IA159 Formal Methods for Software Analysis

Verification Witnesses, SV-COMP, and Test-Comp

Jan Strejček

Faculty of Informatics
Masaryk University

Focus and sources

focus

- verification witnesses in GraphML and YAML
- competition on software verification (SV-COMP)
- competition on software testing (Test-Comp)

sources

- D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig: Verification Witnesses, ACM TOSEM 2022.
- P. Ayaziová, D. Beyer, M. Lingsch-Rosenfeld, M. Spiessl, and J. Strejček: Software Verification Witnesses 2.0, SPIN 2024.

Motivation

- verification tools contain bugs
  - 72 verifiers participated in the Overall category of SV-COMP 2018-2023
  - How many provided only valid results?

Motivation

- verification tools contain bugs
  - 72 verifiers participated in the Overall category of SV-COMP 2018-2023
  - How many provided only valid results? Only 6!
  - 2 out of 17 in SV-COMP 2024
- verification result should be accompanied by a witness
- witness can be checked by independent witness validators
- $\Longrightarrow$ witness format needed

Motivation

- verification tools contain bugs
  - 72 verifiers participated in the Overall category of SV-COMP 2018-2023
  - How many provided only valid results? Only 6!
  - 2 out of 17 in SV-COMP 2024
- verification result should be accompanied by a witness
- witness can be checked by independent witness validators
- $\Longrightarrow$ witness format needed
- checking witness validity should be easier than deciding the verification task

Verification tasks and properties

verification task is a pair of

- program to be verified (usually in source code)
- property to be verified

Verification tasks and properties

verification task is a pair of

- program to be verified (usually in source code)
- property to be verified

common properties

- reachability safety
  - error locations/functions are unreachable
  - many specific properties can be reduced to this: division by zero, assertion checking, ...
- memory safety
  - no invalid pointer dereference, no invalid deallocation, no memory leaks
- no signed integer overflow
- program termination
- no data race, ...

Violation witnesses

- a violation witness represents property violation, i.e., some program execution violating the property
- witnesses representing only the violating execution would be big and hard to validate as they have to represent
    - values of all inputs
    - interaction with environment
    - thread scheduling
    - program non-determinism
        - order of evaluation of subexpressions: `f(x) + g(y)`
        - addresses of allocations: `p = malloc(10)`
    - …

Example of program nondeterminism

```c
int g = 0;

int f1() {
  g = 2 * g;
  return 5;
}

int f2() {
  g++;
  return 7;
}

int h() {
  return f1() + f2();
}

int main() {
  int c = h() + h();
  \\ what is the value of g here?
}
```

Example of program nondeterminism

```c
int g = 0;

int f1() {
  g = 2 * g;
  return 5;
}

int f2() {
  g++;
  return 7;
}

int h() {
  return f1() + f2();
}

int main() {
  int c = h() + h();
  \\ what is the value of g here?
}
```

3

Example of program nondeterminism

```
int g = 0;

int f1() {
  g = 2 * g;
  return 5;
}

int f2() {
  g++;
  return 7;
}

int h() {
  return f1() + f2();
}

int main() {
  int c = h() + h();
  \\ what is the value of g here?
}
```

3, 6

Example of program nondeterminism

```
int g = 0;

int f1() {
  g = 2 * g;
  return 5;
}

int f2() {
  g++;
  return 7;
}

int h() {
  return f1() + f2();
}

int main() {
  int c = h() + h();
  \\ what is the value of g here?
}
```

3, 6, 4, 5

Violation witnesses

- a violation witness represents property violation, i.e., some program execution violating the property
- witnesses representing only the violating execution would be big and hard to validate as they have to represent
  - values of all inputs
  - interaction with environment
  - thread scheduling
  - program non-determinism
    - order of evaluation of subexpressions: `f(x) + g(y)`
    - addresses of allocations: `p = malloc(10)`
  - . . .
- $\Longrightarrow$ a violation witness can represent more executions
- it is considered valid if at least one of the represented executions violates the property

