Beyond Automaton-Based Witnesses and Location Invariants

Simmo Saan[1] Julian Erhard[2]

[1]University of Tartu, Estonia

[2]Technical University of Munich, Germany

COOP '23
April 23, 2023
Paris

Outline

Motivation

1. Make witnesses easier to generate and consume for non–model-checking approaches
   - Reduce use of trivial witnesses

2. Avoid semantics issues due to underspecified control-flow automata
   - Identified by Strejček (2022)

```
if (f(x) < g(x))
  /* ... */
else
  /* ... */
```

3. Propose correctness witness format for multi-threaded programs
   - So far missing in SV-COMP

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

Witness workflow

As visualized by Beyer and Wehrheim (2020):

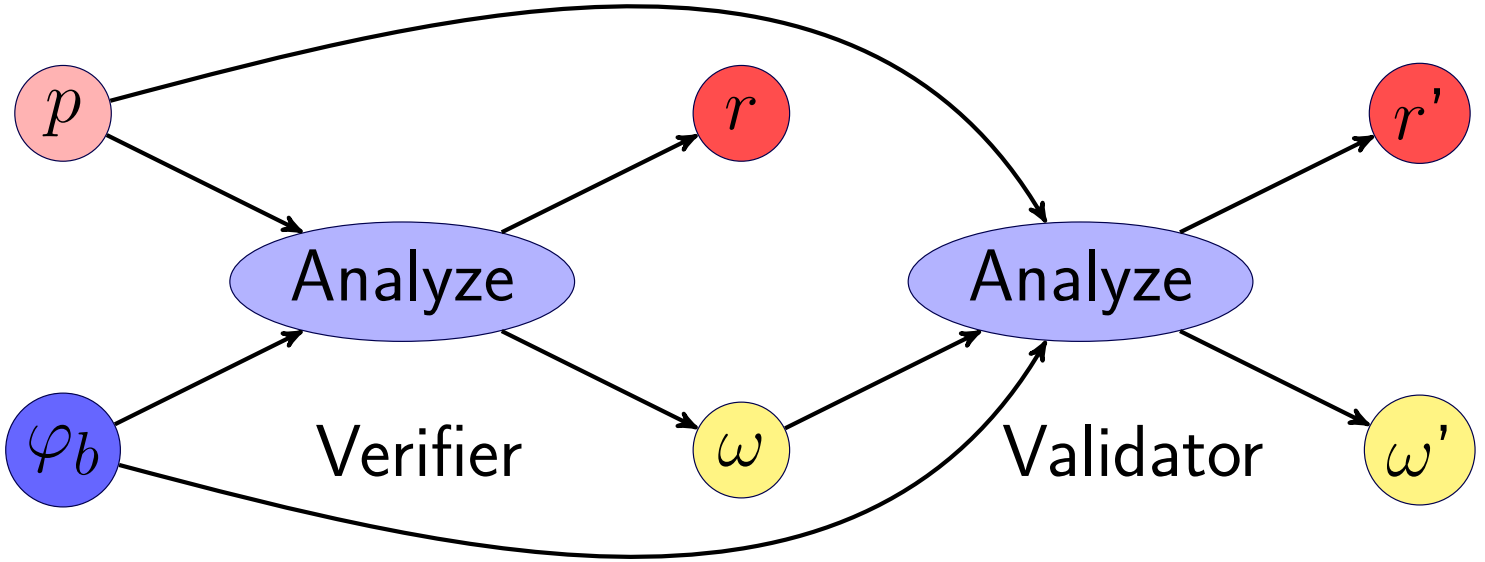


Fig. 7. Witness-based results validation

Witnesses

Intuition: explain why program P satisfies some safety property Φ.

Correctness witness

A *correctness witness* for a program P is a tuple (W, Φ), where W is a property of P.

Witnesses

Intuition: explain why program P satisfies some safety property Φ.

