Correctness Witnesses for Thread-Modular Program Analysis

Simmo Saan

Supervisor: Vesal Vojdani

April 23, 2026

Motivation

Concurrency:

- *Multi-threading*
- Non-deterministic execution
- Hard to write & test

Correctness:

- Formal verification
- *Automated program analysis*
  - Thread-modular
  - Abstract interpretation

Trustworthiness:

- Bugs in analyzers
- *Witnesses*

Motivation

Concurrency:

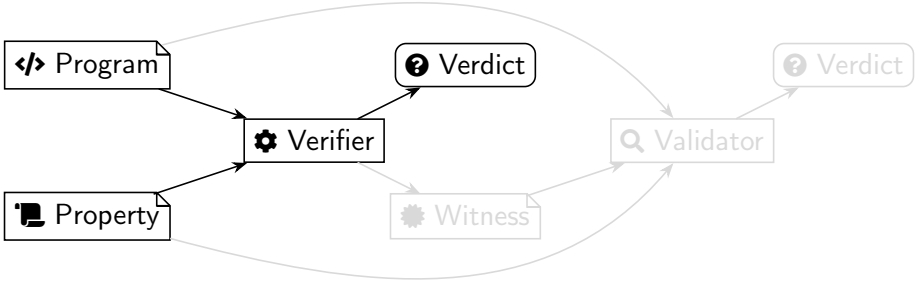
- *Multi-threading*
- Non-deterministic execution
- Hard to write & test

Correctness:

- Formal verification
- *Automated program analysis*
  - Thread-modular
  - Abstract interpretation

Trustworthiness:

- Bugs in analyzers
- *Witnesses*



Motivation

Concurrency:

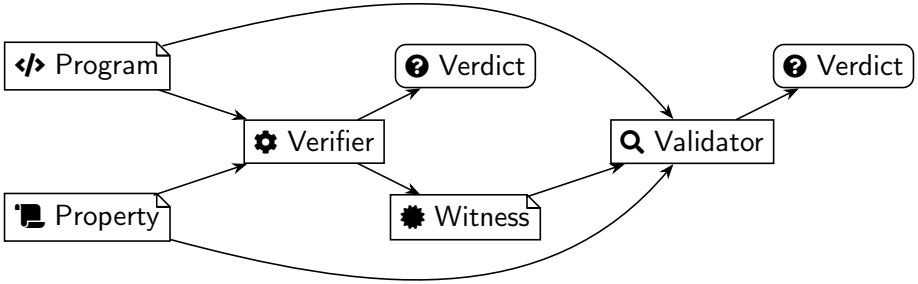
- *Multi-threading*
- Non-deterministic execution
- Hard to write & test

Correctness:

- Formal verification
- *Automated program analysis*
  - Thread-modular
  - Abstract interpretation

Trustworthiness:

- Bugs in analyzers
- *Witnesses*



Goals

Overall goal

Performant, precise and trustworthy analysis of concurrent programs (using witnesses).

Subgoals:

- 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.

Goals

Overall goal

Performant, precise and trustworthy analysis of concurrent programs (using witnesses).

Subgoals:

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.

Goals

Overall goal

Performant, precise and trustworthy analysis of concurrent programs (using witnesses).

Subgoals:

- 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.

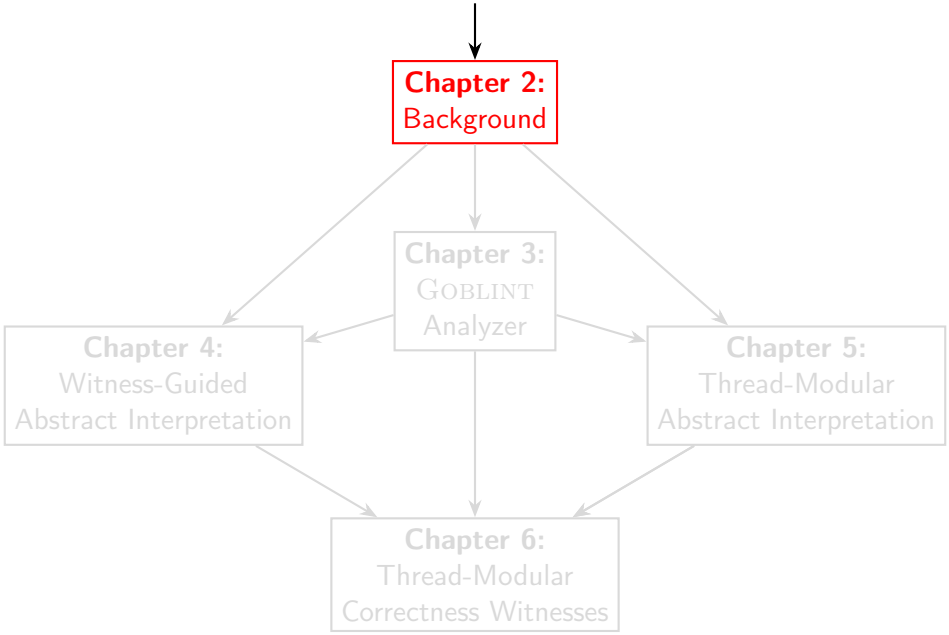
Goals

Overall goal

Performant, precise and trustworthy analysis of concurrent programs (using witnesses).

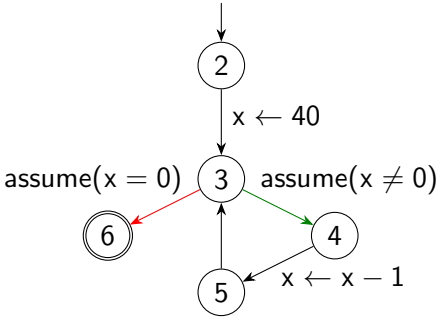
Subgoals:

- 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.


Chapter 2:
Background
Chapter 3:
GOBLINT
Analyzer
Chapter 4:
Witness-Guided
Abstract Interpretation
Chapter 5:
Thread-Modular
Abstract Interpretation
Chapter 6:
Thread-Modular
Correctness Witnesses

Abstract interpretation example

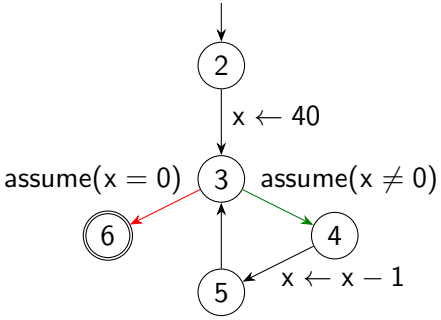
```
1 void main() {
2   int x = 40;
3   while (x != 0) {
4     x--;
5   }
6 }
```



Line	x interval	Comment
2	$[-\infty, \infty]$	
3	$[40, 40]$	
4	$[40, 40]$	
5	$[39, 39]$	
3	$[39, 40]$	$[40, 40] \sqcup [39, 39]$
4	$[39, 40]$	
5	$[38, 39]$	
⋮	⋮	Repeat 38 times
3	$[0, 40]$	$[40, 40] \sqcup [0, 39]$
4	$[1, 40]$	
5	$[0, 39]$	
3	$[0, 40]$	$[40, 40] \sqcup [0, 39]$
6	$[0, 0]$	

Abstract interpretation example

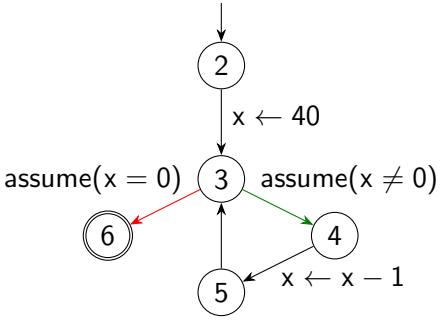
```
1 void main() {
2   int x = 40;
3   while (x != 0) {
4     x--;
5   }
6 }
```



Line	x interval	Comment
2	$[-\infty, \infty]$	
3	$[40, 40]$	
4	$[40, 40]$	
5	$[39, 39]$	
3	$[39, 40]$	$[40, 40] \sqcup [39, 39]$
4	$[39, 40]$	
5	$[38, 39]$	
⋮	⋮	Repeat 38 times
3	$[0, 40]$	$[40, 40] \sqcup [0, 39]$
4	$[1, 40]$	
5	$[0, 39]$	
3	$[0, 40]$	$[40, 40] \sqcup [0, 39]$
6	$[0, 0]$	

Abstract interpretation example

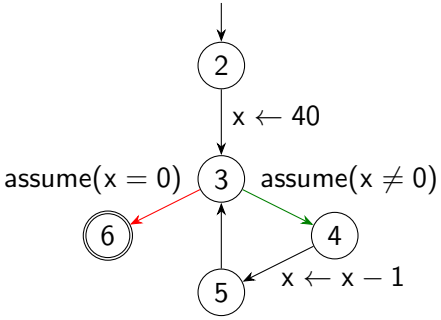
```
1 void main() {
2   int x = 40;
3   while (x != 0) {
4     x--;
5   }
6 }
```



Line	x interval	Comment
2	$[-\infty, \infty]$	
3	$[40, 40]$	
4	$[40, 40]$	
5	$[39, 39]$	
3	$[39, 40]$	$[40, 40] \sqcup [39, 39]$
4	$[39, 40]$	
5	$[38, 39]$	
⋮	⋮	Repeat 38 times
3	$[0, 40]$	$[40, 40] \sqcup [0, 39]$
4	$[1, 40]$	
5	$[0, 39]$	
3	$[0, 40]$	$[40, 40] \sqcup [0, 39]$
6	$[0, 0]$	

