Concurrency Correctness Witnesses with Ghosts

Manuel Bentele[1,2] Dominik Klumpp[1] Frank Schüssele[1]

July 17, 2023

[1]University of Freiburg, Freiburg im Breisgau, Germany
[2]Hahn-Schickard-Gesellschaft, Villingen-Schwenningen, Germany

State of witnesses

	Sequential		Concurrent	
	Correctness	Violation	Correctness	Violation
ReachSafety	✓	✓	⚠	✓
NoOverflows	✓	✓	⚠	✓
Memsafety	✓	✓	⚠	✓
NoDataRace	-	-	???	???

State of witnesses

	Sequential		Concurrent	
	Correctness	Violation	Correctness	Violation
ReachSafety	✓	✓		✓
NoOverflows	✓	✓		✓
Memsafety	✓	✓		✓
NoDataRace	-	-	???	???

Motivation

Concurrency correctness witness proposal[1]:

- *Thread-modular* location invariants

[1] Simmo Saan and Julian Erhard. **"Beyond Automaton-Based Witnesses and Location Invariants".** 4th Workshop on Cooperative Software Verification (COOP 2023). Apr. 2023.

Motivation

Concurrency correctness witness proposal[1]:

- *Thread-modular* location invariants
  - Problem: thread-modular reasoning is incomplete
  - **Thesis:** Witness Format should be based on *complete* notion of proof

[1]Simmo Saan and Julian Erhard. **"Beyond Automaton-Based Witnesses and Location Invariants".** 4th Workshop on Cooperative Software Verification (COOP 2023). Apr. 2023.

Motivation

Concurrency correctness witness proposal[1]:

- *Thread-modular* location invariants
  - Problem: thread-modular reasoning is incomplete
  - **Thesis:** Witness Format should be based on *complete* notion of proof
- Additional extension to reason about mutexes

[1] Simmo Saan and Julian Erhard. **"Beyond Automaton-Based Witnesses and Location Invariants".** 4th Workshop on Cooperative Software Verification (COOP 2023). Apr. 2023.

Motivation

Concurrency correctness witness proposal[1]:

- *Thread-modular* location invariants
  - Problem: thread-modular reasoning is incomplete
  - **Thesis:** Witness Format should be based on *complete* notion of proof
- Additional extension to reason about mutexes
  - Specific to language / pthread features

[1] Simmo Saan and Julian Erhard. **"Beyond Automaton-Based Witnesses and Location Invariants".** 4th Workshop on Cooperative Software Verification (COOP 2023). Apr. 2023.

Motivation

Concurrency correctness witness proposal[1]:

- *Thread-modular* location invariants
  - Problem: thread-modular reasoning is incomplete
  - **Thesis:** Witness Format should be based on *complete* notion of proof
- Additional extension to reason about mutexes
  - Specific to language / pthread features
  - However: reasoning about mutual exclusion is crucial for concurrent program proofs

[1] Simmo Saan and Julian Erhard. **“Beyond Automaton-Based Witnesses and Location Invariants”.** 4th Workshop on Cooperative Software Verification (COOP 2023). Apr. 2023.

Incompleteness

```
int x;

thread inc() {
  int n = __VERIFIER_nondet_int();
  while (x < n) {
    x++;
    //@ invariant ???
  }
}

thread main() {
  pthread_create(inc);
  x = 42;
  assert x >= 42;
}
```

Incompleteness

```
int x;

thread inc() {
  int n = __VERIFIER_nondet_int();
  while (x < n) {
    x++;
    //@ invariant ???
  }
}

thread main() {
  pthread_create(inc);
  x = 42;
  assert x >= 42;
}
```

- Goal: Give *useful* invariant at specified location

Incompleteness

```
int x;

thread inc() {
  int n = __VERIFIER_nondet_int();
  while (x < n) {
    x++;
    //@ invariant ???
  }
}

thread main() {
  pthread_create(inc);
  x = 42;
  assert x >= 42;
}
```

- Goal: Give *useful* invariant at specified location
- Problem: depends on the interleaving

Incompleteness

```
int x;

thread inc() {
  int n = __VERIFIER_nondet_int();
  while (x < n) {
    x++;
    //@ invariant ???
  }
}

thread main() {
  pthread_create(inc);
  x = 42;
  assert x >= 42;
}
```

