FEDERAL UNIVERSITY OF RORAIMA
and
FEDERAL UNIVESITY OF AMAZONAS

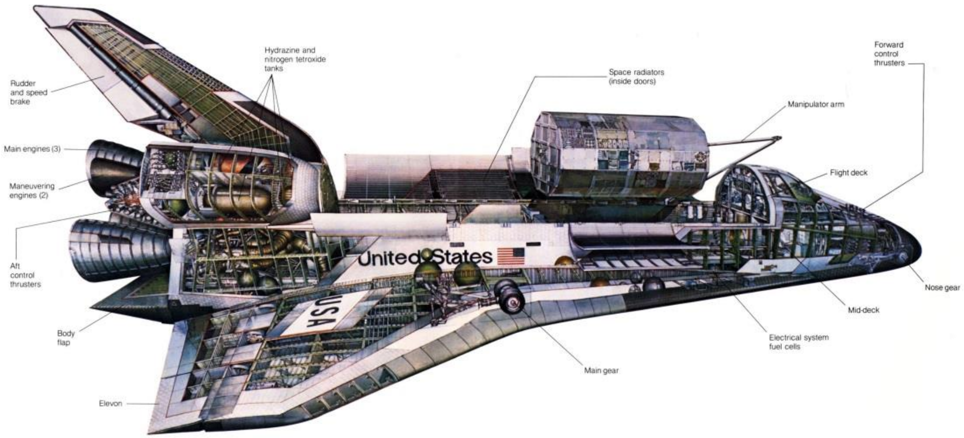
Model Checking Embedded C Software using *k*-Induction and Invariants

Herbert Rocha, Hussama Ismail,
Lucas Cordeiro and Raimundo Barreto

Agenda

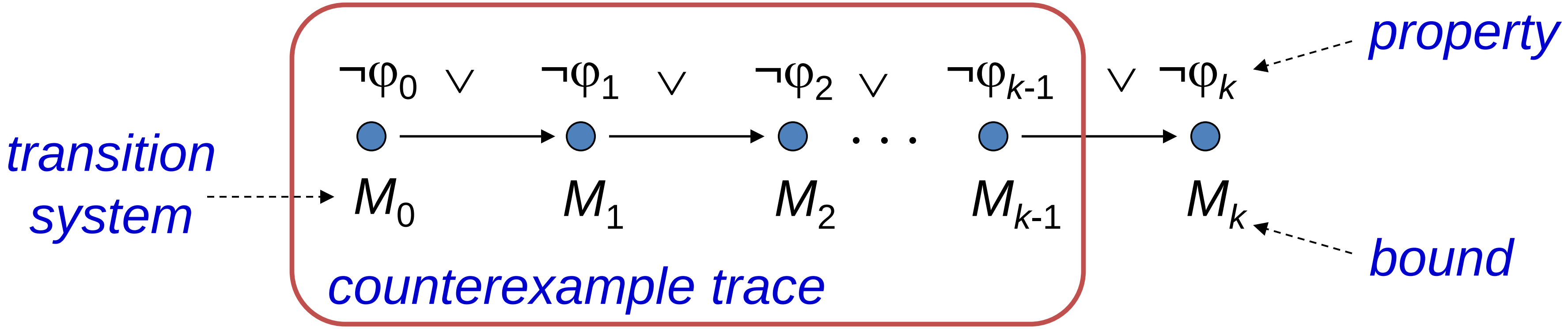
1. Introduction
2. Background
3. Proposed Method
4. Experimental Evaluation
5. Related Work
6. Conclusions and Future Work

Software Applications



Bounded Model Checking (BMC)