Abstract interpretation example

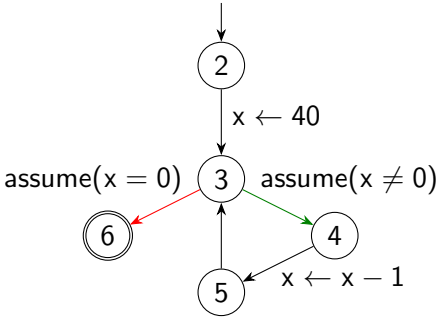
```
1 void main() {
2   int x = 40;
3   while (x != 0) {
4     x--;
5   }
6 }
```



Line	x interval	Comment
2	$[-\infty, \infty]$	
3	$[40, 40]$	
4	$[40, 40]$	
5	$[39, 39]$	
3	$[39, 40]$	$[40, 40] \sqcup [39, 39]$
4	$[39, 40]$	
5	$[38, 39]$	
⋮	⋮	Repeat 38 times
3	$[0, 40]$	$[40, 40] \sqcup [0, 39]$
4	$[1, 40]$	
5	$[0, 39]$	
3	$[0, 40]$	$[40, 40] \sqcup [0, 39]$
6	$[0, 0]$	

Abstract interpretation example

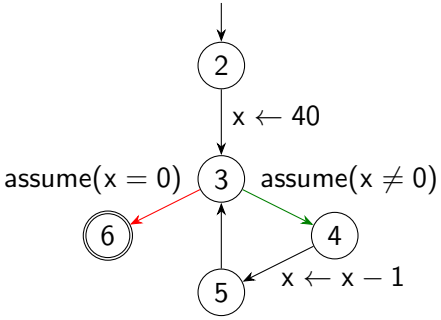
```
1 void main() {
2   int x = 40;
3   while (x != 0) {
4     x--;
5   }
6 }
```



Line	x interval	Comment
2	$[-\infty, \infty]$	
3	$[40, 40]$	
4	$[40, 40]$	
5	$[39, 39]$	
3	$[39, 40]$	$[40, 40] \sqcup [39, 39]$
4	$[39, 40]$	
5	$[38, 39]$	
⋮	⋮	Repeat 38 times
3	$[0, 40]$	$[40, 40] \sqcup [0, 39]$
4	$[1, 40]$	
5	$[0, 39]$	
3	$[0, 40]$	$[40, 40] \sqcup [0, 39]$
6	$[0, 0]$	

Abstract interpretation example

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

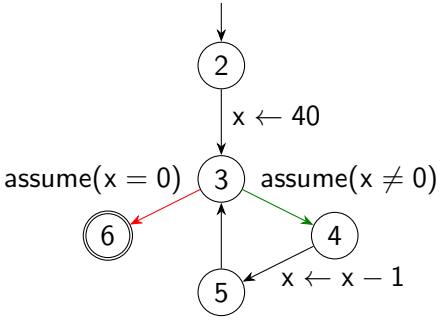


Using widening to enforce termination:

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

Abstract interpretation example

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

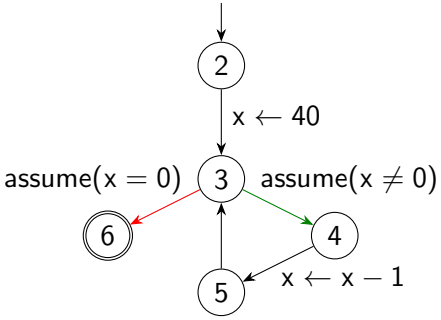


Using widening to enforce termination:

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

Abstract interpretation example

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



Using widening to enforce termination:

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

Abstract interpretation

Technique for automated program analysis:

- Introduced by Cousot & Cousot (1977)
- Mathematical framework

Strengths

- Sound — does not miss violations (by design)
- Terminating — thanks to widening
- Efficient — scales to real-world programs

Weaknesses

- Incomplete/imprecise — finds spurious violations (due to over-approximation)

Abstract interpretation formally

Set of all concrete program states S.

Abstract domain $\mathbb{D}$:

- Partial order $\sqsubseteq$
- Monotonic concretization $\gamma : \mathbb{D} \to 2^S$
- Least element $\bot$
- Greatest element $\top$
- Join $\sqcup$ and meet $\sqcap$
- Widening ∇ and narrowing Δ

Abstract semantics $[\![s]\!]^\sharp : \mathbb{D} \to \mathbb{D}$:

- $[\![x \leftarrow e]\!]^\sharp$ (assignment)
- $[\![\text{assume}(e)]\!]^\sharp$ (branching)
- $[\![\text{havoc}(V)]\!]^\sharp$ (non-det. assignment)
- ...

Abstract interpreter:

- Fixpoint iteration for loops
- Provides a mapping $\sigma : \mathcal{N} \to \mathbb{D}$ from program locations to abstract values

SV-COMP

Competition on Software Verification

Competition for automated program analyzers:

- Main organizer: Dirk Beyer
- Pioneered witnesses
- Tracks:
  - C, Java, ...
  - Verification & validation
- Annual: 15 times, since 2012
- SV-COMP 2026:
  - 36,402 verification tasks
  - 61 verifiers
  - 16 validators

Safety properties:

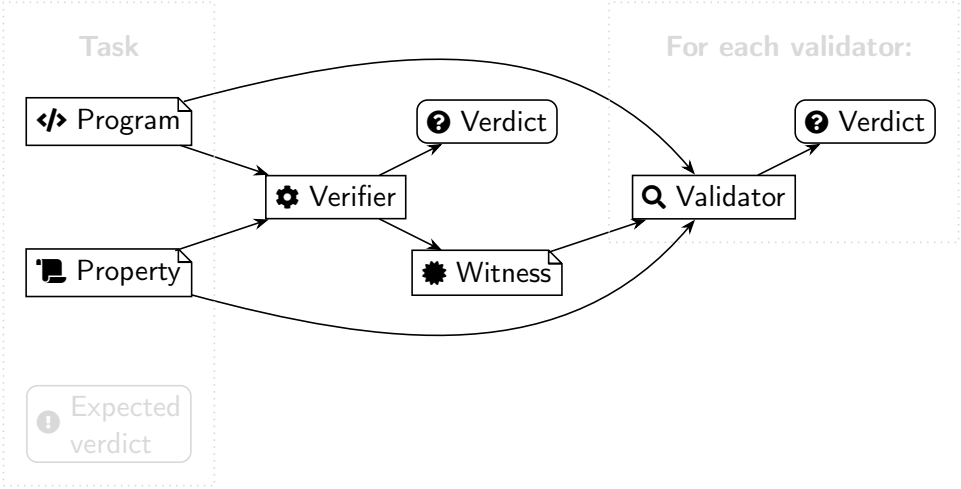
- Reachability
- Memory safety
- Memory leaks
- Overflows
- Data races

Liveness properties:

- Termination

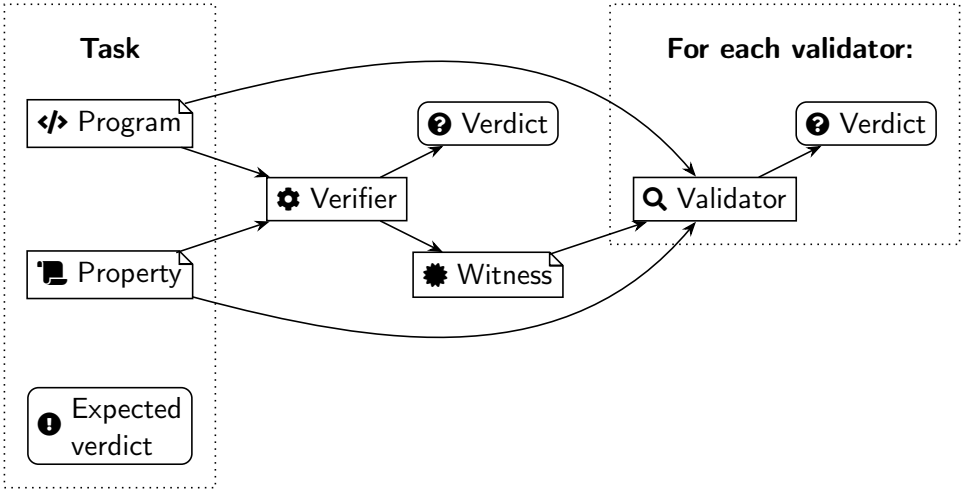
Verification track workflow

For each verifier and verification task:



Verification track workflow

For each verifier and verification task:



Witnesses

Exchangeable proof objects for program analysis results:

Violation Counterexample trace

Correctness "Should contain some hints for the proof of program correctness" – Beyer & Strejček

Program P:

- $\mathcal{N}$ – set of program locations
- $\mathcal{V}$ – set of program variables

Invariant language $\mathcal{E}$:

- C expressions, ACSL, ...
- $e_{\mathsf{true}} \in \mathcal{E}$ – always holds

Definition

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

Witness validation

Definition

A witness (W, P, Φ) is *valid* if

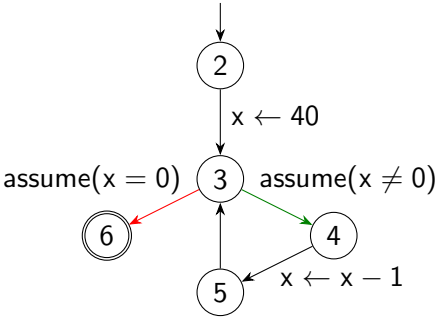
1. P satisfies the property Φ;
2. whenever the execution of P reaches the location $n \in \mathcal{N}$, the invariant $W\,n$ holds.