Correctness witness

A *correctness witness* for a program P is a tuple (W, Φ), where W is a property of P.

Valid witness (Beyer et al., 2016)

A correctness witness (W, Φ) for a program P is *valid* if

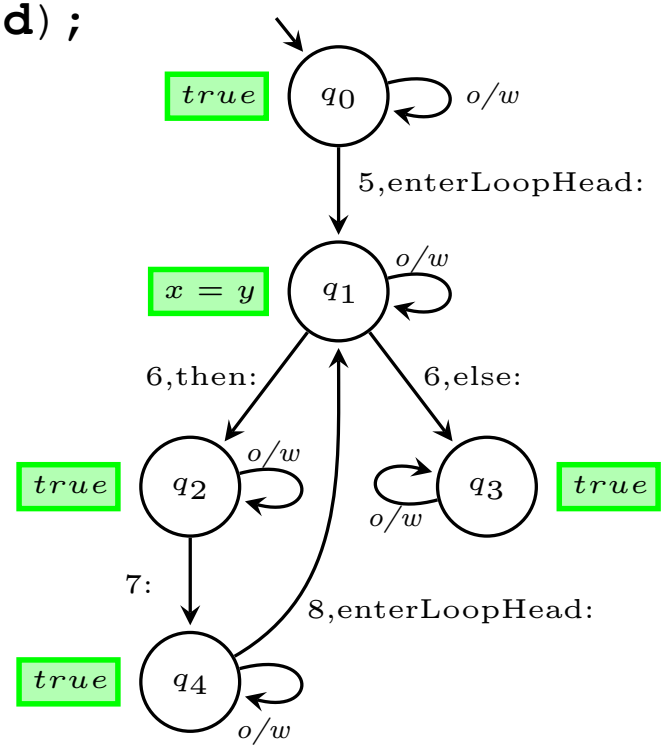
1. P satisfies W,
2. W implies (P satisfies Φ).

Automaton-based witnesses

Automaton-based format for correctness witnesses by Beyer et al. (2016):

```
1  extern unsigned int nondet(void);
2
3  int main() {
4    unsigned int x = nondet();
5    unsigned int y = x;
6    while (x < 1024) {
7      x = x + 1;
8      y = y + 1;
9    }
10   // Valid safety property
11   if (x != y) {
12     ERROR: return 1;
13   }
14   return 0;
15 }
```

(a) Safe program



(b) Witness automaton

Entry-based YAML format

- Introduced by SoSy-Lab (2021)
  - For an invariant store (Weise, 2021)
- Correctness witnesses
- List of entries of predefined types
- Extensible by design

Entry-based YAML format

- Introduced by SoSy-Lab (2021)
  - For an invariant store (Weise, 2021)
- Correctness witnesses
- List of entries of predefined types
- Extensible by design

- So far practically unused
- Only one entry type: location invariants

Location invariants

- Invariant which holds every time the location is reached
- Automaton-based witnesses are not always necessary in practice (Beyer et al., 2022)

Location invariants

- Invariant which holds every time the location is reached
- Automaton-based witnesses are not always necessary in practice (Beyer et al., 2022)

```
1  void main() {
2    int *p =
3      malloc(sizeof(int));
4    *p = 10;
5    int i = 0;
6    while (i < 100)
7      i++;
8    assert(i == 100);
9    assert(*p == 10);
10 }
```

Location invariants

- Invariant which holds every time the location is reached
- Automaton-based witnesses are not always necessary in practice (Beyer et al., 2022)

```
- entry_type: location_invariant
  metadata: # omitted
  location:
    function: main
    line: 7
    column: 4
  location_invariant:
    type: assertion
    format: C
    string: 0 <= i && i <= 99 && *p == 10
```

```
1  void main() {
2    int *p =
3      malloc(sizeof(int));
4    *p = 10;
5    int i = 0;
6    while (i < 100)
7      i++;
8    assert(i == 100);
9    assert(*p == 10);
10 }
```

Proposed extensions

New entry types:

- Precondition invariants
- Flow-insensitive invariants
- Variables protected by mutexes
- Mutexes held at location

Proposed extensions

New entry types:

- Precondition invariants
- Flow-insensitive invariants
- Variables protected by mutexes
- Mutexes held at location

Consequence

YAML correctness witnesses for *multi-threaded* programs!