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



property

$\neg\varphi_0 \lor \neg\varphi_1 \lor \neg\varphi_2 \lor \neg\varphi_{k-1} \lor \neg\varphi_k$

transition system

$M_0 \quad M_1 \quad M_2 \quad \ldots \quad M_{k-1} \quad M_k$

bound

counterexample trace

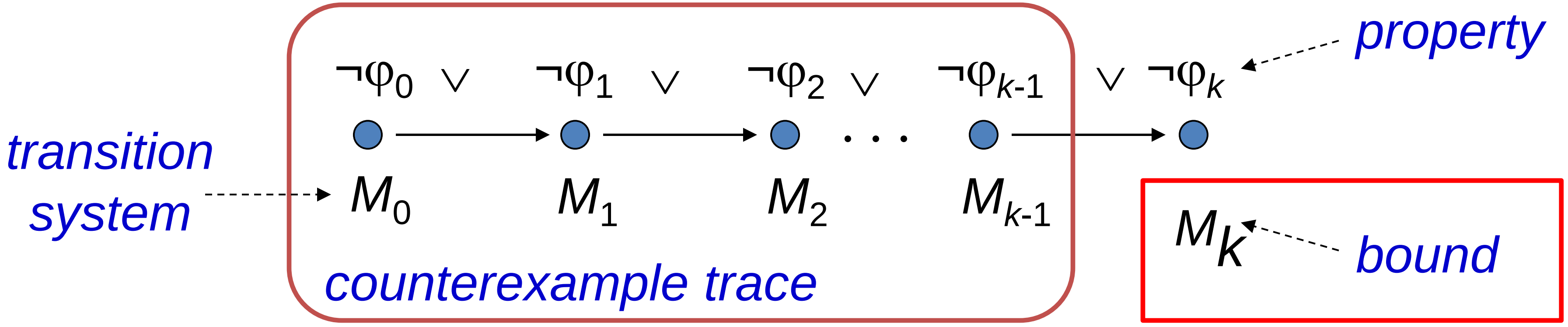
- transition system *M* unrolled *k* times
  - for programs: loops, arrays, …
- translated into verification condition ψ such that

ψ satisfiable iff φ has counterexample of max. depth *k*

- has been applied successfully to verify (embedded) software

Bounded Model Checking (BMC)

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


$\neg\varphi_0 \vee \neg\varphi_1 \vee \neg\varphi_2 \vee \cdots \neg\varphi_{k-1} \vee \neg\varphi_k$

$M_0 \rightarrow M_1 \rightarrow M_2 \cdots M_{k-1} \rightarrow M_k$

- BMC techniques limit the visited regions of data structures (*e.g.*, arrays) and the number of loop iterations.
- BMC tools are susceptible to exhaustion of time or memory limits for programs with **loops**.

Example

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

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

```

Bound ***loop***

Example

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

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

```

Bound ***loop***

For a 32 bits integer, the loop will be unfolded $2^n - 1\ times =$ **4,294,967,295 times**

Difficulties in proving the correctness of programs with loops in BMC

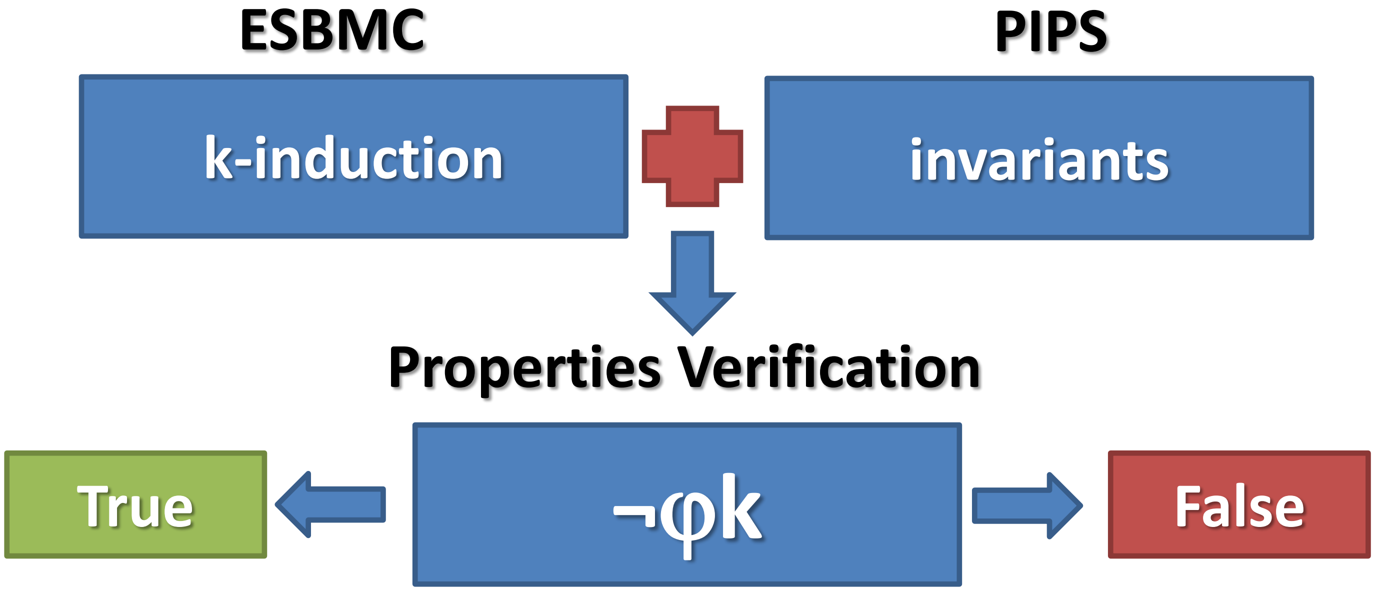
- BMC techniques can falsify properties up to a given depth *k*
  - they can prove correctness only if an upper bound of *k* is known (**unwinding assertion**)

