Handling Unbounded Loops with ESBMC 1.20

Jeremy Morse, Lucas Cordeiro Denis Nicole, Bernd Fischer







ESBMC: SMT-based BMC of singleand multi-threaded software

- exploits SMT solvers and their background theories:
 - optimized encodings for pointers, bit operations, unions and arithmetic over- and underflow
 - efficient search methods (non-chronological backtracking, conflict clauses learning)
- supports verifying multi-threaded software that uses pthreads threading library
 - interleaves only at "visible" instructions
 - *lazy exploration* of the reachability tree
 - optional context-bound
- derived from CBMC

ESBMC Verification Support

- built-in properties:
 - arithmetic under- and overflow
 - pointer safety
 - array bounds
 - division by zero
 - memory leaks
 - atomicity and order violations
 - deadlocks
 - data races
- user-specified assertions
 - (___ESBMC_assume, ___ESBMC_assert)
- built-in scheduling functions (__ESBMC_atomic_begin, __ESBMC_atomic_end, __ESBMC_yield)

Differences to ESBMC 1.17

- ESBMC 1.20 is largely a bugfixing release:
 - memory handling
 - internal data structure (replaced CBMC's string-based accessor functions)
 - Z3 encoding (replaced the name equivalence used in the pointer representation)
- improved our pthread-handling and added missing sequence points (pthread join-function)
- produces a smaller number of false results
 - score improvement of more than 25%
 - overall verification time reduced by about 25%