Correctness witnesses

- a correctness witness provides arguments that the program is correct
- it provides invariants for some locations
- it is considered valid if
  1. all the provided invariants are indeed invariants and
  2. the program satisfies the property
- witness validity does not depend on the relevance of the invariants in the witness to the property satisfaction

Witness format 1.0

aka GraphML witness format

GraphML witness format aka witness format 1.0

- based on automata represented in GraphML
- format for violation witnesses introduced in 2015
- correctness witnesses added in 2016
- semantics defined in terms of control flow automata (CFA)

GraphML witness format aka witness format 1.0

- based on automata represented in GraphML
- format for violation witnesses introduced in 2015
- correctness witnesses added in 2016
- semantics defined in terms of control flow automata (CFA)

witness automaton

- a nondeterministic finite automaton accepting a set of program executions
- each edge is labelled with a pair (S, ψ) of
  - a source-code guard S representing a subset of CFA edges
  - a state-space guard ψ restricting the state space
- states are labelled with invariants
- sink states are non-accepting states without any successor

Source-code guards

- startline: *x* matches CFA edges that begin at line *x*
- endline: *x* matches CFA edges that end at line *x*
- startoffset: *x* matches CFA edges that start at column *x*
- endoffset: *x* matches CFA edges that end at column *x*
- control: *true*/*false* matches CFA edges entering *true*/*false* branch of a branching statement
- enterFunction: *name* matches calls of function *name*
- returnFromFunction: *name* matches returns from *name*
- enterLoopHead matches CFA edges entering a loop head (in CFA sense)
- support for concurrent programs: threadId and createThread
- an edge can contain more guards (all restrictions apply)

State-space guards

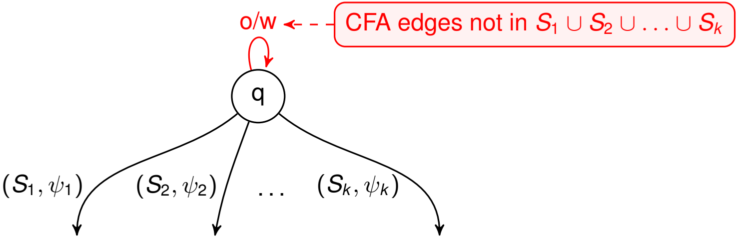
assumption: φ

- says that φ holds in the execution state immediately after the automaton edge is passed
- φ is an expression that
  - evaluates to a Boolean type or equivalent (`int` in C)
  - cannot contain function calls and cannot have any side-effects
  - can use only program variables and \result
- \result refers to the return value from the function given by assumption.resultfunction

Implicit loop edges

each non-sink state *q* has an implicit otherwise (o/w) loop with

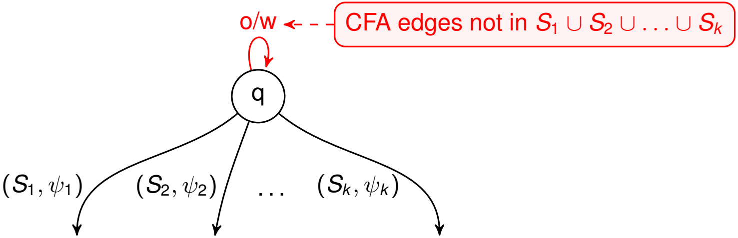
- source-code guard that corresponds to all CFA edges not matched by explicit edges of *q* and



Implicit loop edges

each non-sink state *q* has an implicit otherwise (o/w) loop with

- source-code guard that corresponds to all CFA edges not matched by explicit edges of *q* and



- state-space guard *true*

Violation witnesses

no invariants in automata states (*true* by default)