Solution: handles such (unbounded) problems using proof by induction

- k-induction has been successfully combined with continuously-refined invariants
  - to prove that (restricted) C programs do not contain data races (Donaldson et al., 2010)
  - in hardware verification (Eén and Sörensson, 2003)

Difficulties in proving the correctness of programs with loops in BMC

- This paper contributes:
    - a new algorithm to prove correctness of C programs
    - combining k-induction with invariants
    - in a completely automatic way



ESBMC

PIPS

k-induction

invariants

Properties Verification

True

¬φk

False

Agenda

1. Introduction
2. Background
3. Proposed Method
4. Experimental Evaluation
5. Related Work
6. Conclusions and Future Work

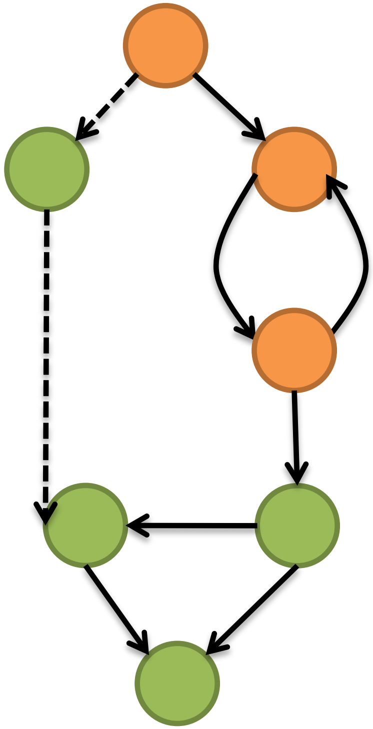
Efficient SMT-Based Bounded Model Checking - ESBMC

ESBMC is a bounded model checker for embedded ANSI-C software based on SMT (Satisfiability Modulo Theories) solvers, which allows:

- ✓ Out-of-bounds array indexing;
- ✓ Division by zero;
- ✓ Pointers safety
- ✓ Dynamic memory allocation;
- ✓ Multi-Threaded software
- ✓ Data races;
- ✓ Deadlocks;
- ✓ Underflow e Overflow;

Program Invariants

Invariants are properties of program variables and relationships between these variables in a specific line of code which called program point.



`n = size(b)`

```
i := 0;
s := 0;
while i ≠ n{
   s := s+b[i];
   i := i+1;
}
```

`s = sum(b[0..i-1])`

`s = sum(b)`

Agenda

1. Introduction
2. Background
3. Proposed Method
4. Experimental Evaluation
5. Related Work
6. Conclusions and Future Work