Tools:

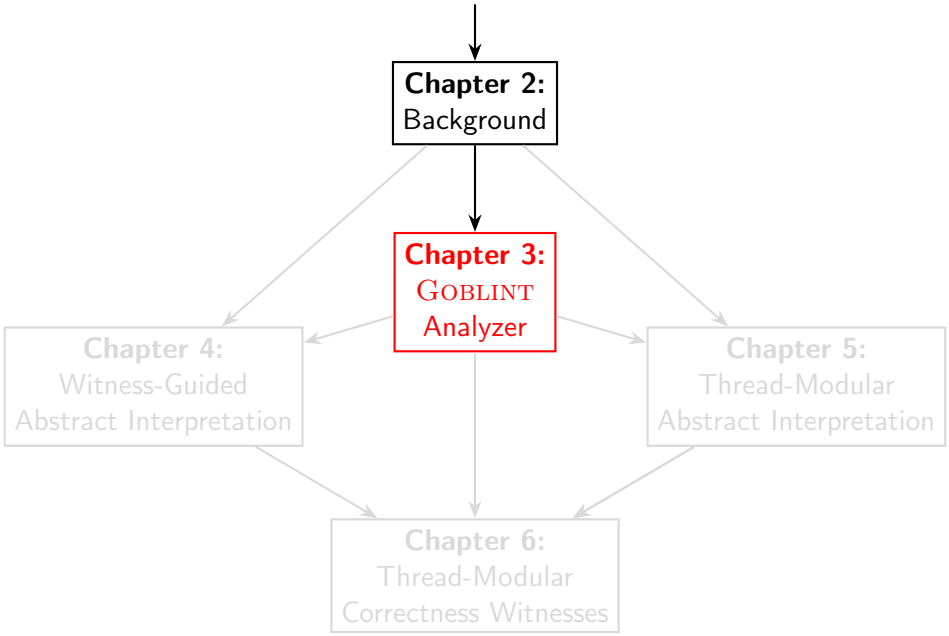
Witness validator	Attempts to prove that a witness is valid	(1 & 2)
Witness-guided verifier	Uses the witness as a guidance toward verification of Φ	(1)
Witness-only validator	Checks that the witness invariants hold at their locations	(2)

YAML witness example

```c
void main() {
  int x = 40;
  while (x != 0) {
    x--;
  }
}
```




```yaml
- entry_type: invariant_set
  content:
  - invariant:
      type: loop_invariant
      location:
        line: 3
      value: 0 <= x && x <= 40
      format: c_expression
  - invariant:
      type: location_invariant
      location:
        line: 4
      value: 1 <= x && x <= 40
      format: c_expression
```


Chapter 2:
Background
Chapter 3:
Goblint
Analyzer
Chapter 4:
Witness-Guided
Abstract Interpretation
Chapter 5:
Thread-Modular
Abstract Interpretation
Chapter 6:
Thread-Modular
Correctness Witnesses

Goblint

Static analyzer for C programs:

- Specializes in multi-threaded programs
- Implemented in OCaml
- Open source (MIT license)
- Developed at:

⋮

Verification approach:

- Abstract interpretation
- Thread-modular
- Side-effecting constraint systems

Goblint in SV-COMP

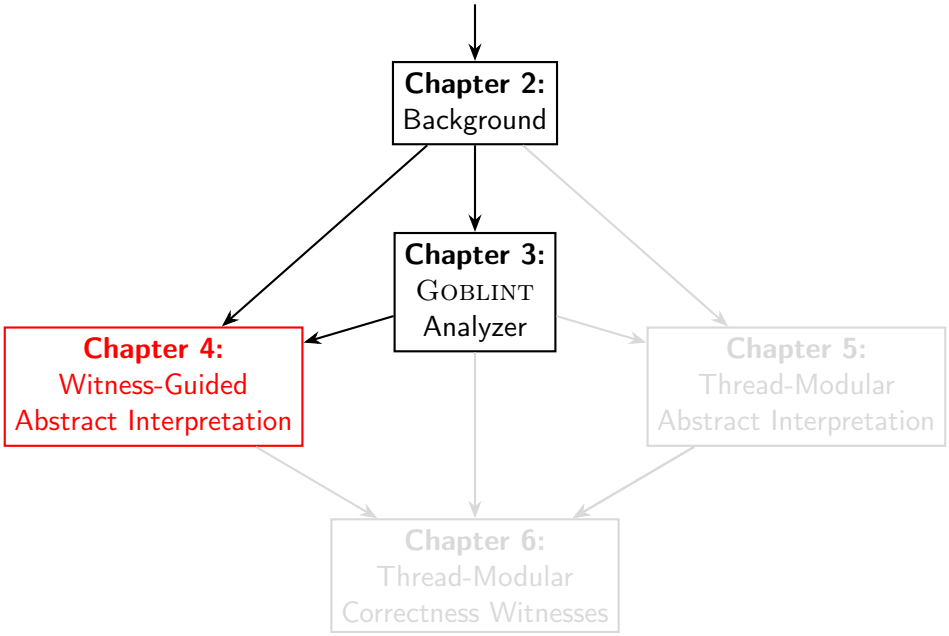
SV-COMP 2021 Data races category
SV-COMP 2022 Most correct & greenest
SV-COMP 2023 Autotuning, relational, MHP
SV-COMP 2024 Memory safety & termination
SV-COMP 2025 Simpler witness invariants
SV-COMP 2026 Sequential portfolio

Strengths

- Supports all properties
- Sound (×6)
- Fast
- 1st in data races category (×5)

Weaknesses

- No definite violations
- Imprecise


Chapter 2:
Background
Chapter 3:
GOBLINT
Analyzer
Chapter 4:
Witness-Guided
Abstract Interpretation
Chapter 5:
Thread-Modular
Abstract Interpretation
Chapter 6:
Thread-Modular
Correctness Witnesses

Witness-guided abstract interpretation

Witness validators in SV-COMP 2023:

- **Violation** Based on model checking, test execution, interpretation and SMT
- **Correctness** Based on model checking

"Remarkable gap in software-verification research" – Beyer

"There are verification results that cannot be independently confirmed, according to the state of the art in software verification."

Goal

Complement existing validators by developing an abstract-interpretation–based correctness witness validator.

Witness-guided abstract interpretation

Witness validators in SV-COMP 2023:

- **Violation** Based on model checking, test execution, interpretation and SMT
- **Correctness** Based on model checking

"Remarkable gap in software-verification research" – Beyer

"There are verification results that cannot be independently confirmed, according to the state of the art in software verification."

Goal

Complement existing validators by developing an abstract-interpretation–based correctness witness validator.

Witnesses and abstract interpretation

Abstract interpreter provides a mapping $\sigma : \mathcal{N} \to \mathbb{D}$ from locations to abstract values.

Witness validation

A witness (W, P, Φ) is validated, 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 $\gamma\,(\sigma\,n)$.

Witness-guided verification

Given a witness (W, P, Φ), the *challenge* is to simultaneously achieve the following:

1. use the invariants in W to reach a (more precise) fixpoint $\sigma_W : \mathcal{N} \to \mathbb{D}$ (in fewer iterations);
2. avoid overshooting the required property, i.e., σ_W suffices to prove Φ;
3. *not trust* the witness, i.e., σ_W should remain sound even in presence of *wrong* invariants.

Witnesses and abstract interpretation

Abstract interpreter provides a mapping $\sigma : \mathcal{N} \to \mathbb{D}$ from locations to abstract values.

Witness validation

A witness (W, P, Φ) is validated, 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 $\gamma\,(\sigma\,n)$.

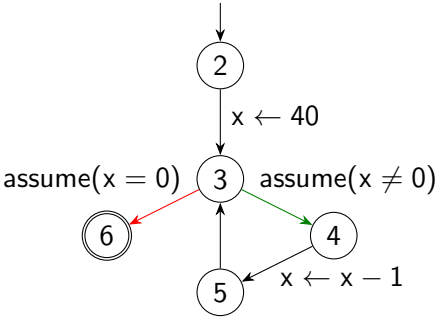
Witness-guided verification

Given a witness (W, P, Φ), the *challenge* is to simultaneously achieve the following:

1. use the invariants in W to reach a (more precise) fixpoint $\sigma_W : \mathcal{N} \to \mathbb{D}$ (in fewer iterations);
2. avoid overshooting the required property, i.e., σ_W suffices to prove Φ;
3. *not trust* the witness, i.e., σ_W should remain sound even in presence of *wrong* invariants.

Illustrative example

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



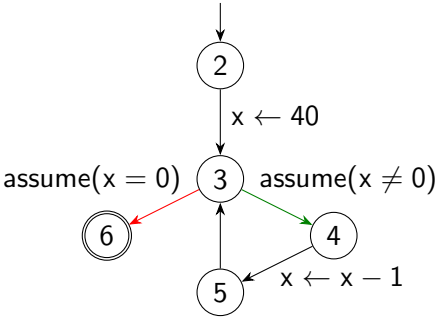
Line	x interval	Comment
2	$[-\infty, \infty]$	
3	$[40, 40]$	
4	$[40, 40]$	
5	$[39, 39]$	
3	$[-\infty, 40]$	$[40, 40] \nabla ([40, 40] \sqcup [39, 39])$
4	$[-\infty, 40]$	
5	$[-\infty, 39]$	
3	$[-\infty, 40]$	$[-\infty, 40] \nabla ([40, 40] \sqcup [-\infty, 39])$
6	$[0, 0]$	

From-scratch result

No lower bound inferred for x at loop head.

Illustrative example

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



Witness provides $0 \leq x \leq 40$ for loop head on line 3:

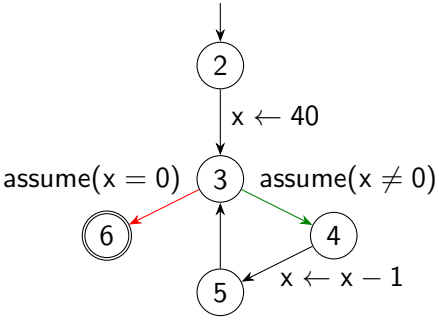
Line	x interval	Comment
2	$[-\infty, \infty]$	
3	$[0, 40]$	Relax $[40, 40]$ up to $0 \leq x \leq 40$
4	$[1, 40]$	
5	$[0, 39]$	
3	$[0, 40]$	$[0, 40] \triangledown ([40, 40] \sqcup [0, 39])$
6	$[0, 0]$	

Witness-guided result

Stronger invariant is confirmed in fewer steps.

