Handling Loops in Bounded Model Checking of C Programs via *k*-Induction

Lucas Cordeiro

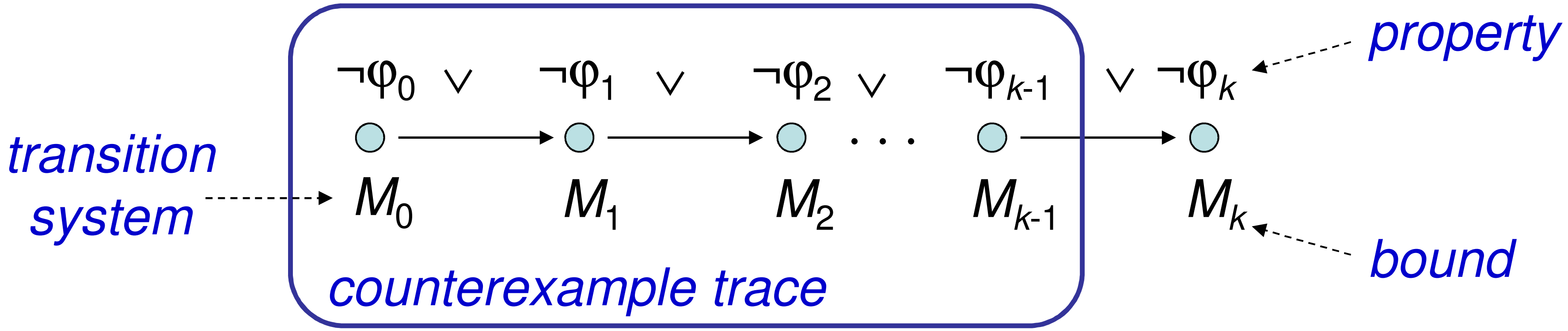
Joint work with
Jeremy Morse, Mikhail Ramalho, Herberto Rocha, Hussama Ismail, Raimundo Barreto, Denis Nicole, and Bernd Fischer

UFAM

UNIVERSITY OF Southampton
School of Electronics and Computer Science

Bounded Model Checking (BMC)

basic Idea: check negation of given property up to given depth



- transition system M unrolled k times
  - for programs: loops, arrays, …
- translated into verification condition ψ such that

  ψ satisfiable iff φ has counterexample of max. depth k

- has been applied successfully to verify (embedded) software

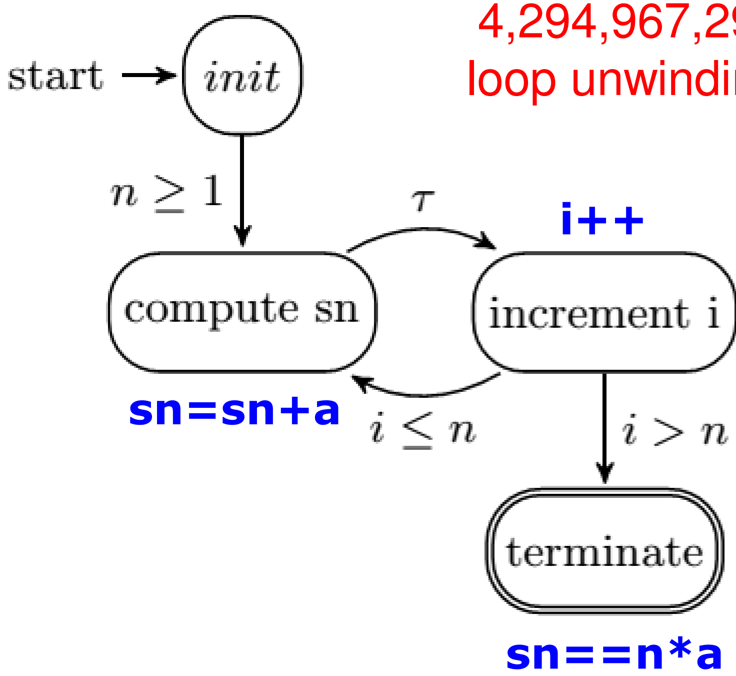
Difficulties in proving the correctness of programs with loops in BMC