Precondition invariants

Goal

Express context-sensitive information about functions.

Precondition invariants

Goal

Express context-sensitive information about functions.

Idea

Relate invariants that hold in f for some calls to the corresponding entry-state of f.

Precondition invariants

Goal

Express context-sensitive information about functions.

Idea

Relate invariants that hold in f for some calls to the corresponding entry-state of f.

Precondition invariant example

```
1  int foo(int *p, int *q) {
2    *p = 1;
3    *q = 2;
4    int result = *p + 3;
5    return result;
6  }
7
8  void main() {
9    int x, y, z;
10   assert(foo(&x, &y) == 4);
11   assert(foo(&z, &z) == 5);
12 }
```

Precondition invariant example

```yaml
- entry_type: precondition_location_invariant
  location:
    function: foo
    line: 5
    column: 2
  precondition:
    type: assertion
    format: C
    string: p != q
  location_invariant:
    type: assertion
    format: C
    string: result == 4
```

```c
int foo(int *p, int *q) {
  *p = 1;
  *q = 2;
  int result = *p + 3;
  return result;
}

void main() {
  int x, y, z;
  assert(foo(&x, &y) == 4);
  assert(foo(&z, &z) == 5);
}
```

Precondition invariant example

```yaml
- entry_type: precondition_location_invariant
  location:
    function: foo
    line: 5
    column: 2
  precondition:
    type: assertion
    format: C
    string: p == q
  location_invariant:
    type: assertion
    format: C
    string: result == 5
```

```
1  int foo(int *p, int *q) {
2    *p = 1;
3    *q = 2;
4    int result = *p + 3;
5    return result;
6  }
7
8  void main() {
9    int x, y, z;
10   assert(foo(&x, &y) == 4);
11   assert(foo(&z, &z) == 5);
12 }
```

Precondition invariant definition

Precondition invariant

A *precondition invariant* (L, P, I), with:

- L: program location
- P: precondition at entry of func(L), the function containing L
- I: invariant at location L

Semantics

If P held when func(L) was entered,
I holds at program location L.

Precondition invariants from context-sensitive analysis

Analysis yields pairs $(C_i, S_i)_{1 \leq i \leq n}$ at location L:

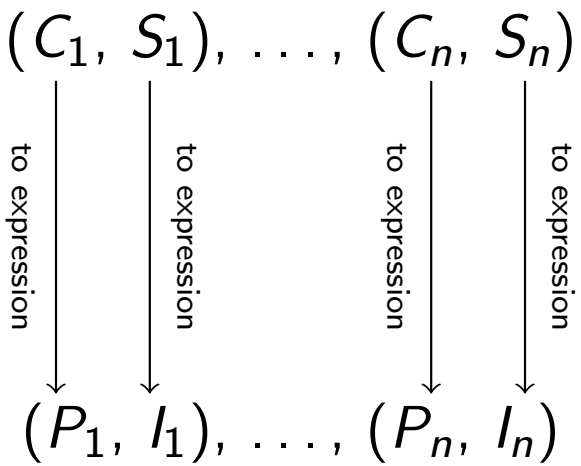
- when analyzing func(L) starting in context C_i,
- S_i is the resulting abstract state at L
- $(C_i)_{1 \leq i \leq n}$ abstract all actual calling contexts of f

Precondition invariants from context-sensitive analysis

Analysis yields pairs $(C_i, S_i)_{1 \leq i \leq n}$ at location L:

- when analyzing func(L) starting in context C_i,
- S_i is the resulting abstract state at L
- $(C_i)_{1 \leq i \leq n}$ abstract all actual calling contexts of f

Translation to expressions:



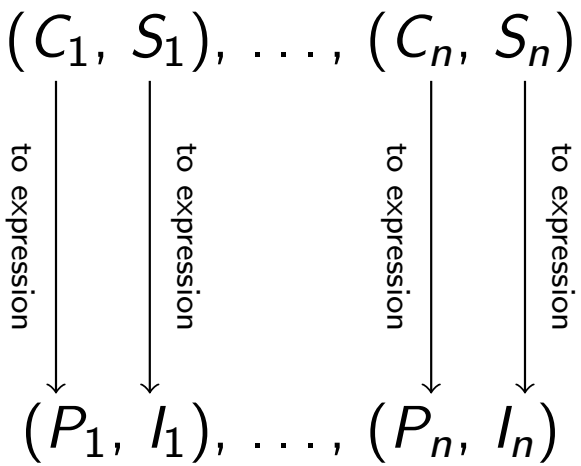
$\Longrightarrow$ P_i precondition, I_i invariant.

Precondition invariants from context-sensitive analysis

Analysis yields pairs $(C_i, S_i)_{1 \leq i \leq n}$ at location L:

- when analyzing func(L) starting in context C_i,
- S_i is the resulting abstract state at L
- $(C_i)_{1 \leq i \leq n}$ abstract all actual calling contexts of f

Translation to expressions:



$\Longrightarrow$ P_i precondition, I_i invariant.

Direct translation yields precondition invariants:

$$(L, P_1, I_1), \ldots, (L, P_n, I_n).$$

Precondition expressivity problem

```
1  int choose(int a[], int n) {
2    int i = rand() % n;
3    int result = a[i];
4    return result;
5  }
6
7  void main() {
8    int A[5] = {0, 0, 0, 0, 0};
9    int B[5] = {1, 1, 1, 1, 1};
10   assert(choose(A, 5) == 0);
11   assert(choose(B, 5) == 1);
12 }
```

Precondition expressivity problem

At line 4, column 2:

1. Context:
$$\left\{\begin{array}{l} \mathtt{a} \mapsto \mathrm{array}([0,0]) \\ \mathtt{n} \mapsto [5,5] \end{array}\right\}$$
State:
$$\left\{\begin{array}{rl} \mathtt{i} & \mapsto [0,4] \\ \mathtt{result} & \mapsto [0,0] \end{array}\right\}$$
2. Context:
$$\left\{\begin{array}{l} \mathtt{a} \mapsto \mathrm{array}([1,1]) \\ \mathtt{n} \mapsto [5,5] \end{array}\right\}$$
State:
$$\left\{\begin{array}{rl} \mathtt{i} & \mapsto [0,4] \\ \mathtt{result} & \mapsto [1,1] \end{array}\right\}$$

```
1  int choose(int a[], int n) {
2    int i = rand() % n;
3    int result = a[i];
4    return result;
5  }
6
7  void main() {
8    int A[5] = {0, 0, 0, 0, 0};
9    int B[5] = {1, 1, 1, 1, 1};
10   assert(choose(A, 5) == 0);
11   assert(choose(B, 5) == 1);
12 }
```

Precondition expressivity problem

```
- entry_type: precondition_location_invariant
  location:
    function: choose
    line: 4
    column: 2
  precondition:
    type: assertion
    format: C
    string: n == 5
  location_invariant:
    type: assertion
    format: C
    string: result == 0
# similar for result == 1
```

```
1  int choose(int a[], int n) {
2    int i = rand() % n;
3    int result = a[i];
4    return result;
5  }
6
7  void main() {
8    int A[5] = {0, 0, 0, 0, 0};
9    int B[5] = {1, 1, 1, 1, 1};
10   assert(choose(A, 5) == 0);
11   assert(choose(B, 5) == 1);
12 }
```

Precondition expressivity problem

Direct translation problem

Direct translation to precondition invariants is problematic if

$$[\![P_i]\!]^\sharp(C_j) \ni \text{true}$$

for some $i \neq j$, i.e. the precondition P_i may also evaluate to true in some *different* context C_j.