Illustrative example

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



Witness provides $0 \leq x \leq 40$ for loop head on line 3:

Line	x interval	Comment
2	$[-\infty, \infty]$	
3	$[0, 40]$	Relax $[40, 40]$ up to $0 \leq x \leq 40$
4	$[1, 40]$	
5	$[0, 39]$	
3	$[0, 40]$	$[0, 40] \nabla ([40, 40] \sqcup [0, 39])$
6	$[0, 0]$	

Witness-guided result

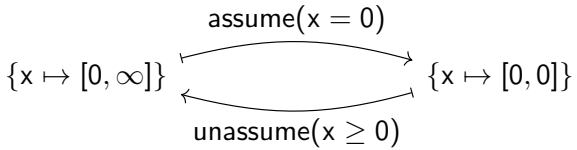
Stronger invariant is confirmed in fewer steps.

Unassume

Given a witness (W, P, Φ), insert at every location n, the statement unassume$(W\ n)$:

Concrete semantics No-op

Abstract semantics Relax the state:



Specification By example

Abstract operators $[\![\text{unassume}_V(e)]\!]^\sharp : \mathbb{D} \to \mathbb{D}$, where $e \in \mathcal{E}$, $V \subseteq \mathcal{V}$:

- *Sound* if $\gamma\, d \subseteq \gamma\, ([\![\text{unassume}_V(e)]\!]^\sharp\, d)$

Theorem

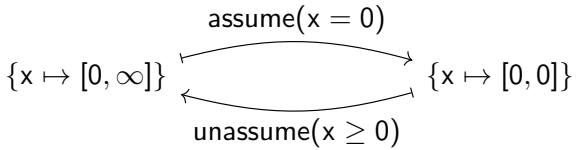
The abstract interpretation result remains sound w.r.t. the original program.

Unassume

Given a witness (W, P, Φ), insert at every location n, the statement unassume$(W\ n)$:

Concrete semantics No-op

Abstract semantics Relax the state:



Specification By example

Abstract operators $[\![\text{unassume}_V(e)]\!]^\sharp : \mathbb{D} \to \mathbb{D}$, where $e \in \mathcal{E}$, $V \subseteq \mathcal{V}$:

- *Sound* if $\gamma\, d \subseteq \gamma\, ([\![\text{unassume}_V(e)]\!]^\sharp\, d)$

Theorem

The abstract interpretation result remains sound w.r.t. the original program.

Definitions

1. Naïve unassume:
   - Uses standard assume and havoc operators
   - Succeeds for illustrative example
   - Fails with relations between V and $\mathcal{V} \setminus V$
2. Dual-narrowing unassume:
   - Uses dual-narrowing operator $\widetilde{\Delta}$
   - Fails with ambiguous pointers
3. Exploding unassume:
   - Explodes ambiguous points-to sets
   - Fails with dependent (pointer) subexpressions
4. Propagating unassume:
   - Based on backward evaluation
   - Only for non-relational domains
   - Chooses V per subexpression

Soundness

All sound by construction

Specification	1	2	3	4
S1–S3	✓	✓	✓	✓
S4–S6	✓	✓	✓	✓
S7	✗	✓	✗	✗
S8	✓	✓	✓	✓
S9	✗	✗	✓	✓
S10	✗	✗	✗	✓
S11	✗	✗	✗	✗

Definitions

1. Naïve unassume:
   - Uses standard assume and havoc operators
   - Succeeds for illustrative example
   - Fails with relations between V and $\mathcal{V} \setminus V$
2. Dual-narrowing unassume:
   - Uses dual-narrowing operator $\widetilde{\Delta}$
   - Fails with ambiguous pointers
3. Exploding unassume:
   - Explodes ambiguous points-to sets
   - Fails with dependent (pointer) subexpressions
4. Propagating unassume:
   - Based on backward evaluation
   - Only for non-relational domains
   - Chooses V per subexpression

Soundness

All sound by construction

Specification	1	2	3	4
S1–S3	✓	✓	✓	✓
S4–S6	✓	✓	✓	✓
S7	✗	✓	✗	✗
S8	✓	✓	✓	✓
S9	✗	✗	✓	✓
S10	✗	✗	✗	✓
S11	✗	✗	✗	✗

Definitions

1. Naïve unassume:
   - Uses standard assume and havoc operators
   - Succeeds for illustrative example
   - Fails with relations between V and $\mathcal{V} \setminus V$
2. Dual-narrowing unassume:
   - Uses dual-narrowing operator $\widetilde{\Delta}$
   - Fails with ambiguous pointers
3. Exploding unassume:
   - Explodes ambiguous points-to sets
   - Fails with dependent (pointer) subexpressions
4. Propagating unassume:
   - Based on backward evaluation
   - Only for non-relational domains
   - Chooses V per subexpression

Soundness

All sound by construction

Specification	1	2	3	4
S1–S3	✓	✓	✓	✓
S4–S6	✓	✓	✓	✓
S7	✗	✓	✗	✗
S8	✓	✓	✓	✓
S9	✗	✗	✓	✓
S10	✗	✗	✗	✓
S11	✗	✗	✗	✗

Definitions

1. Naïve unassume:
   - Uses standard assume and havoc operators
   - Succeeds for illustrative example
   - Fails with relations between V and $\mathcal{V} \setminus V$
2. Dual-narrowing unassume:
   - Uses dual-narrowing operator $\widetilde{\Delta}$
   - Fails with ambiguous pointers
3. Exploding unassume:
   - Explodes ambiguous points-to sets
   - Fails with dependent (pointer) subexpressions
4. Propagating unassume:
   - Based on backward evaluation
   - Only for non-relational domains
   - Chooses V per subexpression

Soundness

All sound by construction

Specification	1	2	3	4
S1–S3	✓	✓	✓	✓
S4–S6	✓	✓	✓	✓
S7	✗	✓	✗	✗
S8	✓	✓	✓	✓
S9	✗	✗	✓	✓
S10	✗	✗	✗	✓
S11	✗	✗	✗	✗

Definitions

1. Naïve unassume:
   - Uses standard assume and havoc operators
   - Succeeds for illustrative example
   - Fails with relations between V and $\mathcal{V} \setminus V$
2. Dual-narrowing unassume:
   - Uses dual-narrowing operator $\widetilde{\Delta}$
   - Fails with ambiguous pointers
3. Exploding unassume:
   - Explodes ambiguous points-to sets
   - Fails with dependent (pointer) subexpressions
4. Propagating unassume:
   - Based on backward evaluation
   - Only for non-relational domains
   - Chooses V per subexpression

Soundness

All sound by construction

Specification	1	2	3	4
S1–S3	✓	✓	✓	✓
S4–S6	✓	✓	✓	✓
S7	✗	✓	✗	✗
S8	✓	✓	✓	✓
S9	✗	✗	✓	✓
S10	✗	✗	✗	✓
S11	✗	✗	✗	✗

Goblint Validator

Extension of Goblint for validation of YAML correctness witnesses.

Validation approach:

Analysis Unassume witness invariants:

- Non-relational domains – propagating unassume
- Relational domains – dual-narrowing unassume

Post-processing Check witness invariants

Strengths

- Inherited from Goblint
- Supports all properties

Weaknesses

- Inherited from Goblint
- Cannot reject correctness witnesses

Evaluation pillars

Same-framework consistency

Witnesses produced by a verifier should be validated by a validator using the *same framework*.

Cross-framework validation

Witnesses produced by a verifier should be validated by a validator using a *different framework*.

Content-effort dependence

"[The] effort and feasibility of validation depends on witness contents" – Beyer et al.:

Performance Validation of a witness uses *less resources* than it took to produce the witness.

Precision A validator *can validate* a witness for a program which *could not be verified* by a verifier using the same framework as the validator.

Content-effort dependence (precision)

Author(s)	Example	GOBLINT w/o witness	GOBLINT w/ witness from: Manual	CPACHECKER	UAUTOMIZER
Miné	4.6	✗	✓	✓✗	✓
	4.7	✗	✓	✓✗	✓
	4.8	✓	✓	✓	✓
	4.10	✓	✓	✓✗	✓
Halbwachs & Henry	1.b	✓	✓	✓✗	✓✗
	2.b	✗	✓	✓✗	✓
	3	✗	✓	✓✗	✓✗
Boutonnet & Halbwachs	1 (polyhedra)	✗	✓	✓✗	✓✗
	3	✗	✓	✗	✓
	"additional"	✗	✓	✗	✓
Amato & Scozzari	`hybrid`	✗	✓	✓✗	✓