- BMC techniques can falsify properties up to a given depth *k*
  - they can prove correctness only if an upper bound of *k* is known (**unwinding assertion**)
    - » BMC tools typically fail to verify programs that contain bounded and unbounded loops

$$S_n = \sum_{i=1}^{n} a = na, n \geq 1$$

the loop will be unfolded 2^{n-1} times (in the worst case, 2^{32-1} times on 32 bits integer)

4,294,967,295 loop unwindings



ESBMC: SMT-based BMC of single- and multi-threaded software

Goal: prove that an invariant is *k*-inductive

SMT-b … n CBMC:

- symbolicall … utes C into SSA, produces QF formulae
- unrolls loops up to a maximum bound *k*
- assertion failure *iff* corresponding formula is satisfiable
  - safety properties (array bounds, pointer dereferences, overflows,...)
  - user-specified properties

Multi-threaded programs:

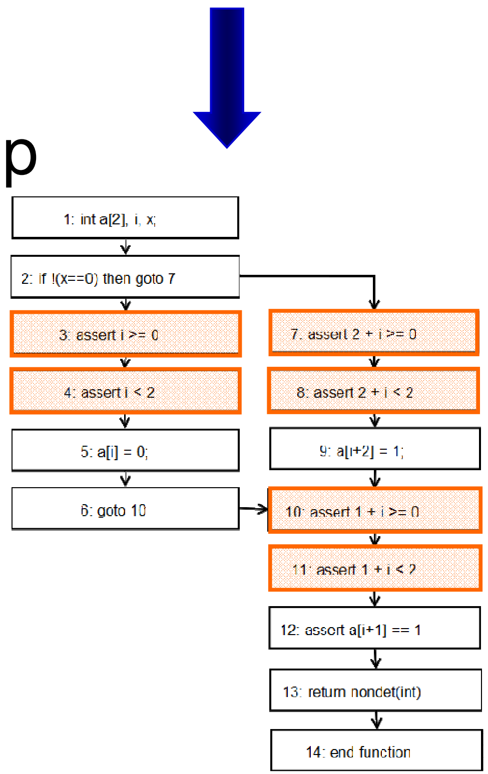
- produces one SSA program for each possible thread interleaving
- interleaves only at “visible” instructions
- optional context bound

Software BMC using ESBMC

- program modelled as state transition system
  - *state*: program counter and program variables
  - derived from control-flow graph
  - checked safety properties give extra nodes
- program unfolded up to given bounds
  - loop iterations
  - context switches
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions

  } crucial

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```



Software BMC using ESBMC

- program modelled as state transition system
  - *state*: program counter and program variables
  - derived from control-flow graph
  - checked safety properties give extra nodes
- program unfolded up to given bounds
  - loop iterations
  - context switches
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions

  } crucial
- front-end converts unrolled and optimized program into SSA

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```

$g_1 = x_1 == 0$
$a_1 = a_0 \text{ WITH } [i_0{:=}0]$
$a_2 = a_0$
$a_3 = a_2 \text{ WITH } [2+i_0{:=}1]$
$a_4 = g_1 \; ? \; a_1 : a_3$
$t_1 = a_4[1+i_0] == 1$

Software BMC using ESBMC

- program modelled as state transition system
  - *state*: program counter and program variables
  - derived from control-flow graph
  - checked safety properties give extra nodes
- program unfolded up to given bounds
  - loop iterations
  - context switches
- unfolded program optimized to reduce blow-up
  - constant propagation } crucial
  - forward substitutions }
- front-end converts unrolled and optimized program into SSA
- extraction of *constraints C* and *properties P*
  - specific to selected SMT solver, uses theories
- satisfiability check of $C \wedge \neg P$

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```

$$C := \begin{bmatrix} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{bmatrix}$$

$$P := \begin{bmatrix} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{bmatrix}$$

