, 192 p.
24. **Arnis Parsovs.** Estonian Electronic Identity Card and its Security Challenges. Tartu 2021, 214 p.
25. **Kaido Lepik.** Inferring causality between transcriptome and complex traits. Tartu 2021, 224 p.
26. **Tauno Palts.** A Model for Assessing Computational Thinking Skills. Tartu 2021, 134 p.
27. **Liis Kolberg.** Developing and applying bioinformatics tools for gene expression data interpretation. Tartu 2021, 195 p.
28. **Dmytro Fishman.** Developing a data analysis pipeline for automated protein profiling in immunology. Tartu 2021, 155 p.
29. **Ivo Kubjas.** Algebraic Approaches to Problems Arising in Decentralized Systems. Tartu 2021, 120 p.
30. **Hina Anwar.** Towards Greener Software Engineering Using Software Analytics. Tartu 2021, 186 p.
31. **Veronika Plotnikova.** FIN-DM: A Data Mining Process for the Financial Services. Tartu 2021, 197 p.
32. **Manuel Camargo.** Automated Discovery of Business Process Simulation Models From Event Logs: A Hybrid Process Mining and Deep Learning Approach. Tartu 2021, 130 p.
33. **Volodymyr Leno.** Robotic Process Mining: Accelerating the Adoption of Robotic Process Automation. Tartu 2021, 119 p.
34. **Kristjan Krips.** Privacy and Coercion-Resistance in Voting. Tartu 2022, 173 p.
35. **Elizaveta Yankovskaya.** Quality Estimation through Attention. Tartu 2022, 115 p.
36. **Mubashar Iqbal.** Reference Framework for Managing Security Risks Using Blockchain. Tartu 2022, 203 p.
37. **Jakob Mass.** Process Management for Internet of Mobile Things. Tartu 2022, 151 p.
38. **Gamal Elkoumy.** Privacy-Enhancing Technologies for Business Process Mining. Tartu 2022, 135 p.
39. **Lidia Feklistova.** Learners of an Introductory Programming MOOC: Background Variables, Engagement Patterns and Performance. Tartu 2022, 151 p.
40. **Mohamed Ragab.** Bench-Ranking: A Prescriptive Analysis Approach for Large Knowledge Graphs Query Workloads. Tartu 2022, 158 p.
41. **Mohammad Anagreh.** Privacy-Preserving Parallel Computations for Graph Problems. Tartu 2023, 181 p.
42. **Rahul Goel.** Mining Social Well-being Using Mobile Data. Tartu 2023, 104 p.

43. **Anti Ingel.** Algorithms using information theory: classification in brain-computer interfaces and characterising reinforcement-learning agents. Tartu 2023, 142 p.
44. **Shakshi Sharma.** Fighting Misinformation in the Digital Age: A Comprehensive Strategy for Characterizing, Identifying, and Mitigating Misinformation on Online Social Media Platforms. Tartu 2023, 158 p.
45. **Kristiina Rahkema.** Quality Analysis of iOS Applications with Focus on Maintainability and Security Aspects. Tartu 2023, 182 p.
46. **Ivan Slobozhan.** Studying Online Social Media Engagement in CIS Countries during Protests, Mass Demonstrations and War. Tartu 2023, 81 p.
47. **Nurlan Kerimov.** Building a catalogue of molecular quantitative trait loci to interpret complex trait associations. Tartu 2023, 248 p.
48. **Pavlo Tertychnyi.** Machine Learning Methods for Anti-Money Laundering Monitoring. Tartu 2023, 117 p.
49. **Abasi-amefon Obot Affia.** A Framework and Teaching Approach for IoT Security Risk Management. Tartu 2023, 180 p.
50. **Raimond-Hendrik Tunnel.** Video Game Design and Development Bachelor's Curriculum for Estonia. Tartu 2024, 137 p.
51. **Ahto Salumets.** Bioinformatics analysis of various aspects in immunology. Tartu 2024, 198 p.