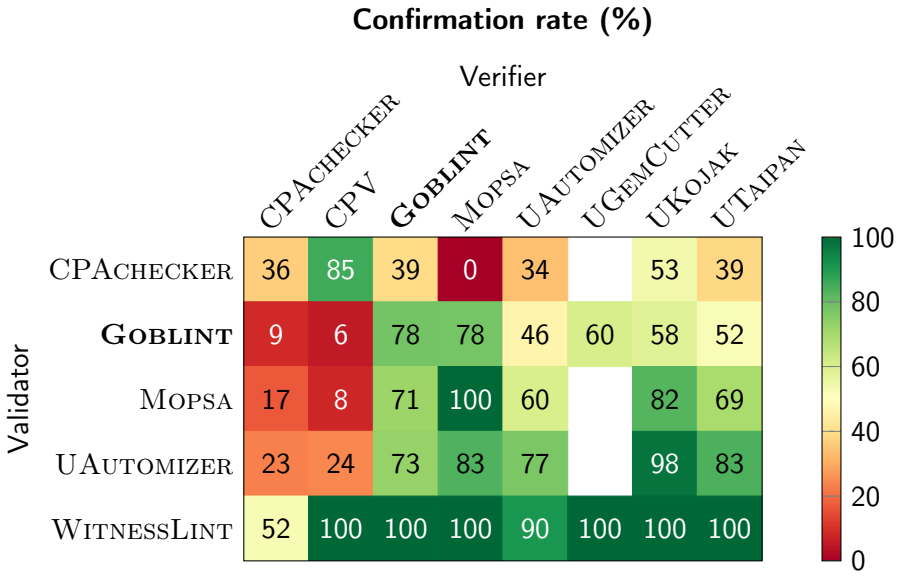
Content-effort dependence (performance)

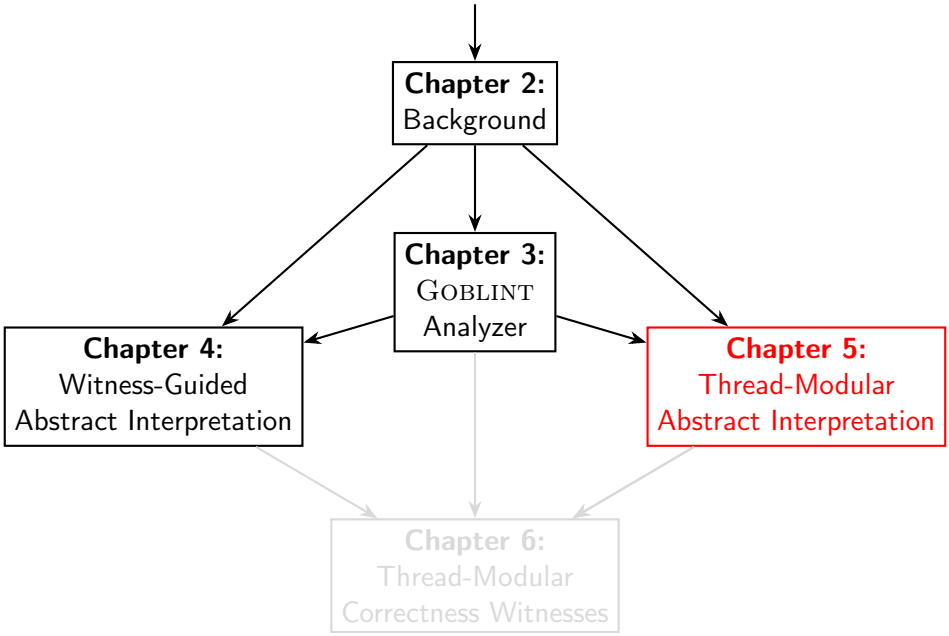
Setup:

- *Evals* – transfer function evaluations (independent of hardware)
- Manual witnesses

		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%)

SV-COMP 2024 evaluation




Chapter 2:
Background
Chapter 3:
GOBLINT
Analyzer
Chapter 4:
Witness-Guided
Abstract Interpretation
Chapter 5:
Thread-Modular
Abstract Interpretation
Chapter 6:
Thread-Modular
Correctness Witnesses

Thread-modular abstract interpretation

Non-deterministic execution of a multi-threaded program yields an *interleaving* of the threads.

Problem

Exponentially many interleavings — exhaustive analysis unscalable

Solution

Thread-modularity — analyze threads separately and propagate interferences

Setting:

- Synchronization with mutexes
- Non-relational value analysis
- Side-effecting constraint systems

Baseline analysis:

- Data flow from write to read
- Ignores synchronization

Illustrative example

```
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 = 1;
12     assert(g == 1);
13     g = 0;
14   unlock(m);
15 }
```

Baseline analysis

- g: $\{0, 1\}$

Illustrative example

```
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 = 1;
12     assert(g == 1);
13     g = 0;
14   unlock(m);
15 }
```

Baseline analysis

- g: $\{0, 1\}$

Prior analyses

Miné's analysis (2012):

- Data flow:
  - from write to read
  - from unlock to lock
- Refined by *background lockset*
- Excluded by mutual exclusion
- More general than our setting

Our analysis (Vojdani 2010):

- Protecting mutexes per global
- Data flow from unlock to lock
- *Privatization* of protected globals
- Used in GOBLINT since 2011

Illustrative example

```
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 = 1;
12     assert(g == 1);
13     g = 0;
14   unlock(m);
15 }
```

Baseline analysis

- g: $\{0, 1\}$

Miné's analysis

- g with {m}: $\{0, 1\}$
- g when m unlocked with $\varnothing$: $\{0\}$

Our analysis

- g protected by {m}
- g when unprotected: $\{0\}$

Illustrative example

```
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 = 1;
12     assert(g == 1);
13     g = 0;
14   unlock(m);
15 }
```

Baseline analysis

- g: $\{0, 1\}$

Miné's analysis

- g with {m}: $\{0, 1\}$
- g when m unlocked with $\varnothing$: $\{0\}$

Our analysis

- g protected by {m}
- g when unprotected: $\{0\}$

Prior analyses

Miné's analysis (2012):

- Data flow:
  - from write to read
  - from unlock to lock
- Refined by *background lockset*
- Excluded by mutual exclusion
- More general than our setting

Our analysis (Vojdani 2010):

- Protecting mutexes per global
- Data flow from unlock to lock
- *Privatization* of protected globals
- Used in GOBLINT since 2011

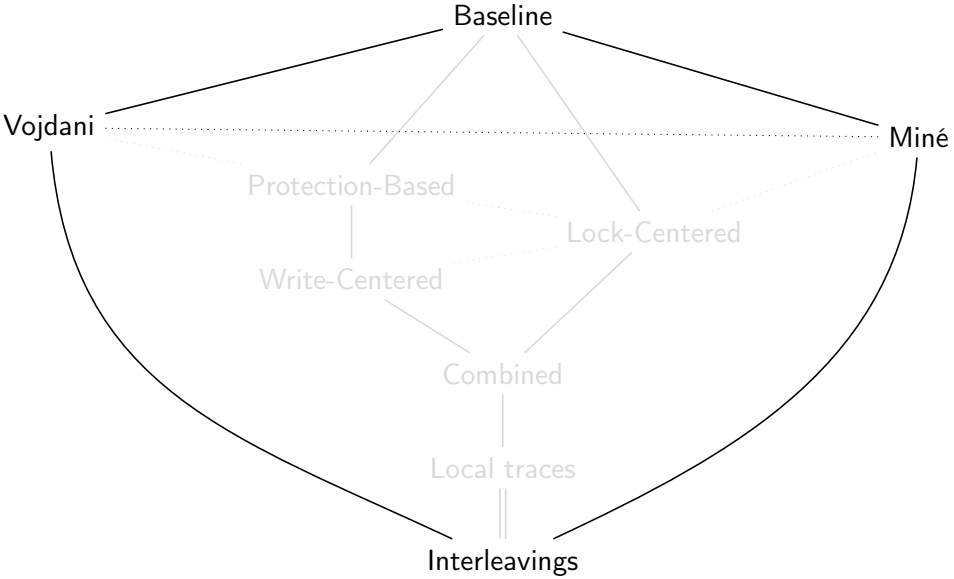
Incomparable

Neither subsumes the other in terms of precision.