- Goal: Give *useful* invariant at specified location
- Problem: depends on the interleaving
- Current witness format not expressive enough

Thread-Modular Proofs with Ghost Variables

Thread-Modular Proofs with Ghost Variables

- Proofs require interleaving information

Thread-Modular Proofs with Ghost Variables

- Proofs require interleaving information
  - “Good” proof: as little interleaving information as possible
  - “Good” witness: as little control flow information as possible

Thread-Modular Proofs with Ghost Variables

- Proofs require interleaving information
  - "Good" proof: as little interleaving information as possible
  - "Good" witness: as little control flow information as possible
- Well-known approach: instrument program with *ghost variables*

Thread-Modular Proofs with Ghost Variables

- Proofs require interleaving information
  - "Good" proof: as little interleaving information as possible
  - "Good" witness: as little control flow information as possible
- Well-known approach: instrument program with *ghost variables*
- Thread-modular invariants + ghost variables: proof rule of Owicki and Gries[2]
  - Sound and (relatively) complete, even for unbounded threads[3]

  ⇒ Theoretical basis for our witness proposal

[2] Susan Owicki and David Gries. **"An Axiomatic Proof Technique for Parallel Programs I".** In: *Acta Informatica* 6 (1976), pp. 319–340. DOI: 10.1007/BF00268134.
[3] Leonor Prensa Nieto. **"Completeness of the Owicki-Gries System for Parameterized Parallel Programs".** In: *IPDPS*. IEEE Computer Society, 2001, p. 150.

Witnesses based on Owicki-Gries

Owicki-Gries Proofs:

- Ghost Variables
- Location Invariants

Concurrency Witnesses with Ghosts:

- Ghost Variables
- Location Invariants

Witnesses based on Owicki-Gries

Owicki-Gries Proofs:

- Ghost Variables
  - record information about execution
  - do not influence execution
  - added to program text
- Location Invariants

Concurrency Witnesses with Ghosts:

- Ghost Variables
- Location Invariants

Witnesses based on Owicki-Gries

Owicki-Gries Proofs:

- Ghost Variables
  - record information about execution
  - do not influence execution
  - added to program text
- Location Invariants

Concurrency Witnesses with Ghosts:

- Ghost Variables
  - record information about execution
  - do not influence execution
  - specified in witness
- Location Invariants

Witnesses based on Owicki-Gries

Owicki-Gries Proofs:

- Ghost Variables
  - record information about execution
  - do not influence execution
  - added to program text
- Location Invariants
  - use ghosts & program variables
  - inductive within a thread
  - interference-free wrt. other threads

Concurrency Witnesses with Ghosts:

- Ghost Variables
  - record information about execution
  - do not influence execution
  - specified in witness
- Location Invariants

Witnesses based on Owicki-Gries

Owicki-Gries Proofs:

- Ghost Variables
  - record information about execution
  - do not influence execution
  - added to program text
- Location Invariants
  - use ghosts & program variables
  - inductive within a thread
  - interference-free wrt. other threads

Concurrency Witnesses with Ghosts:

- Ghost Variables
  - record information about execution
  - do not influence execution
  - specified in witness
- Location Invariants
  - use ghosts & program variables
  - must hold whenever program is in location

Program with ghosts

```
int x;
int g = 0;

thread inc() {
  int n = __VERIFIER_nondet_int();
  while (x < n) {
    x++;
    //@ invariant g != 1 || x >= 42
  }
}

thread main() {
  pthread_create(inc);
  atomic { g = 1; x = 42; }
  assert x >= 42;
}
```

Witness with ghosts

```
int x;
int g = 0;

thread inc() {
  int n = __VERIFIER_nondet_int();
  while (x < n) {
    x++;
    //@ invariant g != 1 || x >= 42
  }
}

thread main() {
  pthread_create(inc);
  atomic { g = 1; x = 42; }
  assert x >= 42;
}
```

```yaml
- entry_type: ghost_variable
  name: g
  scope: global
  type: int
  initial: 0

- entry_type: location_invariant
  location: ...
  location_invariant:
    string: g != 1 || x >= 42

- entry_type: ghost_update
  variable: g
  expression: 1
  location: ...
```