Software BMC using ESBMC

- program modelled as state transition system
  - *state*: program counter and program variables
  - derived from control-flow graph
  - checked safety properties give extra nodes
- program unfolded up to given bounds
  - lo
  - co
- unfo
  - constant propagation
  - forward substitutions

  } crucial
- front-end converts unrolled and optimized program into SSA
- extraction of *constraints C* and *properties P*
  - specific to selected SMT solver, uses theories
- satisfiability check of $C \wedge \neg P$

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
```

ESBMC finds real errors in applications, but it is susceptible to producing time-out or memory-out for correct programs

$$C := \left[\begin{array}{l} \ldots = 0) \\ \wedge a_1 := store(a_0, i_0, 0) \\ \wedge a_2 := a_0 \\ \wedge a_3 := store(a_2, 2+i_0, 1) \\ \wedge a_4 := ite(g_1, a_1, a_3) \end{array}\right]$$

$$P := \left[\begin{array}{l} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge 2+i_0 \geq 0 \wedge 2+i_0 < 2 \\ \wedge 1+i_0 \geq 0 \wedge 1+i_0 < 2 \\ \wedge select(a_4, i_0+1) = 1 \end{array}\right]$$

Induction-Based Verification

***k*-induction** checks loop-free programs...

- **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

⇒ iterative deepening if inconclusive

Loop-free Programs (*base*$_k$ and *fwd*$_k$)

- A loop-free program is represented by a **straight-line program** (without loops) using *if*-statements

```
for(B; c; D) { E; }   ⇒   B while(c) { E; D;}
```

```
L1: while(c) {             L1: if(!c) goto L2
      E; D;          ⇒           E; D;
    }                            goto L1
                           L2: ASSUME or ASSERT
```

```
L1: while(cond1) {         L1: if(!cond1) goto L4
      LOOP1 BODY                   LOOP1 BODY
L2:   while(cond2) {  ⇒    L2:     if(!cond2) goto L3
        LOOP2 BODY                   LOOP2 BODY
      }                              goto L2
    }                      L3:       goto L1
                           L4:  ASSUME or ASSERT
```

Loop-free Programs (*base*$_k$ and *fwd*$_k$)

- A loop-free program is represented by a **straight-line program** (without loops) using *if*-statements

```
for(B; c; D) { E; }   ⇒   B while(c) { E; D;}
```

base$_k$ and _fwd_$_k$ translations can easily be implemented on top of plain BMC

```
L1                                  2
                          L2: ASSUME or ASSERT
```

```
L1: while(cond1) {
      LOOP1 BODY
L2:   while(cond2) {
        LOOP2 BODY
      }
    }
```

⇒

```
L1: if(!cond1) goto L4
      LOOP1 BODY
L2:   if(!cond2) goto L3
        LOOP2 BODY
        goto L2
L3:     goto L1
L4:  ASSUME or ASSERT
```

Loop-free Programs (*step$_k$*)

- In the inductive step, loops are converted into:

```
while(c) { E; }        ⇒        A while(c) { S; E; U; } R;
```

  - **A:** assigns **non-deterministic values** to all loops variables (the state is havocked before the loop)
  - **c:** is the **halt condition** of the loop
  - **S: stores the current state** of the program variables before executing the statements of E
  - **E:** is the actual **code inside the loop**
  - **U: updates all program variables** with local values after executing E

The *k*-induction algorithm

```
k=1
while k<=max_iterations do
    if  base_{P,ϕ,k} then
        return trace s[0..k]
    else
      k=k+1
      if fwd_{P,ϕ,k} then
        return true
      else if step_{P′,ϕ,k} then
        return true
    end if
end
return unknown
```

The *k*-induction algorithm

I : initial condition
T : transition relation of P
σ : termination condition
ϕ : safey property

$I \wedge T \wedge \sigma \Rightarrow \phi$

inserts **unwinding assumption** after each loop

k=1
while *k<=max_iterations* **do**
  if $base_{P,\phi,k}$ **then**
    return *trace s[0..k]*
  else
    k=k+1
    if $fwd_{P,\phi,k}$ **then**
      return *true*
    else if $step_{P',\phi,k}$ **then**
      return *true*
  end if
end
return *unknown*

The *k*-induction algorithm