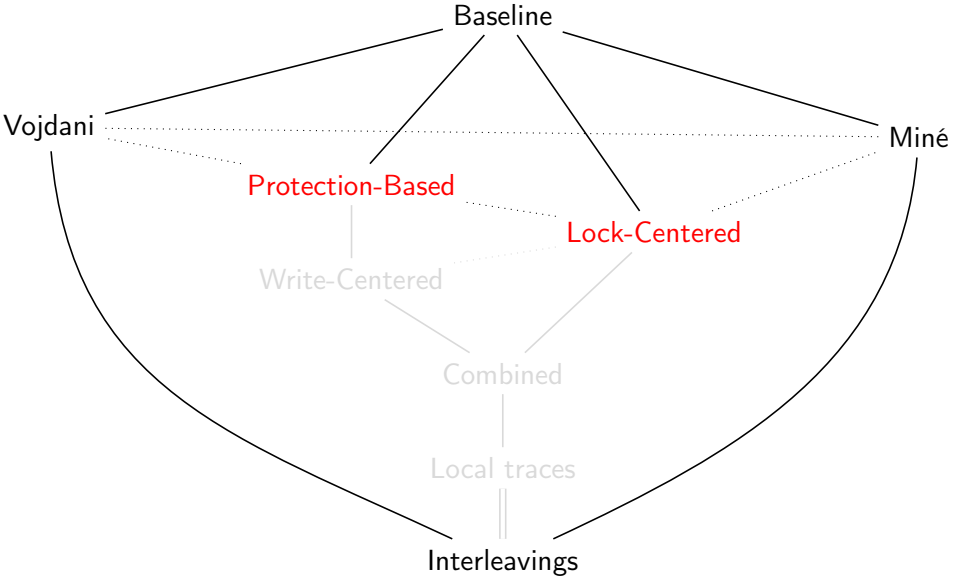
Deficiencies

1. Eager reading
2. Republishing
3. (Poor dependency structure)

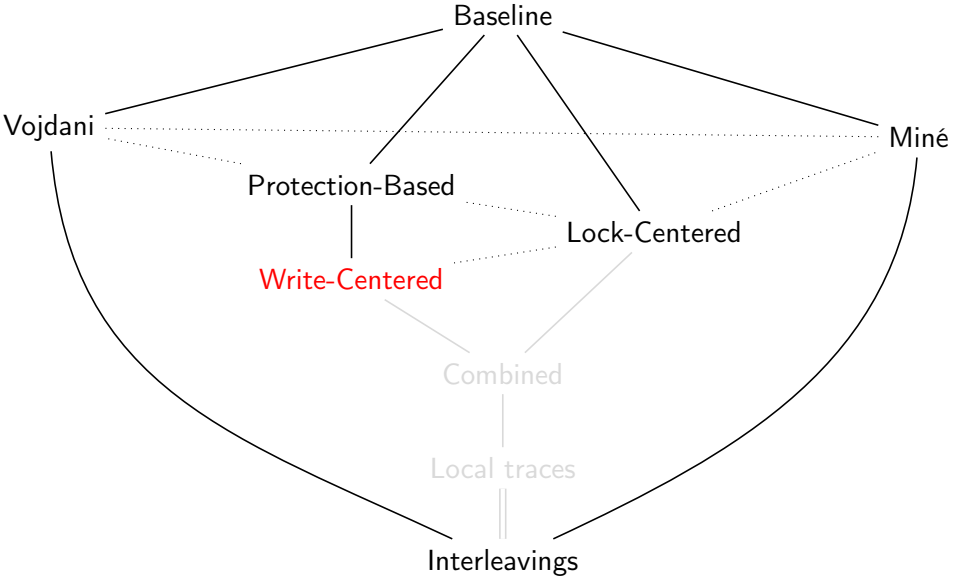
Improved analyses



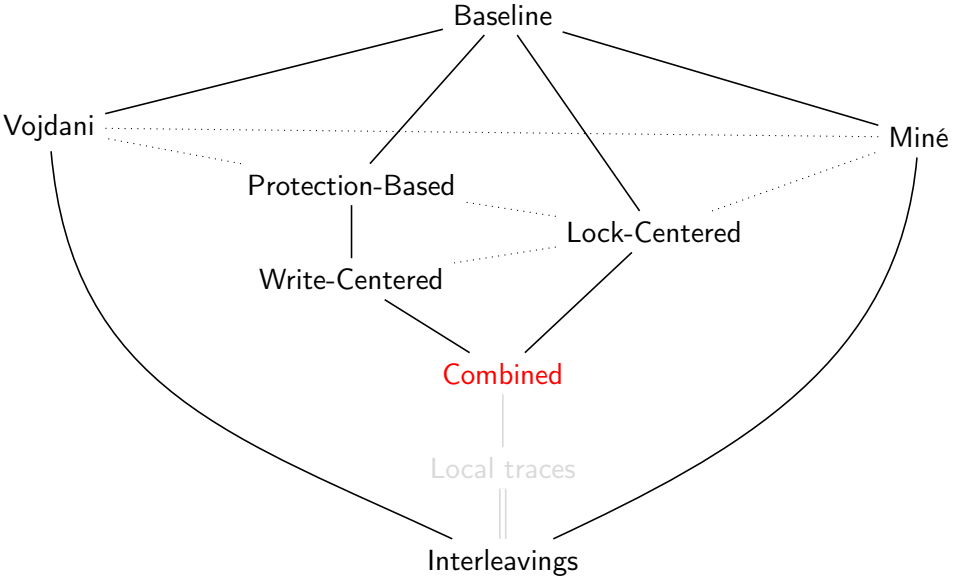
Improved analyses



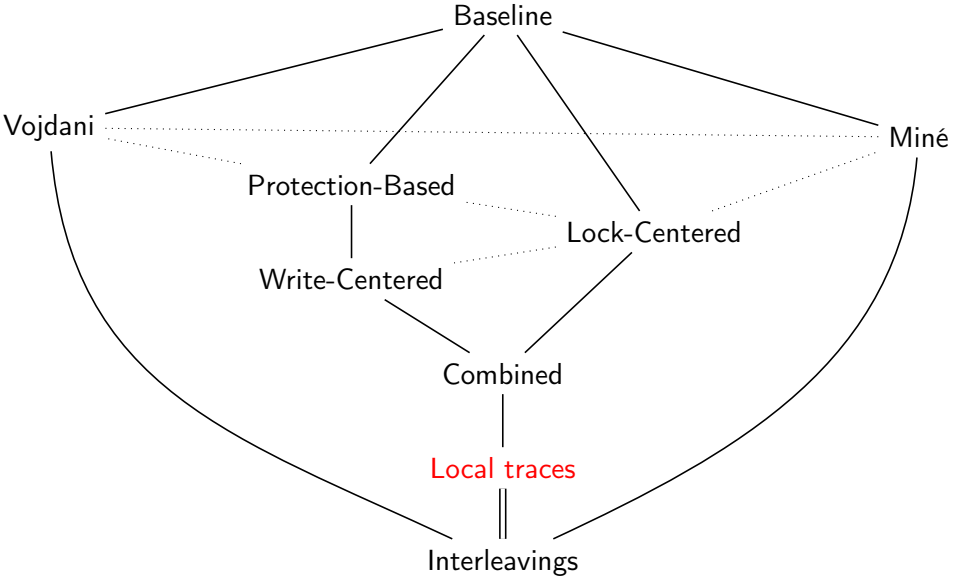
Improved analyses



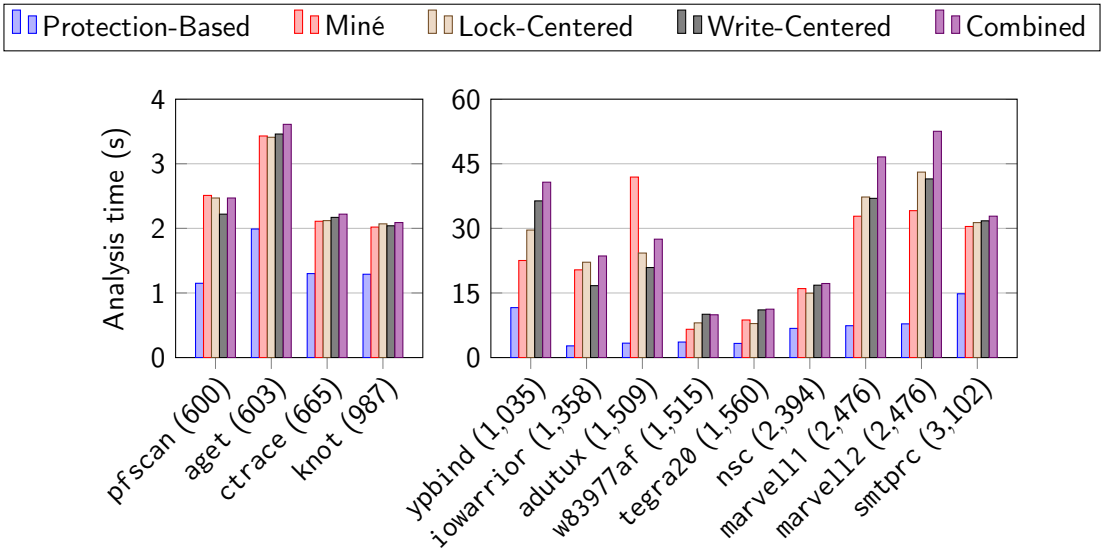
Improved analyses

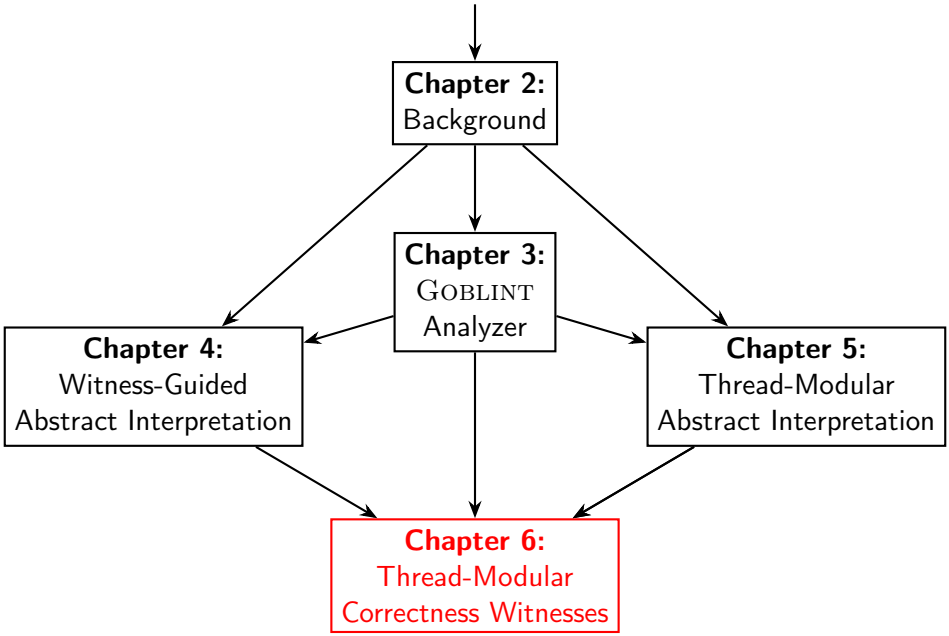


Improved analyses



Evaluation


Protection-Based
Miné
Lock-Centered
Write-Centered
Combined
Analysis time (s)
0
1
2
3
4
0
15
30
45
60
pfscan (600)
aget (603)
ctrace (665)
knot (987)
ypbind (1,035)
iowarrior (1,358)
adutux (1,509)
w83977af (1,515)
tegra20 (1,560)
nsc (2,394)
marvell1 (2,476)
marvell2 (2,476)
smtprc (3,102)


Chapter 2:
Background
Chapter 3:
GOBLINT
Analyzer
Chapter 4:
Witness-Guided
Abstract Interpretation
Chapter 5:
Thread-Modular
Abstract Interpretation
Chapter 6:
Thread-Modular
Correctness Witnesses