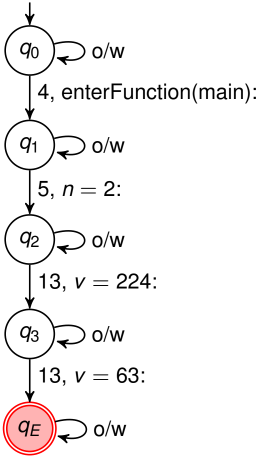
```c
void reach_error(){}
extern unsigned char __nondet_uchar(void);

int main() {
  unsigned char n = __nondet_uchar();
  if (n == 0) {
    return 0;
  }
  unsigned char v = 0;
  unsigned char s = 0;
  unsigned int i = 0;
  while (i < n) {
    v = __nondet_uchar();
    s += v;
    ++i;
  }
  if (s < v) {
    reach_error();
    return 1;
  }
  return 0;
}
```

Violation witnesses

no invariants in automata states (*true* by default)

```
 1 void reach_error(){}
 2 extern unsigned char __nondet_uchar(void);
 3
 4 int main() {
 5   unsigned char n = __nondet_uchar();
 6   if (n == 0) {
 7     return 0;
 8   }
 9   unsigned char v = 0;
10   unsigned char s = 0;
11   unsigned int i = 0;
12   while (i < n) {
13     v = __nondet_uchar();
14     s += v;
15     ++i;
16   }
17   if (s < v) {
18     reach_error();
19     return 1;
20   }
21   return 0;
22 }
```

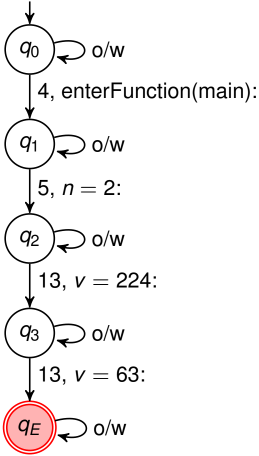


a label $S : \varphi$ is an abbreviation for (S, φ)
assumptions *true* are omitted

Violation witnesses

no invariants in automata states (*true* by default)

```
 1  void reach_error(){}
 2  extern unsigned char __nondet_uchar(void);
 3
 4  int main() {
 5   unsigned char n = __nondet_uchar();
 6   if (n == 0) {
 7     return 0;
 8   }
 9   unsigned char v = 0;
10   unsigned char s = 0;
11   unsigned int i = 0;
12   while (i < n) {
13     v = __nondet_uchar();
14     s += v;
15     ++i;
16   }
17   if (s < v) {
18     reach_error();
19     return 1;
20   }
21   return 0;
22  }
```

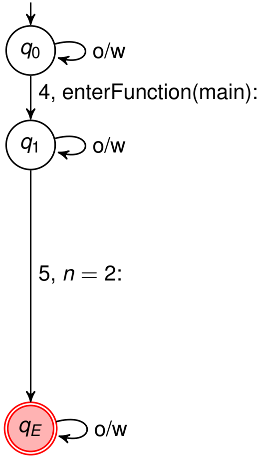


represents 1 violating execution

Violation witnesses

no invariants in automata states (*true* by default)

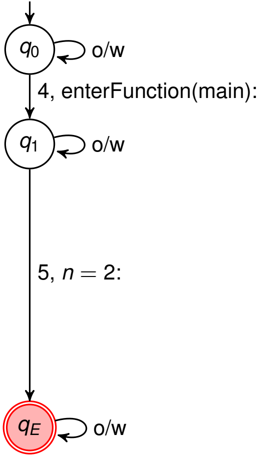
```
 1  void reach_error(){}
 2  extern unsigned char __nondet_uchar(void);
 3
 4  int main() {
 5   unsigned char n = __nondet_uchar();
 6   if (n == 0) {
 7     return 0;
 8   }
 9   unsigned char v = 0;
10   unsigned char s = 0;
11   unsigned int i = 0;
12   while (i < n) {
13     v = __nondet_uchar();
14     s += v;
15     ++i;
16   }
17   if (s < v) {
18     reach_error();
19     return 1;
20   }
21   return 0;
22  }
```