```
k=1
while k<=max_iterations do
    if base_{P,φ,k} then
        return trace s[0..k]
    else
      k=k+1
      if fwd_{P,φ,k} then
        return true
      else if step_{P',φ,k} then
        return true
    end if
end
return unknown
```

$I \wedge T \wedge \sigma \Rightarrow \phi$

$I \wedge T \Rightarrow \sigma \wedge \phi$

inserts **unwinding assertion** after each loop

The *k*-induction algorithm

```
k=1
while k<=max_iterations do
    if  base_{P,ϕ,k} then
        return trace s[0..k]
    else
      k=k+1
      if fwd_{P,ϕ,k} then
        return true
      else if step_{P′,ϕ,k} then
        return true
    end if
end
return unknown
```

$I \wedge T \wedge \sigma \Rightarrow \phi$

γ: transition relation of *P′*

$\gamma \wedge \sigma \Rightarrow \phi$

havoc variables that occur in the loop's termination condition

The *k*-induction algorithm

k=1

while *k<=max_iterations* **do**

  if $base_{P,\phi,k}$ **then** ← $I \wedge T \wedge \sigma \Rightarrow \phi$

   return *trace s[0..k]*

  else

   k=k+1

   if $fwd_{P,\phi,k}$ **then** ← $I \wedge T \Rightarrow \sigma \wedge \phi$

    return *true*

   else if $step_{P',\phi,k}$ **then** ← $\gamma \wedge \sigma \Rightarrow \phi$

    return *true*

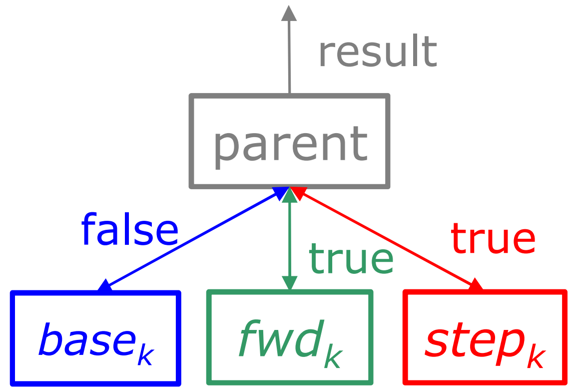
  end if

end

return *unknown*

> unable to **falsify** or **prove** the property

Parallel *k*-Induction Algorithm



- The parallel implementation consists of **four different processes**
  - running in different processing cores
  - splitting each step potentially **divides the work clock-time by a factor of three**
- Parent process initializes three child processes, executes the logic of the *k*-induction algorithm, and shows the verification results
  - **two pipes** are used in each process for the **inter-process communication**
- Once the solution is found, the child process communicates to the parent process, which sends signals to the other two child processes to finalize them

Running example

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

```
int main() {
  long long int i=1, sn=0;
  unsigned int n;
  assume (n>=1);
  while(i<=n) {
    sn = sn + a;
    i++;
  }
  assert(sn==n*a);
}
```

Running example: *base case*

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

– find a counter-example with *k* loop unwindings

```
int main() {
  unsigned int n=nondet_uint();
  long long int i=1, sn=0;
  assume (n>=1);
  if (i<=n) {          ⎫
    sn = sn + a;       ⎬  k-copies
    i++;               ⎪
  }                    ⎭
  ...
  assume(i>n); //unwinding assumption
  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

```
int main() {
  unsigned int n=nondet_uint();
  long long int i=1, sn=0;
  assume (n>=1);
  if (i<=n) {
    sn = sn + a;        } k-copies
    i++;
  }
  ...
  assert(i>n);  //unwinding assertion
  assert(sn==n*a);
}
```