Induction-based Verification of C programs using Invariants

k-induction checks for each step k

- **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

Induction-based Verification of C programs using Invariants

```
k = 1; force_basecase = FALSE; last_result = UNKNOWN;
while k <= max_iterations do
if force_basecase then
    k = k + 5;
    if base_case(P, ϕ, k) then
        show counterexample s[0..k];
        return FALSE;
    else
      if force_basecase then return last_result
      k=k+1
      if forward_condition(P, ϕ, k) then
         force_basecase = TRUE;  last_result = TRUE;
      else
        if inductive_step(P, ϕ, k) then
          force_basecase = TRUE;  last_result = TRUE;

return UNKNOWN
```

Induction-based Verification of C programs using Invariants

input: program P and safety property ϕ

output: true, false, or unknown

```
    k = k + 5;
    if base_case(P, $\phi$, k) then
        show counterexample s[0..k];
        return FALSE;
    else
      if force_basecase then return last_result
      k=k+1
      if forward_condition(P, $\phi$, k) then
         force_basecase = TRUE;  last_result = TRUE;
      else
        if inductive_step(P, $\phi$, k) then
          force_basecase = TRUE;  last_result = TRUE;

return UNKNOWN
```

Induction-based Verification of C programs using Invariants

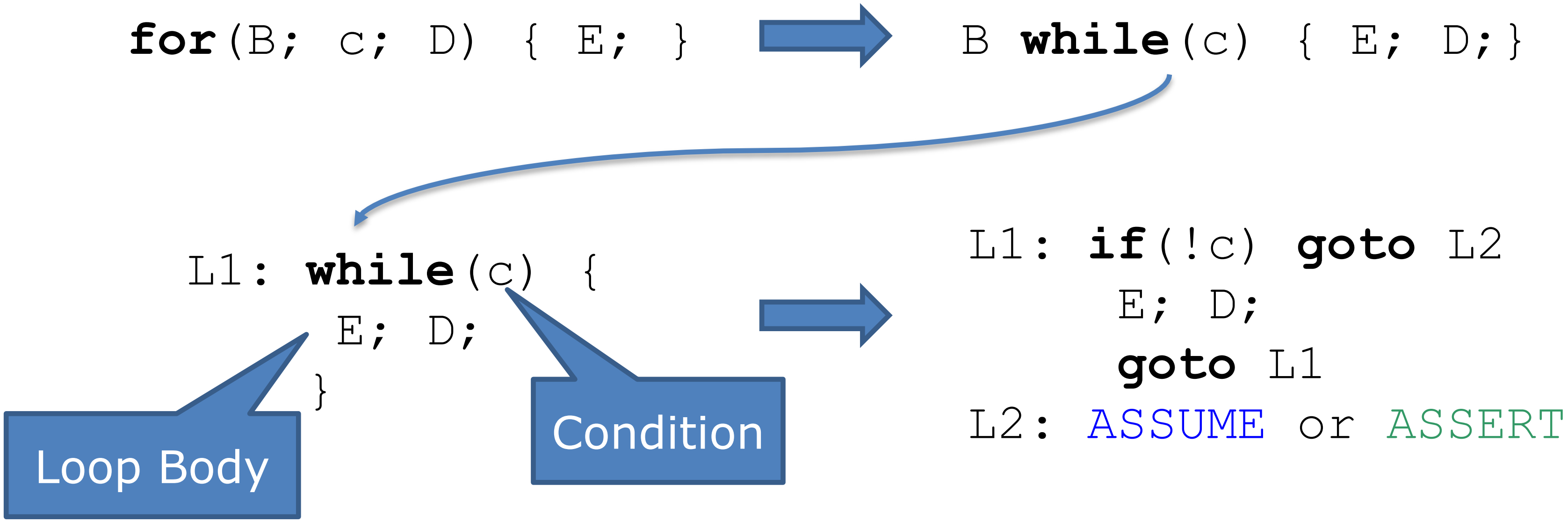
```
k = 1; force_basecase = FALSE; last_result = UNKNOWN;
while k <= max_iterations
if force_basecase then
    k = k + 5;
    if base_case(P, $\phi$, k) then
        show counterexample s[0..k];
        return FALSE;
    else
      if force_basecase then return last_result
      k=k+1
      if forward_condition(P, $\phi$, k) then
        force_basecase = TRUE;  last_result = TRUE;
      else
        if inductive_step(P, $\phi$, k) then
         force_basecase = TRUE;  last_result = TRUE;

return UNKNOWN
```

Rechecking of the result.
It is needed due to the
inclusion of invariants.