Thread-modular correctness witnesses

Witnesses in SV-COMP 2025:

	Witness type	
Program type	Violation	Correctness
Single-threaded	✓	✓
Multi-threaded	✓	✗

Difficulties

- Scalability (interleavings)
- Different analysis approaches
- Different concurrency semantics

Solution

- Thread-modular witnesses
- Avoid analysis specifics in witnesses

Two views

The local view:

- Internal thread perspective
  - Natural for thread-modular analyses
- Implicit scheduling
  - Forgets abstraction semantics
  - No scheduling invariants
- Witness format unmodified
- Can unassume

The global view:

- External perspective
  - Natural for interleaving semantics
- Ghost variables
  - Encode scheduling
  - Use in witness invariants
- Encode abstraction semantics using ghosts
  - Natural for protection-based analyses
- Witness format extended
  - SV-COMP 2026 demo

Local view example

```
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 }
```

Local view example

```
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 }
```

Invariants

- g protected by {m}
- $-10 \leq g \leq 10$

Protection-Based Reading:

From-scratch result

- $g : [-\infty, \infty]$ (due to widening)

Witness provides $-10 \leq g \leq 10$:

Witness-guided result

- $g : [-10, 10]$ (thanks to unassume)

Local view example

```
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 }
```

Invariants

- g protected by $\{m\}$
- $-10 \leq g \leq 10$

Protection-Based Reading:

From-scratch result

- $g : [-\infty, \infty]$ (due to widening)

Witness provides $-10 \leq g \leq 10$:

Witness-guided result

- $g : [-10, 10]$ (thanks to unassume)

Local view example

```
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 }
```

Invariants

- g protected by $\{m\}$
- $-10 \leq g \leq 10$

Protection-Based Reading:

From-scratch result

- $g : [-\infty, \infty]$ (due to widening)

Witness provides $-10 \leq g \leq 10$:

Witness-guided result

- $g : [-10, 10]$ (thanks to unassume)

Local view example witness

```c
void inc() {
  while (1) {
    lock(m);
      if (g < 10)
        g++;
    unlock(m);
  }
}

void dec() {
  while (1) {
    lock(m);
      if (g > -10)
        g--;
    unlock(m);
  }
}
```

```yaml
- entry_type: invariant_set
  content:
  - invariant:
      type: location_invariant
      location:
        line: 10
      value: -10 <= g && g <= 10
      format: c_expression
  - invariant:
      type: location_invariant
      location:
        line: 18
      value: -10 <= g && g <= 10
      format: c_expression
```

Content-effort dependence

Setup:

- Protection-Based Reading – already used to evaluate unassume operators!

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%)

Two views

The local view:

- Internal thread perspective
  - Natural for thread-modular analyses
- Implicit scheduling
  - Forgets abstraction semantics
  - No scheduling invariants
- Witness format unmodified
- Can unassume

The global view:

- External perspective
  - Natural for interleaving semantics
- Ghost variables
  - Encode scheduling
  - Use in witness invariants
- Encode abstraction semantics using ghosts
  - Natural for protection-based analyses
- Witness format extended
  - SV-COMP 2026 demo

Global view example

```
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 = 1;
12     g = 0;
13   unlock(m);
14 }
```

Invariants

- "Mutex m is locked" or $g = 0$
- If "mutex m is *not* locked", then $g = 0$

Problem

"Mutex m is (not) locked":

- Not C expression
- Not general – specific to some analyses

Solution

With ghost variable m_locked (mutex m is locked):

- $\texttt{m\_locked} \vee g = 0$
- $\neg\texttt{m\_locked} \Rightarrow g = 0$

Global view example

```
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 = 1;
12     g = 0;
13   unlock(m);
14 }
```

Invariants

- "Mutex m is locked" or $g = 0$
- If "mutex m is *not* locked", then $g = 0$

Problem

"Mutex m is (not) locked":

- Not C expression
- Not general – specific to some analyses

Solution

With ghost variable m_locked (mutex m is locked):

- $m\_locked \vee g = 0$
- $\neg m\_locked \Rightarrow g = 0$

Global view example

```
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 = 1;
12     g = 0;
13   unlock(m);
14 }
```

Invariants

- "Mutex m is locked" or $g = 0$
- If "mutex m is *not* locked", then $g = 0$

Problem

"Mutex m is (not) locked":

- Not C expression
- Not general – specific to some analyses

Solution

With ghost variable m_locked (mutex m is locked):

- $\texttt{m\_locked} \vee g = 0$
- $\neg\texttt{m\_locked} \Rightarrow g = 0$

Global view example instrumentation

```
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 = 1;
12     g = 0;
13   unlock(m);
14 }
```

```
1  int g = 0, m_locked = 0;
2  mutex m;

3  void main() {
4    create(t);
     atomic { assert(m_locked || g == 0); }
5    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 = 1;
12     g = 0;
13   atomic { unlock(m); m_locked = 0; }
14 }
```

Global view example witness

```
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 = 1;
12     g = 0;
13   unlock(m);
14 }
```

```
9  - entry_type: ghost_instrumentation
10   content:
11     ghost_variables:
12     - name: m_locked
13       type: int
14       scope: global
15       initial:
16         value: 0
17         format: c_expression
18     ghost_updates:
19     - location:
20         line: 5
21       updates:
22       - variable: m_locked
23         value: 1
24         format: c_expression
25     # ...
```

Global view example witness

```
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 = 1;
12     g = 0;
13   unlock(m);
14 }
```

```
1 - entry_type: invariant_set
2   content:
3   - invariant:
4       type: location_invariant
5       location:
6         line: 5
7       value: m_locked || g == 0
8       format: c_expression
```

Cross-framework validation

Setup:

- Verifier: GOBLINT – Protection-Based Reading
- Validator: UGEMCUTTER – model checker based on interleaving semantics
- Correct concurrent programs from SV-COMP 2024

Evaluations:

- Witness-only validation
- Local vs global view

Results:

- Found 29 pre-existing bugs

	Witnesses from			
UGEMCUTTER	Protection		Protection🔔	
Confirmed	122	(77%)	171	(94%)
Rejected	0	(0%)	0	(0%)
Out of resources	34	(21%)	5	(3%)
Error	3	(2%)	5	(3%)
Total	159		181	

Cross-framework validation

Setup:

- Verifier: GOBLINT – Protection-Based Reading
- Validator: UGEMCUTTER – model checker based on interleaving semantics
- Correct concurrent programs from SV-COMP 2024

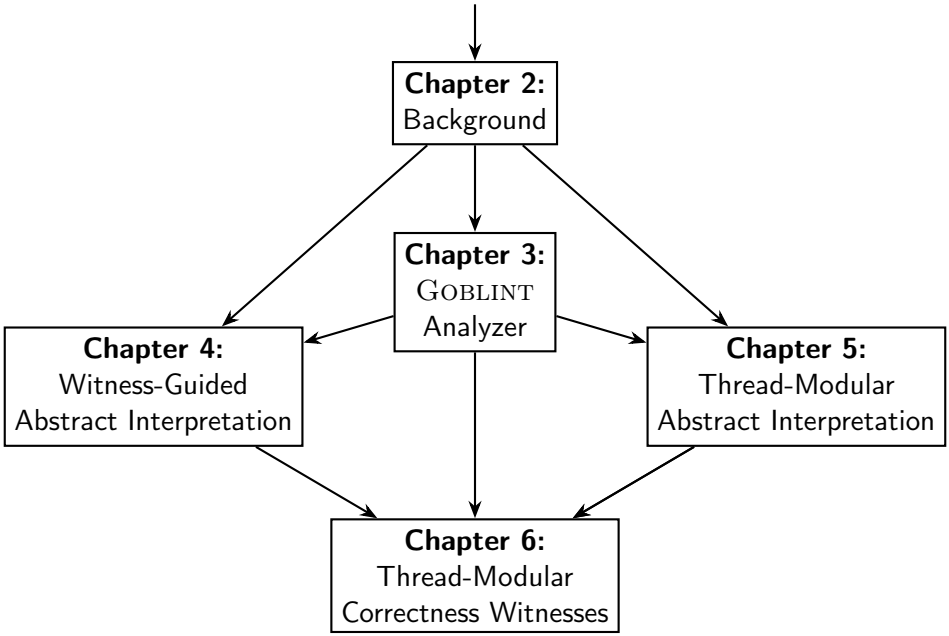
Evaluations:

- Witness-only validation
- Local vs global view

Results:

- Found 29 pre-existing bugs

	Witnesses from			
UGEMCUTTER	Protection		Protection$_{\text{👻}}$	
Confirmed	122	(77%)	171	(94%)
Rejected	0	(0%)	0	(0%)
Out of resources	34	(21%)	5	(3%)
Error	3	(2%)	5	(3%)
Total	159		181	


Chapter 2:
Background
Chapter 3:
GOBLINT
Analyzer
Chapter 4:
Witness-Guided
Abstract Interpretation
Chapter 5:
Thread-Modular
Abstract Interpretation
Chapter 6:
Thread-Modular
Correctness Witnesses

Conclusion

Overall goal

Performant, precise and trustworthy analysis of concurrent programs (using witnesses).

Subgoals:

- 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.

Conclusion

Overall goal

Performant, precise and trustworthy analysis of concurrent programs (using witnesses).

Subgoals:

- 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.

Conclusion

Overall goal

Performant, precise and trustworthy analysis of concurrent programs (using witnesses).

Subgoals:

- 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.

Conclusion

Overall goal

Performant, precise and trustworthy analysis of concurrent programs (using witnesses).

Subgoals:

- 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.

Conclusion

Overall goal

Performant, precise and trustworthy analysis of concurrent programs (using witnesses).