Running example: *inductive step*

Havoc (only) the variables that occur in the loop's termination and branch conditions

```
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, sv[n];
    cs.i=nondet_uint();
    cs.sn=nondet_uint();
    cs.n=n;
```

Running example: *inductive step*

Havoc (only) the variables that occur in the loop's termination and branch conditions

```
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, sv[n];
    cs.i=nondet_uint();
    cs.sn=nondet_uint();
    cs.n=n;
```

define the type of the program state

Running example: *inductive step*

Havoc (only) the variables that occur in the loop's termination and branch conditions

```
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, sv[n];
    cs.i=nondet_uint();
    cs.sn=nondet_uint();
    cs.n=n;
```

define the type of the program state

state vector

Running example: *inductive step*

Havoc (only) the variables that occur in the loop's termination and branch conditions

```
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, sv[n];
    cs.i=nondet_uint();
    cs.sn=nondet_uint();
    cs.n=n;
```

define the type of the program state

state vector

explore all possible values implicitly

Running example: *inductive step*

ESBMC is called to verify the assertions where the first arbitrary state is emulated by **nondeterminism**

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

Running example: *inductive step*

ESBMC is called to verify the assertions where the first arbitrary state is emulated by **nondeterminism**

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

capture the state *cs* before the iteration

Running example: *inductive step*

ESBMC is called to verify the assertions where the first arbitrary state is emulated by **nondeterminism**

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

capture the state *cs* before the iteration

capture the state cs after the iteration

Running example: *inductive step*

ESBMC is called to verify the assertions where the first arbitrary state is emulated by **nondeterminism**

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

capture the state *cs* before the iteration

capture the state cs after the iteration

constraints are included by means of assumptions

Running example: ***inductive step***

ESBMC is called to verify the assertions where the first arbitrary state is emulated by **nondeterminism**

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

capture the state *cs* before the iteration

capture the state cs after the iteration

constraints are included by means of assumptions

insert unwinding assumption

Removing Redundant States

- An assume instruction checks whether the current state is different from the previous one
  - prevent redundant states to be inserted into the state vector

```
assume(sv[i-1]!=cs);
```

- We compare $sv_j[i]$ to cs_j for $0 < j \leq k$ and $0 \leq i \leq k$

$$sv_1[0] \neq cs_1$$

$$sv_1[0] \neq cs_1 \wedge sv_2[1] \neq cs_2$$

...

$$sv_1[0] \neq cs_1 \wedge sv_2[1] \neq cs_2 \wedge \ldots \wedge sv_k[i] \neq cs_k$$