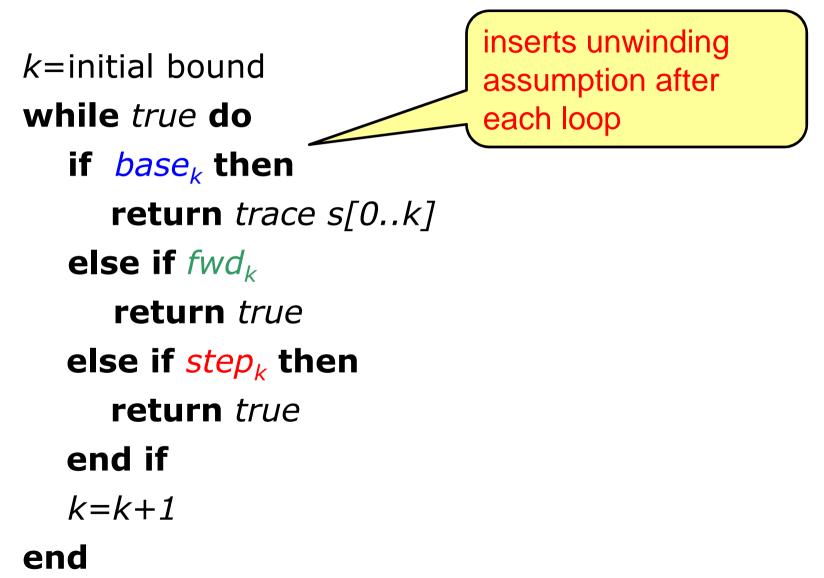
Induction-Based Verification

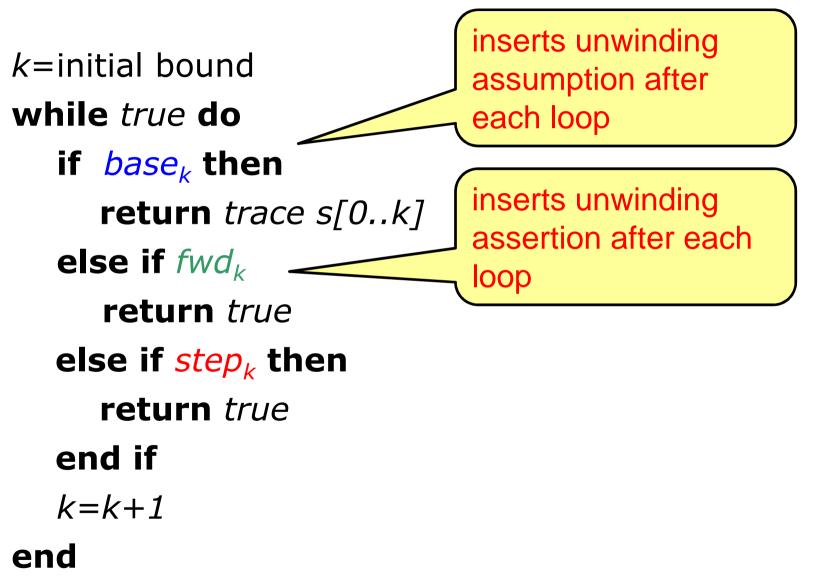
k-induction checks...

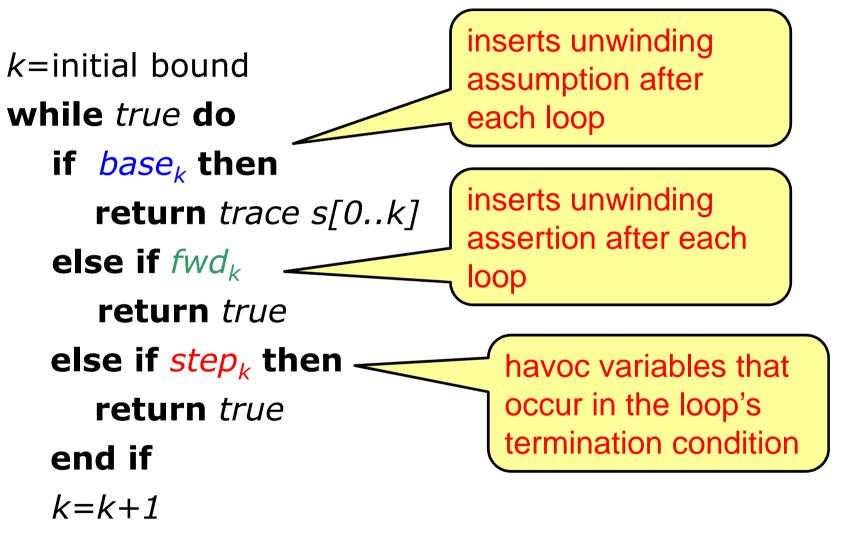
- base case (base_k): find a counter-example with up to k loop unwindings (plain BMC)
- forward condition (*fwd_k*): check that *P* holds in all states reachable within *k* unwindings
- inductive step (step_k): check that whenever P holds for k unwindings, it also holds after next unwinding
 - havoc state
 - run k iterations
 - assume invariant
 - run final iteration
- \Rightarrow iterative deepening if inconclusive

k=initial bound while true do if *base*_k then **return** *trace s*[0..*k*] else if *fwd*_k return true else if step_k then return true end if k=k+1