Violation witnesses

no invariants in automata states (*true* by default)

```
1  void reach_error(){}
2  extern unsigned char __nondet_uchar(void);
3
4  int main() {
5    unsigned char n = __nondet_uchar();
6    if (n == 0) {
7      return 0;
8    }
9    unsigned char v = 0;
10   unsigned char s = 0;
11   unsigned int i = 0;
12   while (i < n) {
13     v = __nondet_uchar();
14     s += v;
15     ++i;
16   }
17   if (s < v) {
18     reach_error();
19     return 1;
20   }
21   return 0;
22 }
```

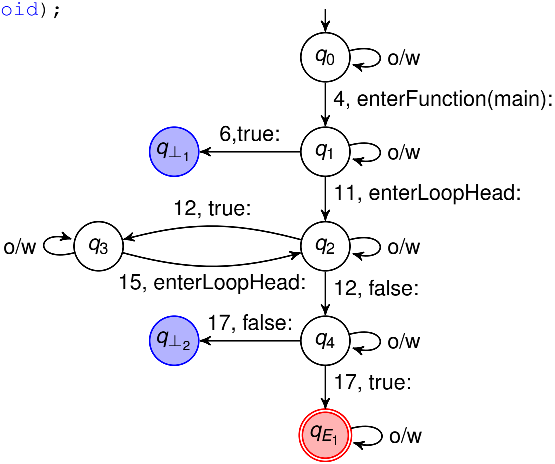


represents violating and non-violating executions

Violation witnesses

no invariants in automata states (*true* by default)

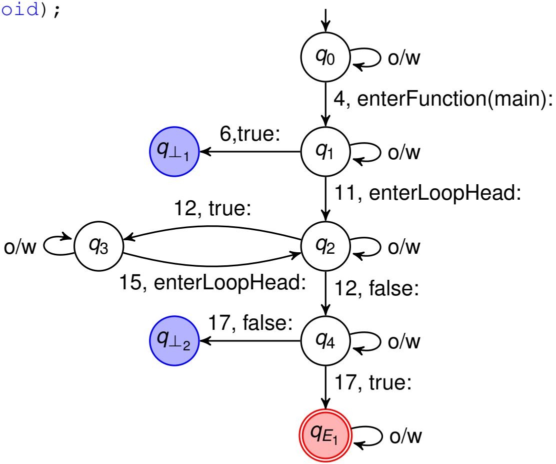
```c
 1  void reach_error(){}
 2  extern unsigned char __nondet_uchar(void);
 3
 4  int main() {
 5   unsigned char n = __nondet_uchar();
 6   if (n == 0) {
 7    return 0;
 8   }
 9   unsigned char v = 0;
10   unsigned char s = 0;
11   unsigned int i = 0;
12   while (i < n) {
13    v = __nondet_uchar();
14    s += v;
15    ++i;
16   }
17   if (s < v) {
18    reach_error();
19    return 1;
20   }
21   return 0;
22  }
```



Violation witnesses

no invariants in automata states (*true* by default)

```
 1  void reach_error(){}
 2  extern unsigned char __nondet_uchar(void);
 3
 4  int main() {
 5   unsigned char n = __nondet_uchar();
 6   if (n == 0) {
 7     return 0;
 8   }
 9   unsigned char v = 0;
10   unsigned char s = 0;
11   unsigned int i = 0;
12   while (i < n) {
13     v = __nondet_uchar();
14     s += v;
15     ++i;
16   }
17   if (s < v) {
18     reach_error();
19     return 1;
20   }
21   return 0;
22  }
```