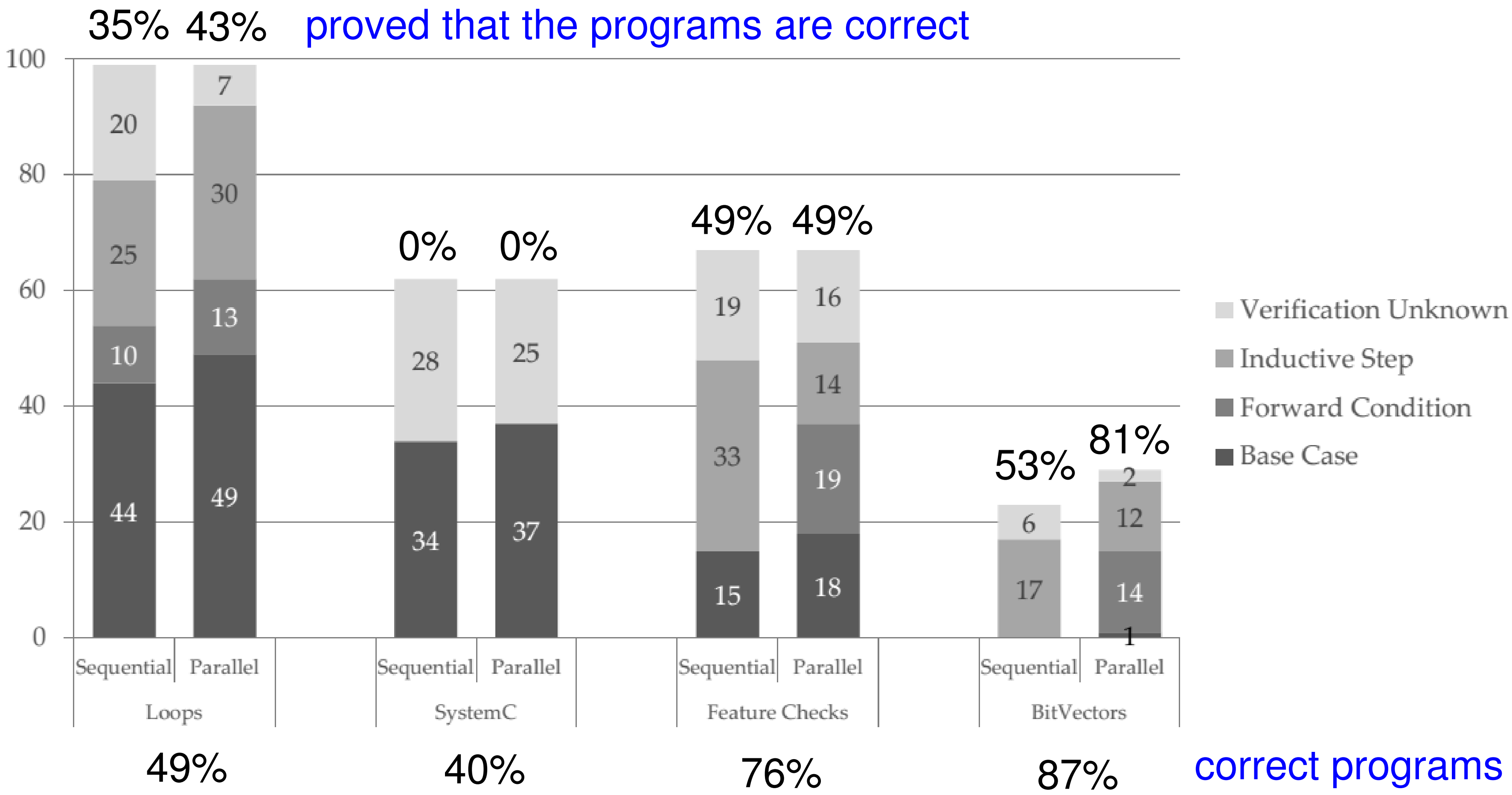
- We could compare $sv_k[i]$ to all cs_k for $i < k$ (since **inequalities are not transitive**)
  - however, the number of constraints can grow very large quickly

Experimental Evaluation

- **Goal:** evaluate the performance of the sequential and parallel implementations using the SV-COMP benchmarks
  - **Loops** (99 programs)
    - » 49% correct and 51% incorrect programs
  - **SystemC** (62 programs)
    - » 40% correct and 60% incorrect programs
  - **FeatureChecks** (67 programs)
    - » 76% correct and 24% incorrect programs
  - ***BitVectors*** (32 programs)
    - » 87% correct and 13% incorrect programs
- Set-up:
  - ESBMC v1.22 together with the SMT solver Z3 v4.0
  - support the logics *QF_AUFBV* and *QF_AUFLIRA*
  - standard desktop PC, time-out 900 seconds