Precondition expressivity problem

Direct translation problem

Direct translation to precondition invariants is problematic if

$$[\![P_i]\!]^\sharp(C_j) \ni \text{true}$$

for some $i \neq j$, i.e. the precondition P_i may also evaluate to true in some *different* context C_j.

Sound translation yields precondition invariants:

$$(L, P_1, I'_1), \ldots, (L, P_n, I'_n),$$

where

$$I'_k = \bigvee_{\substack{1 \leq j \leq n \\ [\![P_k]\!]^\sharp(C_j) \ni \text{true}}} I_j.$$

Precondition expressivity problem fixed

```
- entry_type: precondition_location_invariant
  location:
    function: choose
    line: 4
    column: 2

  precondition:
    type: assertion
    format: C
    string: n == 5

  location_invariant:
    type: assertion
    format: C
    string: result == 0 || result == 1
```

```
1  int choose(int a[], int n) {
2    int i = rand() % n;
3    int result = a[i];
4    return result;
5  }
6
7  void main() {
8    int A[5] = {0, 0, 0, 0, 0};
9    int B[5] = {1, 1, 1, 1, 1};
10   assert(choose(A, 5) == 0);
11   assert(choose(B, 5) == 1);
12 }
```

Flow-insensitive invariants

```
int used = 0;
int free = 100;
void push() {
  if (free >= 1) {
    free--; used++;
  }
}
void pop() {
  if (used >= 1) {
    used--; free++;
  }
}
```

Flow-insensitive invariants

```
- entry_type: flow_insensitive_invariant
  flow_insensitive_invariant:
    type: assertion
    format: C
    string: used + free <= 100
```

```
int used = 0;
int free = 100;
void push() {
  if (free >= 1) {
    free--; used++;
  }
}
void pop() {
  if (used >= 1) {
    used--; free++;
  }
}
```

Flow-insensitive invariants

```yaml
- entry_type: flow_insensitive_invariant
  flow_insensitive_invariant:
    type: assertion
    format: C
    string: used + free <= 100
```

```c
int used = 0;
int free = 100;
void push() {
  if (free >= 1) {
    free--; used++;
  }
}
void pop() {
  if (used >= 1) {
    used--; free++;
  }
}
```

- Invariant on global variables which holds throughout entire execution
- Not tied to any program location
- Inconvenient with automaton-based witnesses or location invariants:
  - Duplicated at many locations/nodes

Flow-insensitive invariants in multi-threaded programs

Also useful for multi-threaded programs, abstracting over interleavings.

```
- entry_type: flow_insensitive_invariant
  flow_insensitive_invariant:
    type: assertion
    format: C
    string: used + free <= 100
```

```
mutex m;
void producer() {
  while (1) {
    lock(m);
    push();
    unlock(m);
  }
}
void consumer() {
  while (1) {
    lock(m);
    pop();
    unlock(m);
  }
}
```

Location invariants in multi-threaded programs

Problem

SV-COMP: No format for multi-threaded correctness-witnesses yet.

A product-automaton expressing all actions of multiple threads faces explosion of possible interleavings.

Location invariants in multi-threaded programs

Problem

SV-COMP: No format for multi-threaded correctness-witnesses yet.

A product-automaton expressing all actions of multiple threads faces explosion of possible interleavings.