Witness with ghosts

```
int x;
int g = 0;

thread inc() {
  int n = __VERIFIER_nondet_int();
  while (x < n) {
    x++;
    //@ invariant g != 1 || x >= 42
  }
}

thread main() {
  pthread_create(inc);
  atomic { g = 1; x = 42; }
  assert x >= 42;
}
```

```
- entry_type: ghost_variable
  name: g
  scope: global
  type: int
  initial: 0

- entry_type: location_invariant
  location: ...
  location_invariant:
    string: g != 1 || x >= 42

- entry_type: ghost_update
  variable: g
  expression: 1
  location: ...
```

Witness semantics

Witness semantics

- Initialization of global ghosts after initialization of program variables

Witness semantics

- Initialization of global ghosts after initialization of program variables
- Update atomically right before leaving the specified location

Witness semantics

- Initialization of global ghosts after initialization of program variables
- Update atomically right before leaving the specified location
- Expression in updates must not have side-effects or undefined behaviour

Witness semantics

- Initialization of global ghosts after initialization of program variables
- Update atomically right before leaving the specified location
- Expression in updates must not have side-effects or undefined behaviour
  - Special handling for data races: Assume every ghost update *happens-before* (or *happens-after*) expression evaluations in the program
    $\Rightarrow$ Ghost updates do not introduce data races

Fancy ghost variables

```
int x;
int g = 0;

thread inc() {
  int n = __VERIFIER_nondet_int();
  while (x < n) {
    x++;
    //@ invariant x >= g
  }
}

thread main() {
  int val = __VERIFIER_nondet_int();
  pthread_create(inc);
  atomic { g = val; x = val; }
  assert x >= val;
}
```

Fancy ghost variables

```
int x;
int g = 0;

thread inc() {
  int n = __VERIFIER_nondet_int();
  while (x < n) {
    x++;
    //@ invariant x >= g
  }
}

thread main() {
  int val = __VERIFIER_nondet_int();
  pthread_create(inc);
  atomic { g = val; x = val; }
  assert x >= val;
}
```

- Ghosts that are set to program variables

Fancy ghost variables

```
int x;
int g = 0;

thread inc() {
  int n = __VERIFIER_nondet_int();
  while (x < n) {
    x++;
    //@ invariant x >= g
  }
}

thread main() {
  int val = __VERIFIER_nondet_int();
  pthread_create(inc);
  atomic { g = val; x = val; }
  assert x >= val;
}
```

- Ghosts that are set to program variables
- Allows reasoning over more than just interleavings

Mutex reasoning with ghosts

```
int used = 0, g = 0;
mutex m;

thread producer() {
  while (1) {
    atomic { g = 1; lock(m); }
    used++; used--;
    atomic { g = 0; unlock(m); }
  }
}

thread main() {
  pthread_create(producer);
  //@ invariant g != 0 || used == 0
  atomic { g = 1; lock(m); }
  assert used == 0;
  atomic { g = 0; unlock(m); }
}
```

universität freiburg

Mutex reasoning with ghosts

```
int used = 0, g = 0;
mutex m;

thread producer() {
  while (1) {
    atomic { g = 1; lock(m); }
    used++; used--;
    atomic { g = 0; unlock(m); }
  }
}

thread main() {
  pthread_create(producer);
  //@ invariant g != 0 || used == 0
  atomic { g = 1; lock(m); }
  assert used == 0;
  atomic { g = 0; unlock(m); }
}
```

- Ghost variables to reason about mutexes

Mutex reasoning with ghosts

```
int used = 0, g = 0;
mutex m;

thread producer() {
  while (1) {
    atomic { g = 1; lock(m); }
    used++; used--;
    atomic { g = 0; unlock(m); }
  }
}

thread main() {
  pthread_create(producer);
  //@ invariant g != 0 || used == 0
  atomic { g = 1; lock(m); }
  assert used == 0;
  atomic { g = 0; unlock(m); }
}
```

- Ghost variables to reason about mutexes
- Invariants can relate program variables and mutexes (via ghosts)

Mutex reasoning with ghosts

```
int used = 0, g = 0;
mutex m;

thread producer() {
  while (1) {
    atomic { g = 1; lock(m); }
    used++; used--;
    atomic { g = 0; unlock(m); }
  }
}

thread main() {
  pthread_create(producer);
  //@ invariant g != 0 || used == 0
  atomic { g = 1; lock(m); }
  assert used == 0;
  atomic { g = 0; unlock(m); }
}
```

- Ghost variables to reason about mutexes
- Invariants can relate program variables and mutexes (via ghosts)
- However: Validator has to find relation between `m` and `g`

Tool Implementation

Witness Generation:

Witness Validation:

Tool Implementation

Witness Generation:

- Standard Owicki-Gries approach: Encode program counters[4]

Witness Validation:

[4] Leslie Lamport. **"The 'Hoare Logic' of Concurrent Programs".** In: *Acta Informatica* 14 (1980), pp. 21–37. DOI: 10.1007/BF00289062.