Subgoals:

- 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.

Conclusion

Overall goal

Performant, precise and trustworthy analysis of concurrent programs (using witnesses).

Subgoals:

- 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. ✔

Correctness Witnesses for Thread-Modular Program Analysis

Simmo Saan

Supervisor: Vesal Vojdani

April 23, 2026

Methodology

Design and implementation

Feedback loop:

1. Theory
   - Soundness
2. Practice:
   - Applicability

Evaluation

Pillars for correctness witnesses:

1. Same-framework consistency
2. Cross-framework validation
3. Content-effort dependence:
   - Performance
   - Precision

Engineering and maintenance

Tools in SV-COMP:

- Actively maintained
- State-of-the-art
- E.g., GOBLINT

Archival and reproducibility

Artifacts evaluated:

- Available
- Functional
- (Reusable)

GOBLINT analyses

Values & reachability Abstract domains for:

- Integers – intervals, exclusion sets, congruences, octagons, polyhedra
- Floats – intervals
- Pointers – points-to sets (heap by allocation-site), symbolic equalities
- Structs, unions, arrays – lifted (recursively)

Overflows Using value analysis

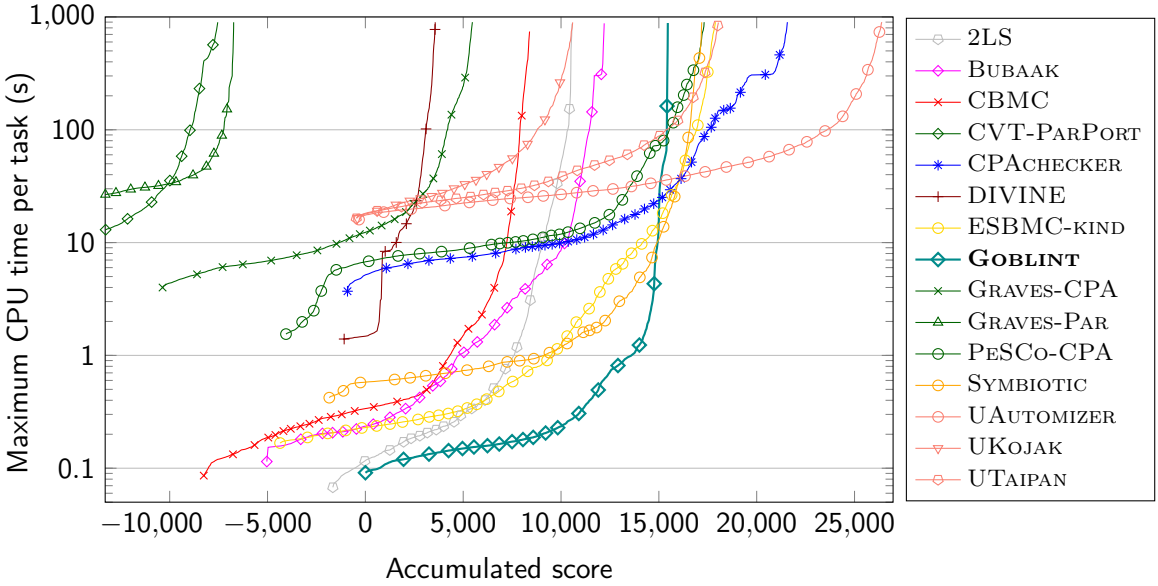
Data races Locksets (also symbolic), regions, may-happen-in-parallel (MHP)

Memory safety `NULL`, out-of-bounds, use after free, (also multi-threaded using MHP)

Memory leaks (Also multi-threaded)

Termination Loops, `gotos`, recursion, `longjmps`, (only single-threaded)

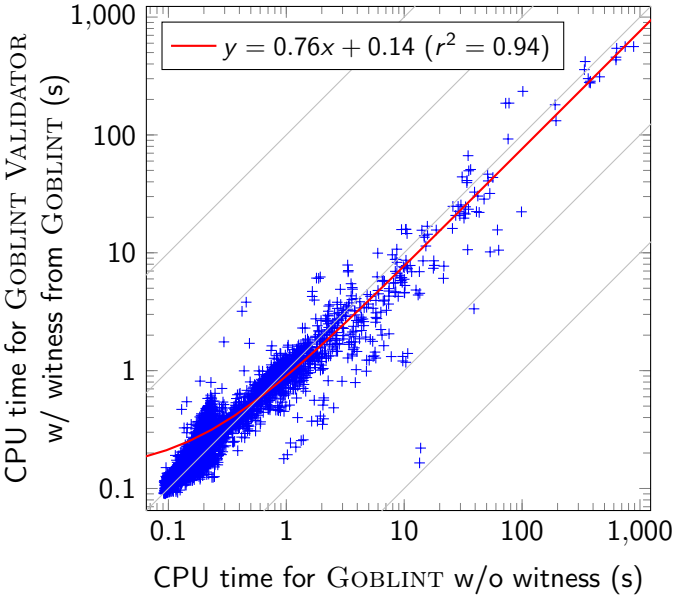
SV-COMP 2024 *Overall*



Same-framework consistency

Property	Correct tasks	GOBLINT verified	GOBLINT VALIDATOR Confirmed		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

Content-effort dependence (performance)



Same-framework and cross-framework evaluation

Setup:

- Incomparable analyses as frameworks: Protection-Based Reading & Miné's analysis
- Programs from their evaluation
- Generated witnesses as labels: invariants about protected globals after locking

Evaluations:

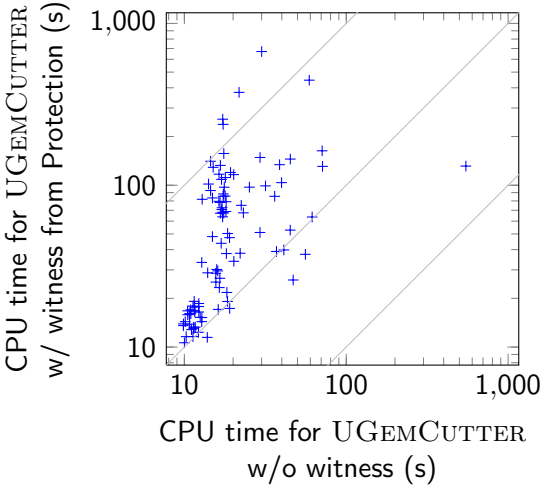
- Same-framework consistency
- Cross-framework *witness-guided verification*

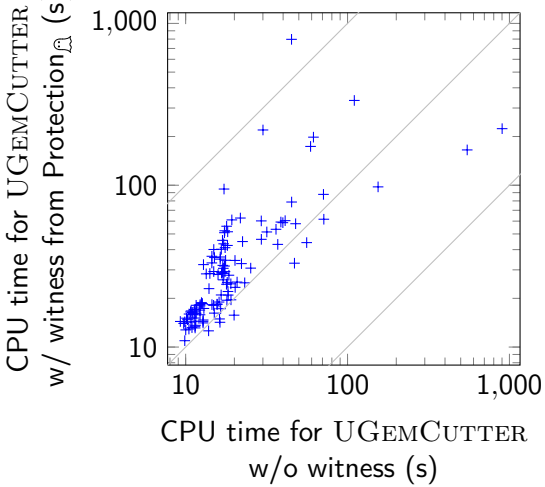
Analysis	Analysis invariants	Confirmed w/ guidance from			
		Protection		Miné	
Protection	1,232	1,231	(99.92%)	1,230	(99.84%)
Miné	1,266	1,257	(99.29%)	1,257	(99.29%)

Content-effort dependence

Evaluation:

- Witness validation – local vs global view





Future work

GOBLINT analyzer

- Analysis of thread pools
- Scalability to real-world programs

Thread-modular abstract interpretation

- Weak memory models
- Synchronization without mutexes

Witness-guided abstract interpretation

- Unsupported unassume specification **S11**
- Improved witness generation
- Witnesses for non-value information

Thread-modular correctness witnesses

- Ghost witness generation by other analyses
- Ghost witness validation by abstract interpretation
- Combined local and global views

Publications

Tool papers

Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.
GOBLINT: Thread-Modular Abstract Interpretation Using Side-Effecting Constraints.
In: TACAS 2021. Springer (2021).

Saan, S., Schwarz, M., Erhard, J., Pietsch, M., Seidl, H., Tilscher, S., Vojdani, V.
GOBLINT: Autotuning Thread-Modular Abstract Interpretation.
In: TACAS 2023. Springer (2023).

Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.
GOBLINT VALIDATOR: Correctness Witness Validation by Abstract Interpretation.
In: TACAS 2024. Springer (2024).

Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.
GOBLINT: Abstract Interpretation for Memory Safety and Termination.
In: TACAS 2024. Springer (2024).

Publications

Full papers

Schwarz, M., **Saan, S.**, Seidl, H., Apinis, K., Erhard, J., Vojdani, V.
Improving Thread-Modular Abstract Interpretation.
In: SAS 2021. Springer (2021).

Schwarz, M., **Saan, S.**, Seidl, H., Erhard, J., Vojdani, V.
Clustered Relational Thread-Modular Abstract Interpretation with Local Traces.
In: ESOP 2023. Springer (2023).

Saan, S., Schwarz, M., Erhard, J., Seidl, H., Tilscher, S., Vojdani, V.
Correctness Witness Validation by Abstract Interpretation.
In: VMCAI 2024. Springer (2024).

Erhard, J., Bentele, M., Heizmann, M., Klumpp, D., **Saan, S.**, Schüssele, F., Schwarz, M., Seidl, H., Tilscher, S., Vojdani, V.
Correctness Witnesses for Concurrent Programs: Bridging the Semantic Divide with Ghosts.
In: VMCAI 2025. Springer (2025).