Avoid incorrect
exploration

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) {
      E; D;
    }
```

Loop Body

Condition

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

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

In the inductive step, loops are converted into:

the code to remove redundant states

`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

Program Transformation

```
k = 1; force_basecase = FALSE; last_result = UNKNOWN;
while k <= max_iterations do
if force_basecase then
    k = k + 5;
    if base_case(P, $\phi$, k) then
        show countere      e s[0..k];



                                   ret

      if forward_condition(P, $\phi$, k) then
         force_basecase = TRUE;  last_result = TRUE;
      else
        if inductive_step(P, $\phi$, k) then
          force_basecase = TRUE;  last_result = TRUE;

return UNKNOWN
```

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

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

inserts **unwinding assumption** after each loop

Program Transformation

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

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

```
                          FALSE; last_result = UNKNOWN;
                          s do


    if base_case(P, $\phi$, k) then
        show counterexample s[0..k];
        return FALSE;
    else
      if force_basecase then retu
      k=k+1
      if forward_condition(P, $\phi$,
         force_basecase = TRUE;  last_result = TRUE;
      else
        if inductive_step(P, $\phi$, k) then
          force_basecase = TRUE;  last_result = TRUE;

return UNKNOWN
```

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

inserts **unwinding assertion** after each loop

Program Transformation

I : initial condition
T : transition relation of P’
σ : termination condition
ϕ : safey property

```
FALSE; last_result = UNKNOWN;
s do


    if base_case(P, $\phi$, k) then
        show counterexample s[0..k];
        return FALSE;
    else
      if force_basecase then return last_result
      k=k+1
      if forward_condition(P, $\phi$, k) then
         force_basecase = TRUE;  last_result = TRUE;
      else
        if inductive_step(P, $\phi$, k) then
          force_basecase = TRUE;  last_result = TRUE;

return UNKNOWN
```

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

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

γ: transition relation of P’

$\boxed{\gamma \wedge \sigma \Rightarrow \phi}$

Program Transformation

```
k = 1; force_basecase = FALSE; last_result = UNKNOWN;
while k <= max_iterations do
if force_basecase then
    k = k + 5;
    if base_case(P, $\phi$, k) then
        show counterexample s[0..k];
        return FALSE;
    else
      if force_basecase then return last_result
      k=k+1
      if forward_condition(P, $\phi$, k) then
         force_basecase = TRUE;  last_result = TRUE;
      else
        if inductive_step(P, $\phi$, k) then
          force_basecase =                          = TRUE;

return UNKNOWN
```

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

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

$\boxed{\gamma \wedge \sigma \Rightarrow \phi}$

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

Invariant Generation

To infer program invariants, we adopted the PIPS tool

- It is an interprocedural source-to-source compiler framework for C and Fortran programs
- It relies on a polyhedral abstraction of program behavior

```
...
// P(i,k,n0,n1,n2){i==0, 0<=k, n0<=k, n0+n1<=k, n0+n1+n2<=k,
                   n0+n2<=k, n1<=k, n1+n2<=k, n2<=k}

while (i<n2) {
   //  P(i,k,n0,n1,n2) {0<=i, n0+n1+n2<=i+k, n0+n2<=i+k,
   //                   n1+n2<=i+k, n2<=i+k, i+1<=n2}

