

Correctness Witnesses for Concurrent Programs

Bridging the Semantic Divide with Ghosts

2nd April 2025, VeWit 2025

Frank Schüssele Julian Erhard Manuel Bentele Matthias Heizmann Dominik Klumpp
Simmo Saan Michael Schwarz Helmut Seidl Sarah Tilscher Vesal Vojdani

- Witness format for sequential programs (loop invariants, location invariants)¹

¹Paulína Ayaziová et al. “**Software Verification Witnesses 2.0**”. In: *SPIN*. vol. 14624. 2024.

- Witness format for sequential programs (loop invariants, location invariants)¹
- Goal: Format extension for concurrent programs

¹Paulína Ayaziová et al. “**Software Verification Witnesses 2.0**”. In: *SPIN*. vol. 14624. 2024.

- Witness format for sequential programs (loop invariants, location invariants)¹
- Goal: Format extension for concurrent programs
- Naive idea: location invariants

¹Paulína Ayaziová et al. “**Software Verification Witnesses 2.0**”. In: *SPIN*. vol. 14624. 2024.

Example

```
1 int used = 0;
2 mutex a;

3 main:
4     create(t1);
5     int y = 7;
6     lock(a);
7     assert(used == 0);
8     unlock(a);

9 t1:
10    lock(a);
11    used = nondet();
12    used = 0;
13    unlock(a);
```

Example

```
1 int used = 0;
2 mutex a;

3 main:
4   create(t1);
5   int y = 7;
6   lock(a);
7   assert(used == 0);
8   unlock(a);

9 t1:
10  lock(a);
11  used = nondet();
12  used = 0;
13  unlock(a);
```

Location invariant at line 6?

Example

```
1 int used = 0;
2 mutex a;

3 main:
4   create(t1);
5   int y = 7;
6   lock(a);
7   assert(used == 0);
8   unlock(a);

9 t1:
10  lock(a);
11  used = nondet();
12  used = 0;
13  unlock(a);
```

Location invariant at line 6? Problem: We want to reason about other threads!

Solution: Ghost Variables

Solution: Ghost Variables

- Thread modular invariants with ghost variables + ghost updates

Solution: Ghost Variables

- Thread modular invariants with ghost variables + ghost updates
- Established concept²

²Susan Owicki and David Gries. **“An Axiomatic Proof Technique for Parallel Programs I”**. In: *Acta Informatica* 6 (1976), pp. 319–340.

Solution: Ghost Variables

- Thread modular invariants with ghost variables + ghost updates
- Established concept²
- Suitable for different views of concurrent programs³

²Susan Owicki and David Gries. **“An Axiomatic Proof Technique for Parallel Programs I”**. In: *Acta Informatica* 6 (1976), pp. 319–340.

³Julian Erhard et al. **“Correctness Witnesses for Concurrent Programs: Bridging the Semantic Divide with Ghosts”**. In: *VMCAI (1)*. Vol. 15529. 2025.

Definition: Ghost Witness

A *ghost witness* for a program P is given by:

Definition: Ghost Witness

A *ghost witness* for a program P is given by:

- $\mathcal{G}_{\text{ghost}}$: Global ghost variables

Definition: Ghost Witness

A *ghost witness* for a program P is given by:

- $\mathcal{G}_{\text{ghost}}$: Global ghost variables
- $\mathcal{L}_{\text{ghost}}$: Local ghost variables

Definition: Ghost Witness

A *ghost witness* for a program P is given by:

- $\mathcal{G}_{\text{ghost}}$: Global ghost variables
- $\mathcal{L}_{\text{ghost}}$: Local ghost variables
- \mathcal{U} : Updates of ghost variables associated to statements

A *ghost witness* for a program P is given by:

- $\mathcal{G}_{\text{ghost}}$: Global ghost variables
- $\mathcal{L}_{\text{ghost}}$: Local ghost variables
- \mathcal{U} : Updates of ghost variables associated to statements
- \mathcal{I} : Invariants associated to locations.