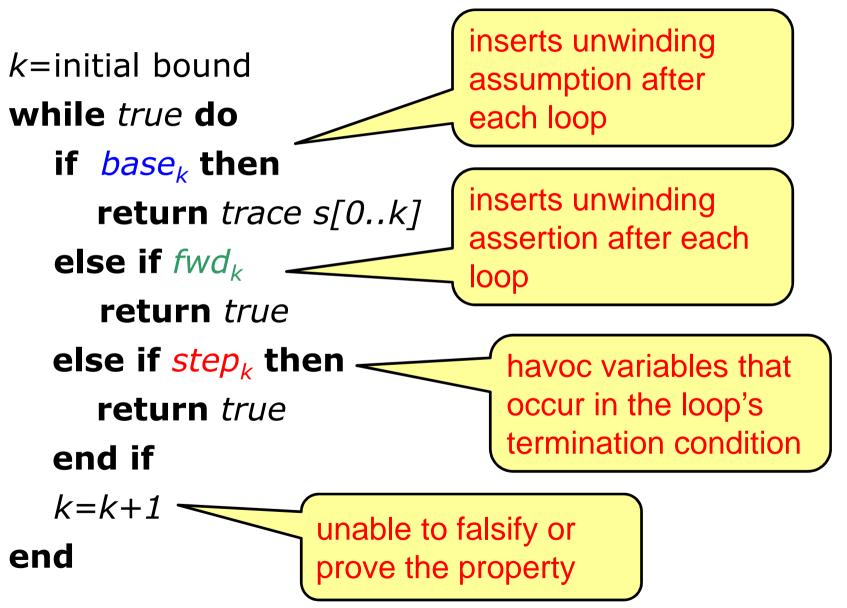
end







end



Running example

Prove that
$$S_n = \sum_{i=1}^n a = na$$
 for $n \ge 1$

```
unsigned int nondet_uint();
int main() {
    unsigned int i, n=nondet_uint(), sn=0;
    assume (n>=1);
    for(i=1; i<=n; i++)
        sn = sn + a;
    assert(sn==n*a);
}</pre>
```

Running example: base case

Insert an **unwinding assumption** consisting of the termination condition after the loop

- find a counter-example with k loop unwindings

```
unsigned int nondet_uint();
int main() {
 unsigned int i, n=nondet_uint(), sn=0;
 assume (n \ge 1);
 for(i=1; i<=n; i++)
  sn = sn + a;
 assume(i>n);
 assert(sn==n*a);
```

Running example: forward condition

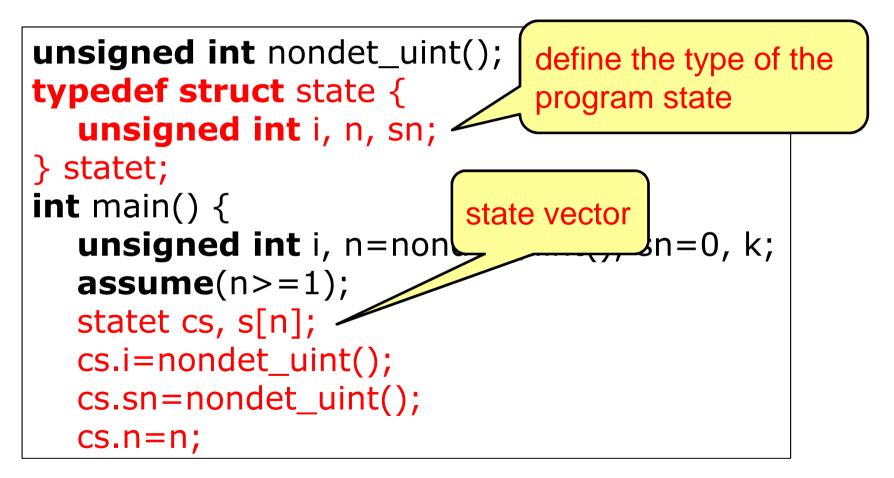
Insert an **unwinding assertion** consisting of the termination condition after the loop