   i++;
...
```

polyhedral invariants are propagated along with instructions

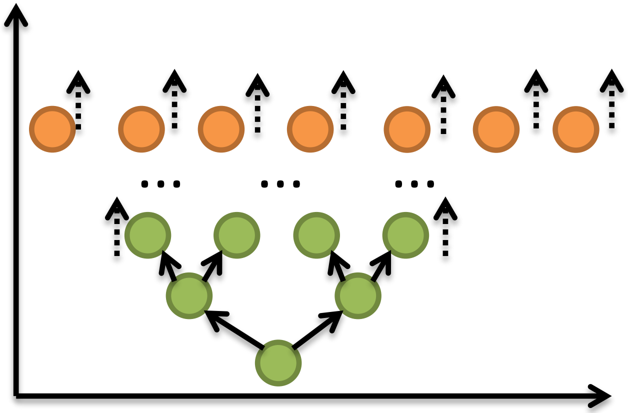
Invariant Generation

PIPS Invariant Translation

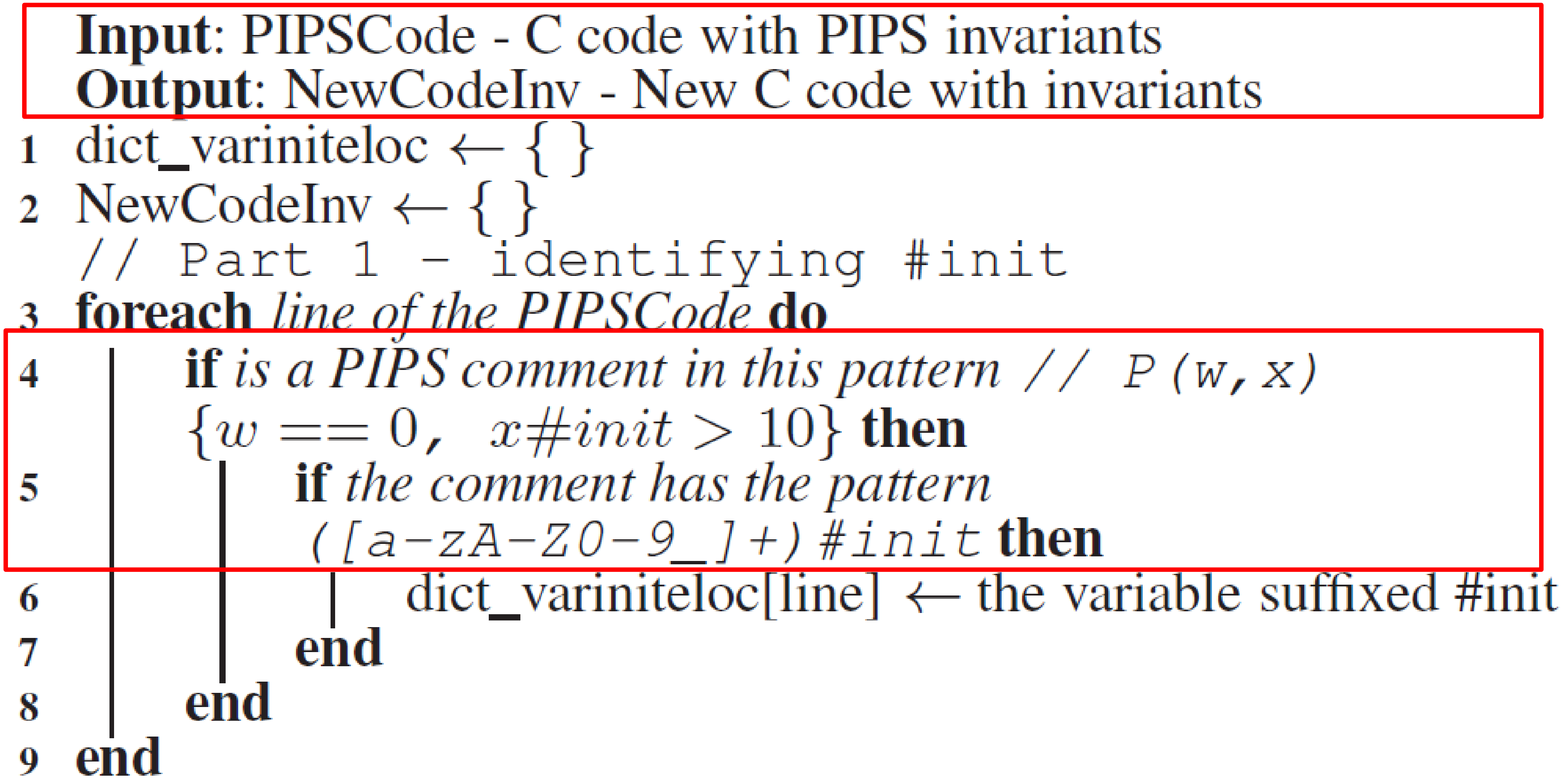
- Mathematical expressions,e.g, $2j < 5t$
- Invariants with $\#init$ suffix that is used