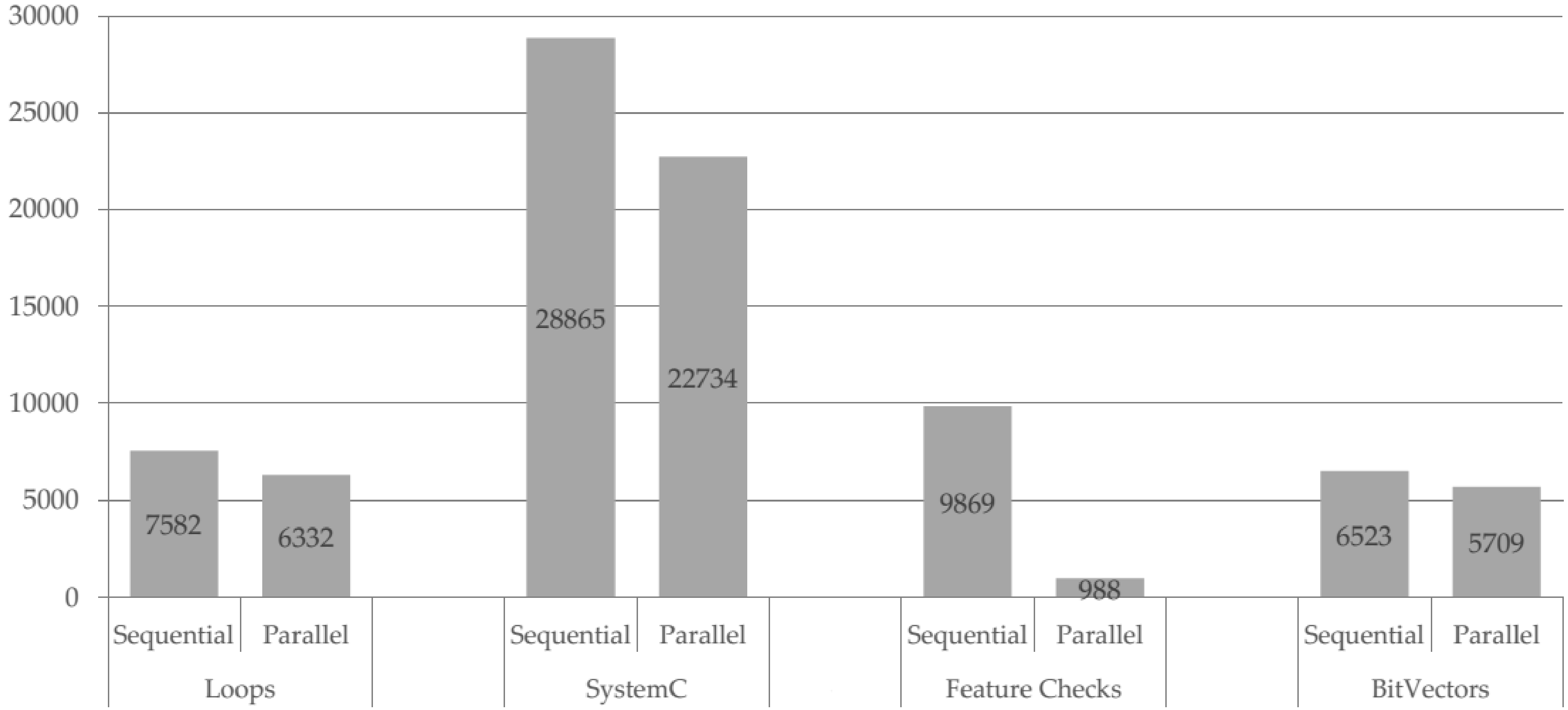
Verification Results for Each Step

- Most of **unknown** results occurred due to nested loops
  - base case produced two **false alarms** due to the memory model adopted by ESBMC