valid, but not very useful witness

Correctness witnesses

have to accept all executions, no sink nodes

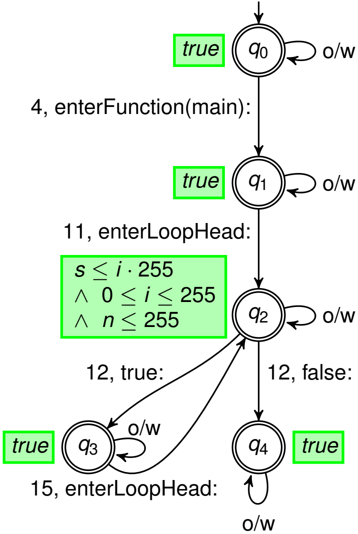
```
void reach_error(){}
extern unsigned char __nondet_uchar(void);

int main() {
 unsigned char n = __nondet_uchar();
 if (n == 0) {
  return 0;
 }
 unsigned char v = 0;
 unsigned int s = 0;
 unsigned int i = 0;
 while (i < n) {
  v = __nondet_uchar();
  s += v;
  ++i;
 }
 if (s < v) {
  reach_error();
  return 1;
 }
 return 0;
}
```

Correctness witnesses

have to accept all executions, no sink nodes

```
 1  void reach_error(){}
 2  extern unsigned char __nondet_uchar(void);
 3
 4  int main() {
 5    unsigned char n = __nondet_uchar();
 6    if (n == 0) {
 7      return 0;
 8    }
 9    unsigned char v = 0;
10    unsigned int s = 0;
11    unsigned int i = 0;
12    while (i < n) {
13      v = __nondet_uchar();
14      s += v;
15      ++i;
16    }
17    if (s < v) {
18      reach_error();
19      return 1;
20    }
21    return 0;
22  }
```



Notes

witnesses contain also metadata about

- the corresponding verification task
- the witnessed verification result
- producer of the witness
- considered architecture (32-bit or 64-bit)
- creation time

Notes

witnesses contain also metadata about

- the corresponding verification task
- the witnessed verification result
- producer of the witness
- considered architecture (32-bit or 64-bit)
- creation time

successes and fails of the GraphML format

+ widely accepted by the community
+ improved the quality of verification tools
+ other applications, e.g., cooperative verification

Notes

witnesses contain also metadata about

- the corresponding verification task
- the witnessed verification result
- producer of the witness
- considered architecture (32-bit or 64-bit)
- creation time

successes and fails of the GraphML format

+ widely accepted by the community
+ improved the quality of verification tools
+ other applications, e.g., cooperative verification
– witness validators do not support all features of the format
    - ignoring unsupported features may lead to incorrect verdict
– verifiers do not use the whole power of the format
– semantics given on CFA, but translation to CFA is ambiguous

Witness format 2.0

aka YAML witness format

Witness format 2.0

design goals

- clear semantics on source code
- validators that fully implement the format needed
- $\Longrightarrow$ as simple as possible

Witness format 2.0

design goals

- clear semantics on source code
- validators that fully implement the format needed
- $\Longrightarrow$ as simple as possible

design decisions

- start with the support of the most common properties and sequential programs and then extend it
- use YAML
- correctness witnesses specify invariants with the corresponding program locations
- violation witnesses describe executions with use of waypoints

Waypoints

waypoint = basic element of witnesses

Waypoints

waypoint = basic element of witnesses

each waypoint has 4 aspects:

- action - the role within the witness
- location - code location the waypoint is associated to
  - file_name
  - file_hash (optional)
  - line
  - column (optional, the default is the first suitable column)
- type - the type of constraint it puts on runs
- constraint - the constraint itself

Waypoint types

1. assumption
    - location: before a statement
    - constraint: a side-effect-free expression
    - for example `x[5] > z + 5` or `ptr != NULL`

Waypoint types

1. assumption
   - location: before a statement
   - constraint: a side-effect-free expression
   - for example `x[5] > z + 5` or `ptr != NULL`
2. branching
   - location: branching keyword like `if`, `while`, ...
   - constraint: `true` or `false`
   - specific support for `switch` statements

Waypoint types

1. assumption
    - location: before a statement
    - constraint: a side-effect-free expression
    - for example `x[5] > z + 5` or `ptr != NULL`
2. branching
    - location: branching keyword like `if`, `while`, . . .
    - constraint: `true` or `false`
    - specific support for `switch` statements
3. function_enter
    - location: the right parenthesis of the function call `foo()`
    - constraint: has to be omitted