to distinguish the old value to the new value

The invariants are translated and instrumented into the program as assume statements

- `assume(expr)` to limit possible values of the variables that are related to the invariants

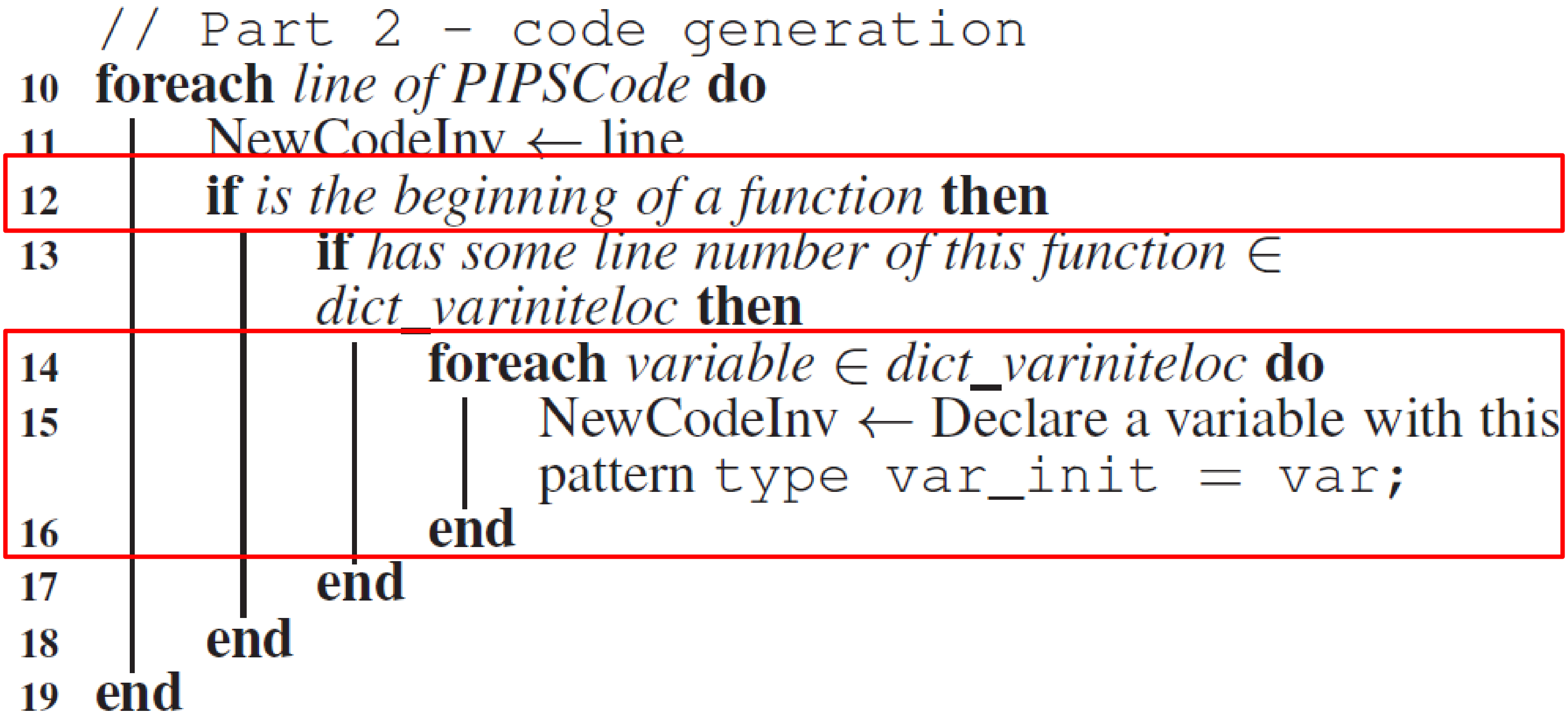


Translation algorithm of invariants