52. **Mohammed Abdulhameed Shaif Ali.** Deep Learning Methods for Cell Microscopy Image Analysis. Tartu 2024, 143 p.
53. **Pille Pullonen-Raudvere**. Foundations of Efficient and Secure Algorithm Development for Secure Multiparty Computation. Tartu 2024, 265 p.
54. **Marili Rõõm.** Multiple approaches to learners' success and factors affecting it in computer programming MOOCs. Tartu 2024, 170 p.
55. **Shivananda Rangappa Poojara.** Design and Orchestration of Scalable, Event-Driven Serverless Data Pipelines for Internet of Things (IoT) Applications. Tartu 2024, 172 p.
56. **Hassan Abdulgaleel Hassan Salim Eldeeb.** Empowering Machine Learning Pipelines with Automated Feature Engineering. Tartu 2024, 121 p.
57. **Muhammad Uzair.** Soft decision making for agri-food 4.0. Tartu 2024, 158 p.
58. **Kirill Milintsevich.** Estimation of Depression Level from Text: Symptom-Based Approach, External Knowledge, Dataset Validity. Tartu 2024, 130 p.
59. **Maksym Del.** Multilingual and Multi-Domain Representational Patterns Across Trpansformer-Based Models. Tartu 2024, 131 p.
60. **Kristo Raun.** Adaptive Out-of-order Handling in Streaming Conformance Checking. Tartu 2024, 118 p.
61. **Toivo Vajakas.** Towards integration of mobile network data into analyzing human mobility. Tartu 2024, 103 p.
62. **Katsiaryna Lashkevich.** Data-Driven Analysis and Optimization of Waiting Times in Business Processes. Tartu 2024, 169 p.
63. **Alejandra Duque-Torres.** Classifying, Constraining and Ranking Metamorphic Relations. Tartu 2025, 159 p.

64. **Mariia Bakhtina.** A Method for Information Security and Privacy Management in Smart Solutions. Tartu 2025, 199 p.
65. **Andre Tättar.** Multilingual Machine Translation for Under-Resourced Languages. Tartu 2025, 170 p.
66. **Mahmoud Shoush.** Prescriptive Process Monitoring Under Uncertainty and Resource Constraints. Tartu 2025, 178 p.
67. **Alireza Akhavi Zadegan.** A Multimodal approach for refining Mapping and Localization by Integrating Generative AI and Pedestrian-Centric Data. Tartu 2025, 147 p.
68. **Eerik Muuli.** Automating the assessment and feedback processes in IT teaching – improving creation and maintenance from the teaching staff perspective. Tartu 2025, 196 p.
69. **Kateryna Kubrak.** Towards User-Centered Prescriptive Process Monitoring Systems. Tartu 2025, 151 p.
70. **Zhigang Yin.** Computing and Sensing in a Smart Ring. Tartu 2025, 251 p.
71. **Abdul-Rasheed Olatunji Ottun.** Practical Trustworthy Artificial Intelligence with Human Oversight. Tartu 2025, 239 p.
72. **Sander Mikelsaar.** Analysis and Optimization of Iteratively Decodable Codes. Tartu 2025, 146 p.
73. **Marharyta Domnich.** Advancing Human-Centric Counterfactual Explanations in Explainable AI. Tartu 2025, 210 p.
74. **Viacheslav Komisarenko.** Aligning Training Loss to Evaluation Metrics in Deep Learning. Tartu 2026, 165 p.
75. **Heidi Taveter.** Using Programming-Process Data of Introductory Programming Courses: Finding Solver Types, Giving Feedback, and Detecting Plagiarism. Tartu 2026, 184 p.
76. **Daniel Majoral Lopez.** Deep neural networks for microscopy images. Tartu 2026, 81 p.
77. **Mahir Gulzar.** Addressing Real-world Scenarios via Motion Prediction in Autonomous Driving. Tartu 2026, 141 p.
78. **Hele-Andra Kuulmets.** Cross-Lingual Transfer Learning and Evaluation in Low-Resource Settings. Tartu 2026, 180 p.