Waypoint types

1. assumption
   - location: before a statement
   - constraint: a side-effect-free expression
   - for example `x[5] > z + 5` or `ptr != NULL`
2. branching
   - location: branching keyword like `if`, `while`, . . .
   - constraint: `true` or `false`
   - specific support for `switch` statements
3. function_enter
   - location: the right parenthesis of the function call `foo()`
   - constraint: has to be omitted
4. function_return
   - location: the right parenthesis of the function call `foo()`
   - constraint: `\result` *op* *const*, where *op* $\in \{==, != , <=, \ldots\}$ and *const* is a constant expression

Waypoint types

1. assumption
   - location: before a statement
   - constraint: a side-effect-free expression
   - for example `x[5] > z + 5` or `ptr != NULL`
2. branching
   - location: branching keyword like `if`, `while`, ...
   - constraint: `true` or `false`
   - specific support for `switch` statements
3. function_enter
   - location: the right parenthesis of the function call `foo()`
   - constraint: has to be omitted
4. function_return
   - location: the right parenthesis of the function call `foo()`
   - constraint: `\result` *op* *const*, where *op* $\in \{==, ! =, <=, \ldots\}$ and *const* is a constant expression
5. target
   - location: the statement that violates the property
   - constraint: has to be omitted

Waypoint actions

- follow - the waypoint has to be passed as soon as the location is entered
- avoid - the run represented by the witness must not pass the waypoint (“sink node”)

Waypoint actions

➡ follow - the waypoint has to be passed as soon as the location is entered

⛔ avoid - the run represented by the witness must not pass the waypoint (“sink node”)

STOP - follow waypoint of type target

Segments

- sequence of 0+ avoid waypoints ended by 1 follow or target waypoint
- segments ended by a follow waypoint are normal segments
- segments ended by a target waypoint are final segments

Segments

- sequence of 0+ avoid waypoints ended by 1 follow or target waypoint
- segments ended by a follow waypoint are normal segments
- segments ended by a target waypoint are final segments

a part of an execution matches a normal segment if

- the part ends by its first visit of the follow waypoint location and the constraint holds in this moment
- no avoid waypoint is passed (constraint is valid at the corresponding location) until that

Segments

- sequence of 0+ avoid waypoints ended by 1 follow or target waypoint
- segments ended by a follow waypoint are normal segments
- segments ended by a target waypoint are final segments

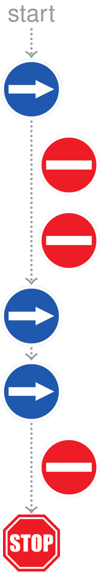
a part of an execution matches a normal segment if

- the part ends by its first visit of the follow waypoint location and the constraint holds in this moment
- no avoid waypoint is passed (constraint is valid at the corresponding location) until that

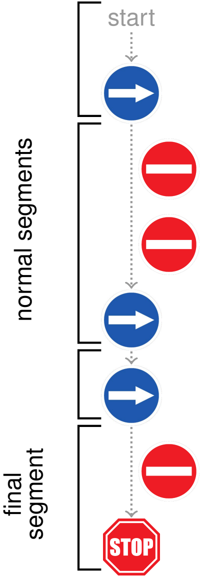
an execution matches a witness if

- it has a prefix that can be divided into parts that match the corresponding normal segments
- the rest does not pass any avoid waypoint of the final segment
- it violates the specified property by the target statement