Solution

Location (and precondition) invariants can fill this gap:

- Independent of thread count
- Thread-modular
- Scales to unbounded thread instances
- Concurrency information is implicit in reachability of location

Location invariants in multi-threaded programs

```yaml
- entry_type: location_invariant
  location:
    function: producer
    line: 5
    column: 4
  location_invariant:
    type: assertion
    format: C
    string: used + free == 100
# similar for consumer
```

```
1  mutex m;
2  void producer() {
3    while (1) {
4      lock(m);
5      push();
6      unlock(m);
7    }
8  }
9  void consumer() {
10   while (1) {
11     lock(m);
12     pop();
13     unlock(m);
14   }
15 }
```

Entry types for multi-threaded programs

So far

Only entry types for program variable values.

Reasoning about multi-threaded programs suggests other possibilities:

Entry types for multi-threaded programs

So far

Only entry types for program variable values.

Reasoning about multi-threaded programs suggests other possibilities:

1. Mutexes protecting variables
   - Concurrency safety reasoning
   - Witness for data-race freedom
   - Prerequisite information for other analyses

Entry types for multi-threaded programs

So far

Only entry types for program variable values.

Reasoning about multi-threaded programs suggests other possibilities:

1. Mutexes protecting variables
   - Concurrency safety reasoning
   - Witness for data-race freedom
   - Prerequisite information for other analyses
2. Mutexes held at locations

Protected-by in multi-threaded programs

```
- entry_type: protected_by
  variable: used
  mutex: m

- entry_type: protected_by
  variable: free
  mutex: m
```

```
mutex m;
void producer() {
  while (1) {
    lock(m);
    push();
    unlock(m);
  }
}
void consumer() {
  while (1) {
    lock(m);
    pop();
    unlock(m);
  }
}
```

Location mutexes in multi-threaded programs

```yaml
- entry_type: location_mutex
  location:
    function: producer
    line: 5
    column: 4
  mutex: m

- entry_type: location_mutex
  location:
    function: producer
    line: 12
    column: 4
  mutex: m
```

```
1  mutex m;
2  void producer() {
3    while (1) {
4      lock(m);
5      push();
6      unlock(m);
7    }
8  }
9  void consumer() {
10   while (1) {
11     lock(m);
12     pop();
13     unlock(m);
14   }
15 }
```

Implementation

	Goblint		CPAchecker	
Entry type	Verifier	Validator	Verifier	Validator
Location invariant	✓	✓	✓	✓
Precondition invariant	✓	✓		
Flow-insensitive invariant	✓			
Protected by Location mutexes				

Conclusion

YAML format:

- Simpler to generate and validate by a non–model-checking tool
- Extensible
- Suitable for correctness witnesses for multi-threaded programs

Conclusion

YAML format:

- Simpler to generate and validate by a non–model-checking tool
- Extensible
- Suitable for correctness witnesses for multi-threaded programs

Our hopes

- Implemented in more tools
- More entry types proposed
- Actually used in SV-COMP as alternative to GraphML witnesses

Bibliography I

Beyer, D., Dangl, M., Dietsch, D., and Heizmann, M. (2016). Correctness witnesses: Exchanging verification results between verifiers. In *Foundations of Software Engineering*, pages 326–337. ACM.

Beyer, D., Spiessl, M., and Umbricht, S. (2022). Cooperation between automatic and interactive software verifiers. In *Software Engineering and Formal Methods*, pages 111–128. Springer.

Beyer, D. and Wehrheim, H. (2020). Verification artifacts in cooperative verification: Survey and unifying component framework. In *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, pages 143–167. Springer.

SoSy-Lab (2021). YAML-based exchange format for correctness witnesses. `https://gitlab.com/sosy-lab/benchmarking/sv-witnesses/-/blob/main/README-YAML.md`.

Bibliography II

Strejček, J. (2022). Notes illustrating some issues of the GraphML witness format. https://gitlab.com/sosy-lab/benchmarking/sv-witnesses/-/blob/main/GraphML_witness_format_issues.pdf.

Weise, N. (2021). A store for software invariants. In *Joint Workshop of the German Research Training Groups in Computer Science*, page 103. Erlangen.