Tool Implementation

Witness Generation:

- Standard Owicki-Gries approach: Encode program counters[4]
  - Optimization: only necessary interleaving info

Witness Validation:

[4] Leslie Lamport. **"The 'Hoare Logic' of Concurrent Programs".** In: *Acta Informatica* 14 (1980), pp. 21–37. DOI: 10.1007/BF00289062.

Tool Implementation

Witness Generation:

- Standard Owicki-Gries approach: Encode program counters[4]
  - Optimization: only necessary interleaving info
- Many more possibilities beyond encoding interleaving

Witness Validation:

---

[4] Leslie Lamport. **"The 'Hoare Logic' of Concurrent Programs".** In: *Acta Informatica* 14 (1980), pp. 21–37. DOI: 10.1007/BF00289062.

Tool Implementation

Witness Generation:

- Standard Owicki-Gries approach: Encode program counters[4]
  - Optimization: only necessary interleaving info
- Many more possibilities beyond encoding interleaving

Witness Validation:

- Transformation of program to instrument with ghosts

[4] Leslie Lamport. **"The 'Hoare Logic' of Concurrent Programs".** In: *Acta Informatica* 14 (1980), pp. 21–37. DOI: 10.1007/BF00289062.

Tool Implementation

Witness Generation:

- Standard Owicki-Gries approach: Encode program counters[4]
  - Optimization: only necessary interleaving info
- Many more possibilities beyond encoding interleaving

Witness Validation:

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

[4] Leslie Lamport. **"The 'Hoare Logic' of Concurrent Programs".** In: *Acta Informatica* 14 (1980), pp. 21–37. DOI: 10.1007/BF00289062.

Strengths

Strengths

- Based on complete proof notion

Strengths

- Based on complete proof notion
- General approach, not bound to tool-specific representation

Strengths

- Based on complete proof notion
- General approach, not bound to tool-specific representation
- Covers many different language features / synchronization mechanisms

Strengths

- Based on complete proof notion
- General approach, not bound to tool-specific representation
- Covers many different language features / synchronization mechanisms
- Remains as (thread-)modular as possible, do not encode all interleavings

Strengths

- Based on complete proof notion
- General approach, not bound to tool-specific representation
- Covers many different language features / synchronization mechanisms
- Remains as (thread-)modular as possible, do not encode all interleavings
- Ghost variables: not restricted to concurrency

Limitations

Limitations

- Proof format of approaches that use reductions (with meta-reasoning) still open research question

Limitations

- Proof format of approaches that use reductions (with meta-reasoning) still open research question
    - General problem of witnesses how to encode such meta-reasoning
    - Ghost variables could help with that encoding

Limitations

- Proof format of approaches that use reductions (with meta-reasoning) still open research question
  - General problem of witnesses how to encode such meta-reasoning
  - Ghost variables could help with that encoding
- Allowed update locations? (e.g. where in loop, switch/case?)
  - Problem with the general format, not only with this extension

Limitations

- Proof format of approaches that use reductions (with meta-reasoning) still open research question
  - General problem of witnesses how to encode such meta-reasoning
  - Ghost variables could help with that encoding
- Allowed update locations? (e.g. where in loop, switch/case?)
  - Problem with the general format, not only with this extension
- Further extension for multiple instances of the same thread template needed?
  - thread-local ghost variables
  - quantification (ACSL)
  - unbounded ghost arrays

Conclusion

Conclusion

- Problem: Incomplete witnesses for concurrency

Conclusion

- Problem: Incomplete witnesses for concurrency
- Proposal of new extension with ghost variables

Conclusion

- Problem: Incomplete witnesses for concurrency
- Proposal of new extension with ghost variables
- General approach, possible to be used by different tools (generation/validation)

Additional materials

https://github.com/ultimate-pa/VEWIT2023-ConcurrencyGhosts