Example ghost witness W

```
1 int used = 0;
2 mutex a;

3 main:
4   create( $t_1$ );
5   int y = 7;
6   lock(a);
7   assert(used == 0);
8   unlock(a);

9  $t_1$ :
10  lock(a);
11  used = nondet();
12  used = 0;
13  unlock(a);
```

Example ghost witness W

```
1 int used = 0;
2 mutex a;

3 main:
4   create( $t_1$ );
5   int y = 7;
6   lock(a);
7   assert(used == 0);
8   unlock(a);

9  $t_1$ :
10  lock(a);
11  used = nondet();
12  used = 0;
13  unlock(a);
```

Goal: "If mutex a is not locked, then $used==0$."

Example ghost witness W

```
1 int used = 0;
2 mutex a;

3 main:
4   create( $t_1$ );
5   int y = 7;
6   lock(a);
7   assert(used == 0);
8   unlock(a);

9  $t_1$ :
10  lock(a);
11  used = nondet();
12  used = 0;
13  unlock(a);
```

Goal: "If mutex a is not locked, then $used==0$."

- $\mathcal{G}_{\text{ghost}} = \{\text{ghost} = 0\}$, $\mathcal{L}_{\text{ghost}} = \emptyset$

Example ghost witness W

```
1 int used = 0;
2 mutex a;

3 main:
4   create(t1);
5   int y = 7;
6   lock(a);
7   assert(used == 0);
8   unlock(a);

9 t1:
10  lock(a);
11  used = nondet();
12  used = 0;
13  unlock(a);
```

Goal: "If mutex a is not locked, then $used==0$."

- $\mathcal{G}_{\text{ghost}} = \{\text{ghost} = 0\}$, $\mathcal{L}_{\text{ghost}} = \emptyset$
- $\mathcal{U} = \{\text{lock}(a) \mapsto \text{ghost} = 1, \text{unlock}(a) \mapsto \text{ghost} = 0\}$

Example ghost witness W

```
1 int used = 0;
2 mutex a;

3 main:
4   create(t1);
5   int y = 7;
6   lock(a);
7   assert(used == 0);
8   unlock(a);

9 t1:
10  lock(a);
11  used = nondet();
12  used = 0;
13  unlock(a);
```

Goal: "If mutex a is not locked, then $used==0$."

- $\mathcal{G}_{\text{ghost}} = \{\text{ghost} = 0\}$, $\mathcal{L}_{\text{ghost}} = \emptyset$
- $\mathcal{U} = \{\text{lock}(a) \mapsto \text{ghost} = 1, \text{unlock}(a) \mapsto \text{ghost} = 0\}$
- $\mathcal{I} = \{\ell_6 \mapsto \text{ghost} == 0 \implies \text{used} == 0\}$

Example of witness-instrumented program P^W

```
int used = 0;
mutex a;
int  $\mathcal{W}$  = 0;

main:
  create( $t_1$ );
  atomic{ assert( $\mathcal{W}$  == 0  $\implies$  used == 0); }
  int y = 7;
  atomic{ lock(a);  $\mathcal{W}$  = 1; }
  assert(used == 0);
  atomic{ unlock(a);  $\mathcal{W}$  = 0; }
```

```
 $t_1$ :
  atomic{ lock(a);  $\mathcal{W}$  = 1; }
  used = nondet();
  used = 0;
  atomic{ unlock(a);  $\mathcal{W}$  = 0; }
```

A witness W for a program P is *valid*.



The witness-instrumented program P_{ghost}^W is safe.

Witnesses format for C programs

We extend the witness 2.0 format^a
— which is used by the Software
Verification Competition.

^aPaulína Ayaziová et al. “**Software
Verification Witnesses 2.0**”. In: *SPIN*.
vol. 14624. 2024.

```
- entry_type: ghost_instrumentation
  metadata: ...
  content:
    ghost_variables:
      - name: 👻
        type: int
        scope: global
        initial:
          value: 0
          format: c_expression
    ghost_updates:
      - location: { line: 4, ... }
        updates:
          - variable: 👻
            value: 1
            format: c_expression
```

Invariants:

Invariants:

- Atomic evaluation

Invariants:

- Atomic evaluation
- Only need to hold at well-defined points (*sequence points*)

Invariants:

- Atomic evaluation
- Only need to hold at well-defined points (*sequence points*)

Ghost Variables:

Invariants:

- Atomic evaluation
- Only need to hold at well-defined points (*sequence points*)

Ghost Variables:

- Initialization: beginning of the program

Ghost Updates:

Ghost Updates:

- Cannot assume that the whole statement at given location is atomic (e.g., may contain function call)

Ghost Updates:

- Cannot assume that the whole statement at given location is atomic (e.g., may contain function call)
- Location for ghost updates restricted to specific statements (assignment or `pthread` library function call)

Ghost Updates:

- Cannot assume that the whole statement at given location is atomic (e.g., may contain function call)
- Location for ghost updates restricted to specific statements (assignment or `pthread` library function call)
- Ghost update atomically with action after evaluating the expressions (i.e., arguments for function call / right-hand side of assignment)

- *ConcurrencySafety-Main* category of SV-COMP

- *ConcurrencySafety-Main* category of SV-COMP
- Witness Generation: GOBLINT⁴ (thread modular abstract interpretation) with two analyses (protection, mutex-meet)

⁴Michael Schwarz et al. “**Improving Thread-Modular Abstract Interpretation**”. In: SAS. vol. 12913. 2021.

- *ConcurrencySafety-Main* category of SV-COMP
- Witness Generation: GOBLINT⁴ (thread modular abstract interpretation) with two analyses (protection, mutex-meet)
- Witness Validation: ULTIMATE GEMCUTTER⁵ (model checking using commutativity on interleaving)

⁴Michael Schwarz et al. “**Improving Thread-Modular Abstract Interpretation**”. In: SAS. vol. 12913. 2021.

⁵Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. “**Sound sequentialization for concurrent program verification**”. In: PLDI. ACM, 2022.

Evaluation: Witness confirmation

witnesses	protection-👻	mutexmeet-👻
witnesses for correct programs (348)		
total	181	217
confirmed	171	193
rejected	0	0
out of resources	5	19

Evaluation: Witness confirmation

witnesses	protection-👻	mutexmeet-👻
witnesses for correct programs (348)		
total	181	217
confirmed	171	193
rejected	0	0
out of resources	5	19
witnesses for incorrect programs (337)		
total	282	288
confirmed	192	164
rejected	0	0
out of resources	85	117

Evaluation: Witness validation runtime

witnesses	protection-👻 (181)	mutexmeet-👻 (217)
validation	1:31:33	1:29:27
verification	1:07:27	1:18:29

How to add tool support?

How to add tool support?

Witness Generation:

Witness Generation:

- Interleaving: encode program counters⁶

⁶Leslie Lamport. “**The ‘Hoare Logic’ of Concurrent Programs**”. In: *Acta Informatica* 14 (1980).

Witness Generation:

- Interleaving: encode program counters⁶
 - Optimization: only necessary interleaving info

⁶Leslie Lamport. “**The ‘Hoare Logic’ of Concurrent Programs**”. In: *Acta Informatica* 14 (1980).

Witness Generation:

- Interleaving: encode program counters⁶
 - Optimization: only necessary interleaving info
- Many more possibilities beyond encoding interleaving (e.g., mutex encoding)

⁶Leslie Lamport. “**The ‘Hoare Logic’ of Concurrent Programs**”. In: *Acta Informatica* 14 (1980).

How to add tool support?

Witness Generation:

- Interleaving: encode program counters⁶
 - Optimization: only necessary interleaving info
- Many more possibilities beyond encoding interleaving (e.g., mutex encoding)

Witness Validation:

⁶Leslie Lamport. “**The ‘Hoare Logic’ of Concurrent Programs**”. In: *Acta Informatica* 14 (1980).

How to add tool support?

Witness Generation:

- Interleaving: encode program counters⁶
 - Optimization: only necessary interleaving info
- Many more possibilities beyond encoding interleaving (e.g., mutex encoding)

Witness Validation:

- Transformation of program to instrument with ghosts
- Verification of transformed program

⁶Leslie Lamport. “**The ‘Hoare Logic’ of Concurrent Programs**”. In: *Acta Informatica* 14 (1980).

- Witness generation in `ULTIMATE`

- Witness generation in `ULTIMATE`
- Witness validation in `GOBLINT`

- Witness generation in `ULTIMATE`
- Witness validation in `GOBLINT`
- Further format extensions (*ghost update locations*)



<https://ultimate-pa.github.io/concurrency-witnesses/>