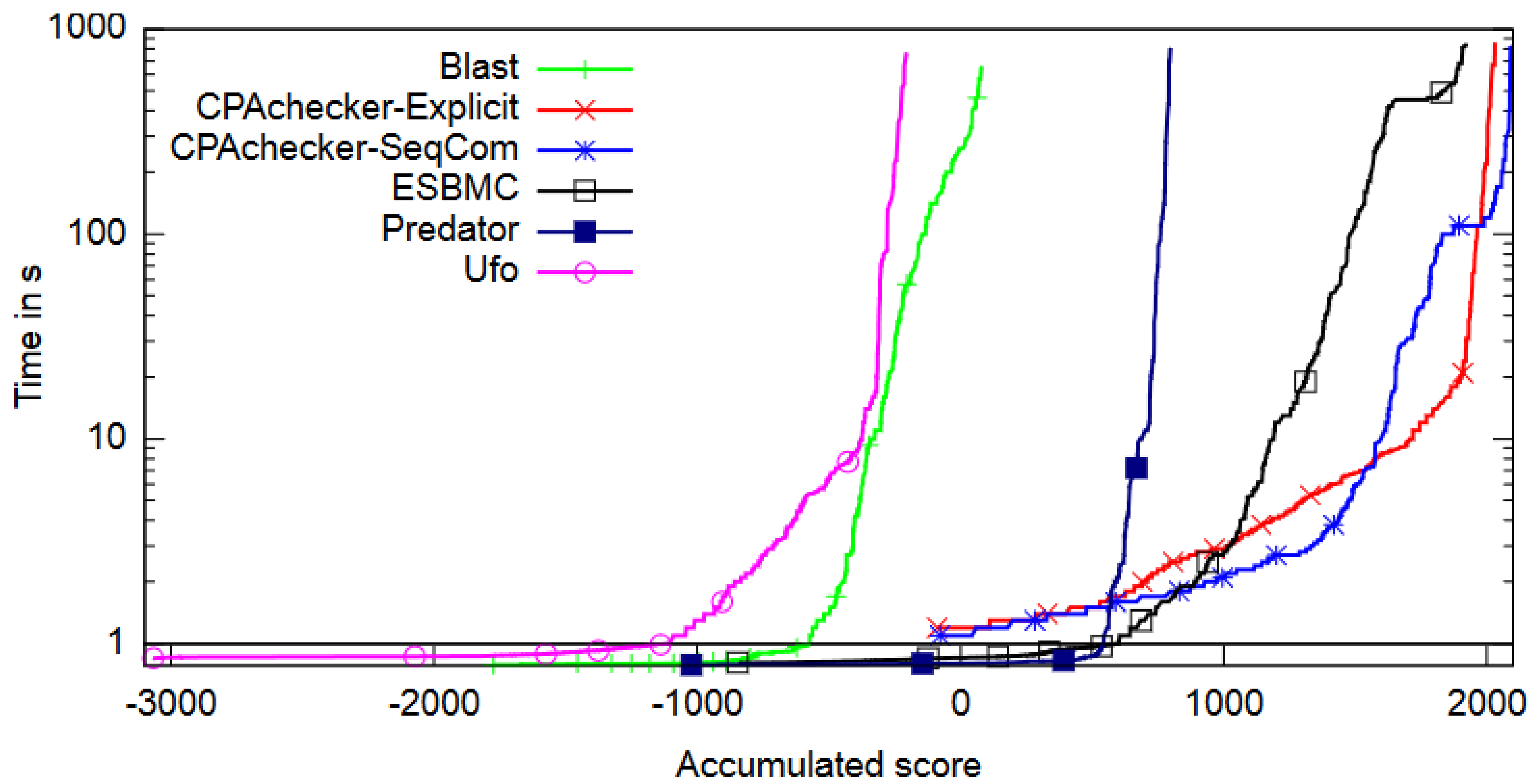
Verification Time per Category

- Sequential *k*-induction verifies 70% of the benchmarks in 52839 seconds, and the parallel *k*-induction verifies 80% in 35763 seconds
  - » **speedup of 32%**



SV-COMP 2013 Results – Overall Ranking

- Sequential *k*-induction participated in the 2nd edition of the SV-COMP
  - **verify by induction** that the safety property holds
    - » If that fails, **search for a bounded reachable state**



Strengths:

- robust *k*-induction algorithm for C programs
  - this marks the first application of the *k*-induction algorithm to a broader range of C programs
- 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 *k*-induction algorithm for C programs
  - this marks the first application of the *k*-induction algorithm to a broader range of C programs
- 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
- investigate whether redundant constraints can be avoided
  - using the results of already completed steps
- refine invariants to strengthen the induction hypothesis