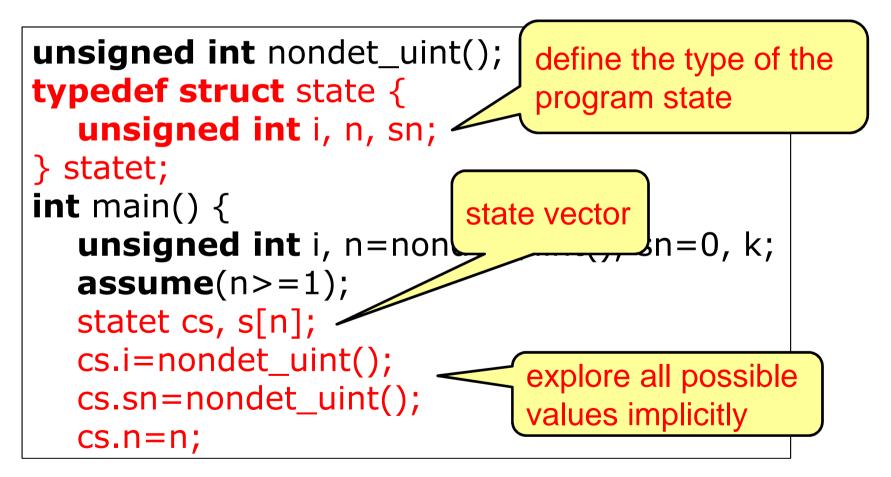
- check that P holds in all states reachable with k unwindings

```
unsigned int nondet_uint();
int main() {
 unsigned int i, n=nondet_uint(), sn=0;
 assume (n \ge 1);
 for(i=1; i<=n; i++)
  sn = sn + a;
 assert(i>n);
 assert(sn==n*a);
```

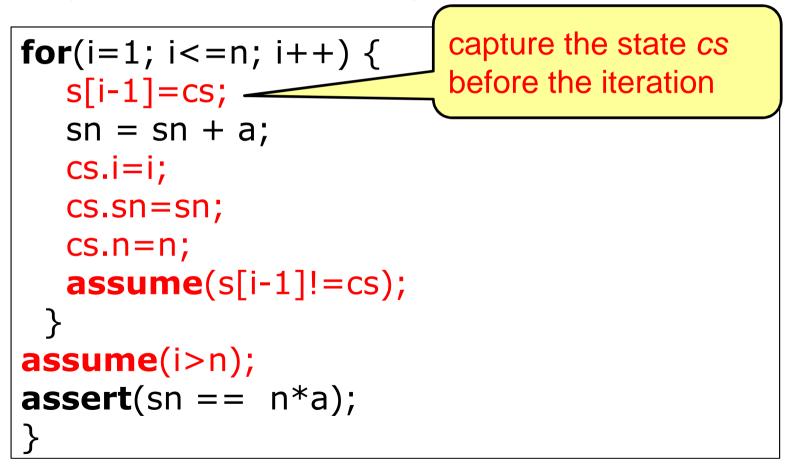
```
unsigned int nondet_uint();
typedef struct state {
  unsigned int i, n, sn;
} statet;
int main() {
  unsigned int i, n=nondet_uint(), sn=0, k;
  assume(n>=1);
  statet cs, s[n];
  cs.i=nondet_uint();
  cs.sn=nondet_uint();
  cs.n=n;
```

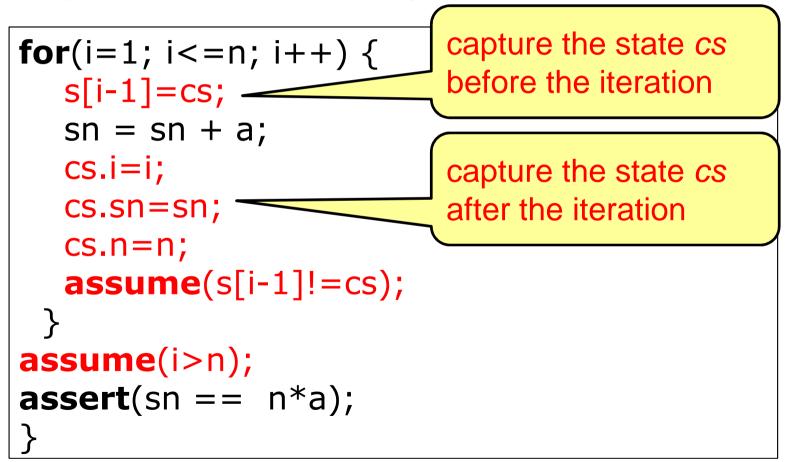
```
unsigned int nondet_uint();
                               define the type of the
typedef struct state {
                               program state
  unsigned int i, n, sn;
} statet;
int main() {
  unsigned int i, n=nondet_uint(), sn=0, k;
  assume(n>=1);
  statet cs, s[n];
  cs.i=nondet_uint();
  cs.sn=nondet_uint();
   cs.n=n;
```