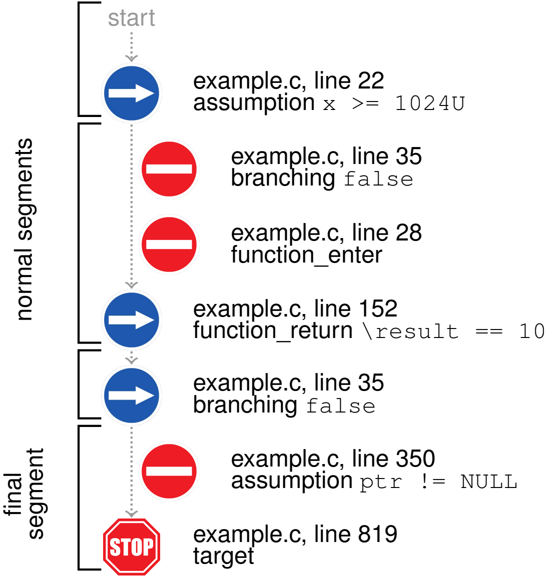
Witness example



Witness example



Witness example



start

normal segments

example.c, line 22
assumption `x >= 1024U`

example.c, line 35
branching `false`

example.c, line 28
function_enter

example.c, line 152
function_return `\result == 10`

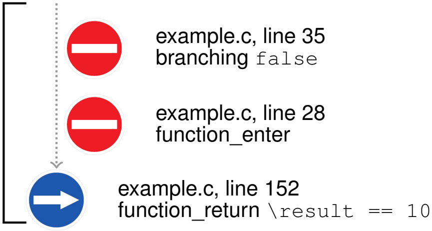
example.c, line 35
branching `false`

final segment

example.c, line 350
assumption `ptr != NULL`

example.c, line 819
target

Witness example - YAML notation



example.c, line 35
branching `false`

example.c, line 28
function_enter

example.c, line 152
function_return `\result == 10`

```yaml
- segment:
  - waypoint:
    action: avoid
    type: branching
    location:
      file_name: example.c
      line: 35
    constraint:
      value: false
  - waypoint:
    action: avoid
    type: function_enter
    location:
      file_name: example.c
      line: 28
  - waypoint:
    action: follow
    type: function_return
    location:
      file_name: example.c
      line: 152
    constraint:
      value: \result == 10
```

Correctness witnesses

```c
1  void reach_error(){}
2  extern unsigned char __nondet_uchar(void);
3
4  int main() {
5    unsigned char n = __nondet_uchar();
6    if (n == 0) {
7      return 0;
8    }
9    unsigned char v = 0;
10   unsigned int s = 0;
11   unsigned int i = 0;
12   while (i < n) {
13     v = __nondet_uchar();
14     s += v;
15     ++i;
16   }
17   if (s < v) {
18     reach_error();
19     return 1;
20   }
21   return 0;
22 }
```

```yaml
- entry_type: invariant_set
  metadata: <...>
  content:
  - invariant:
      type: loop_invariant
      location:
        file_name: "inv-a.c"
        line: 12
        column: 1
        function: main
      value: "s <= i*255 && 0 <= i && i <= 255 && n <= 255"
      format: c_expression
```

Notes

witness format 2.0

- published in 2024
- increasing number of verifiers and validators supporting the format
- better interoperability compared to the old format
- a bit less expressive than the old format
- should be the new standard of SV-COMP in several years
- needs to be extended to support
  - parallel programs
  - correctness witnesses for memory safety
  - violation and correctness witnesses of termination
  - . . .

Competition on Software Verification: SV-COMP

- running every year since 2012
- very popular and growing repository of C and Java verification tasks marked with expected results
- scoring schema

1	point for finding a program bug (if witness is validated)
2	points for proving correctness (if witness is validated)
-16	points for reporting bug in a correct program (false alarm)
-32	points for claiming correctness of an incorrect program (false negative)

  - points in the overall score are weighted by category sizes
- graphs indicate winners, speed, sequential portfolio of algorithms in tools, the number of incorrect answers, programming language of tools, . . .

`https://sv-comp.sosy-lab.org`

Competition on Software Testing: Test-Comp

- running every year since 2019
- uses the same benchmarks as SV-COMP
- the goal is to generate a test suit that
  - finds an error (category Cover-Error), benchmarks are programs with an error
  - has a high branch coverage (category Cover-Branches)

`https://test-comp.sosy-lab.org`