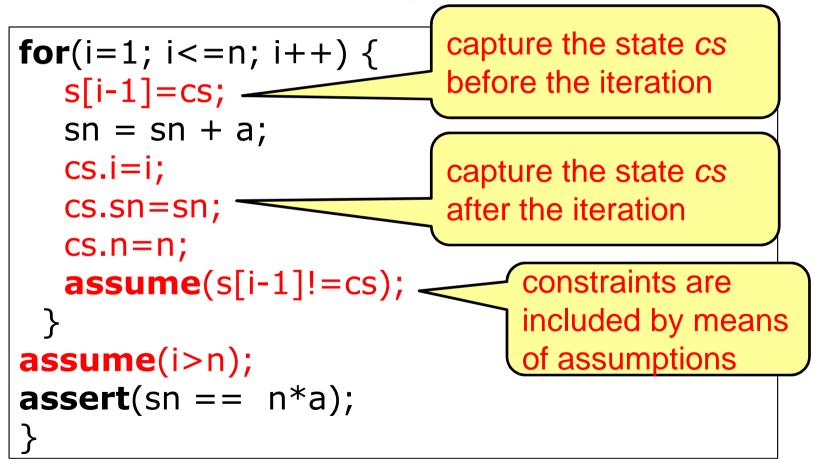


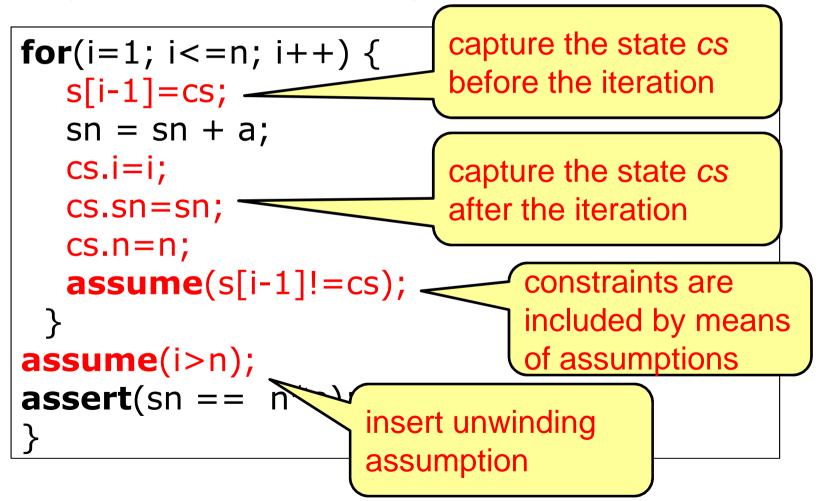


```
for(i=1; i<=n; i++) {
  s[i-1]=cs;
  sn = sn + a;
  cs.i=i;
  cs.sn=sn;
  cs.n=n;
  assume(s[i-1]!=cs);
 }
assume(i>n);
assert(sn == n*a);
```









Strengths:

- robust context-bounded model checker for multithreaded C code
- combines plain BMC with k-induction
 - k-induction by itself is by far not as strong as plain BMC
 - \Rightarrow although it produced substantially fewer false results

Strengths:

- robust context-bounded model checker for multithreaded C code
- combines plain BMC with k-induction
 - *k*-induction by itself is by far not as strong as plain BMC \Rightarrow although it produced substantially fewer false results

Weaknesses:

- scalability (like other BMCs...)
 - loop unrolling
 - interleavings
- pointer handling and points-to analysis
 - exposed by excessive typecasts in the CIL-converted code
 - better memory model in progress