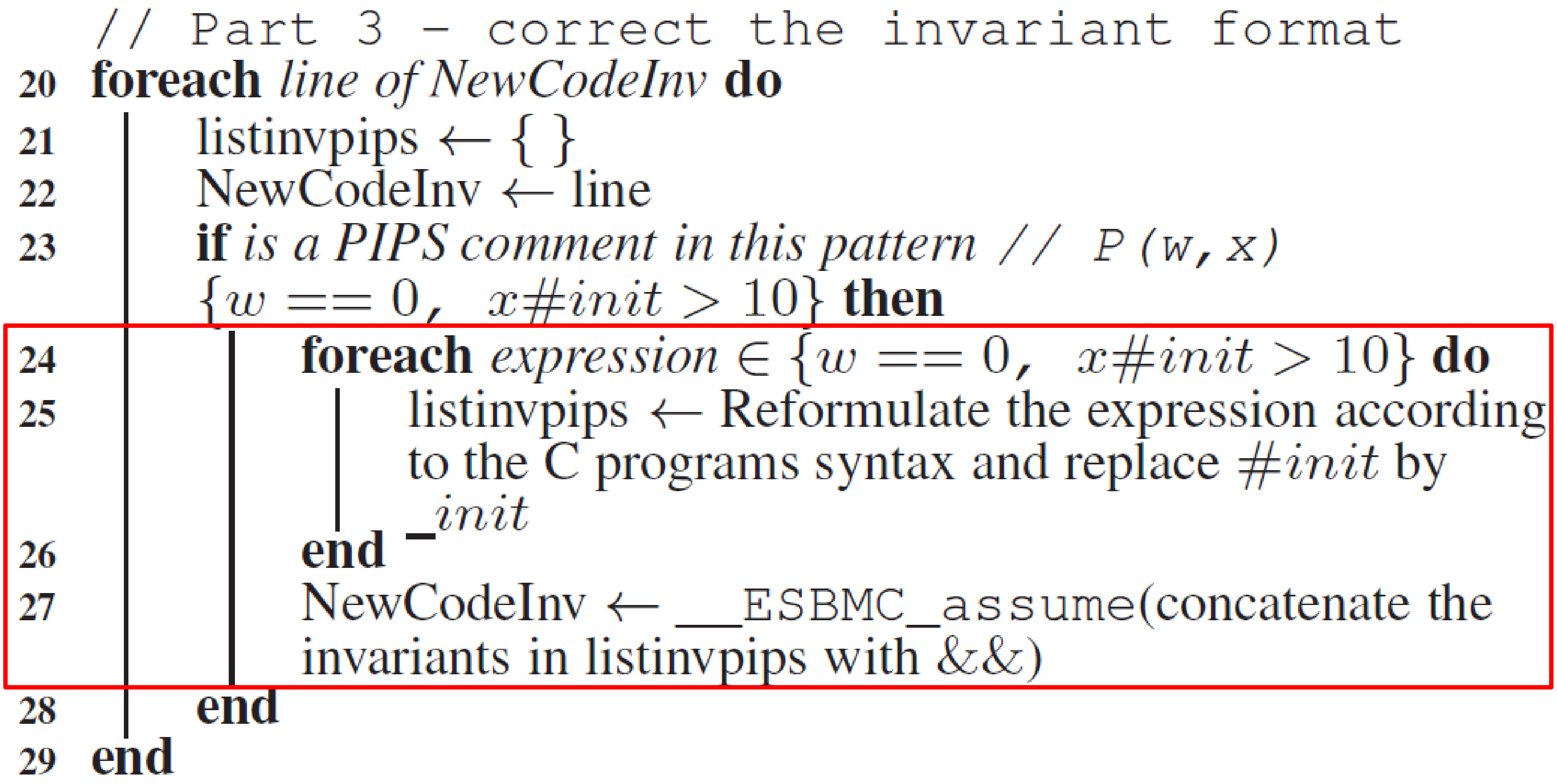
```
Input: PIPSCode - C code with PIPS invariants
Output: NewCodeInv - New C code with invariants
dict_variniteloc ← { }
NewCodeInv ← { }
// Part 1 - identifying #init
foreach line of the PIPSCode do
    if is a PIPS comment in this pattern // P(w,x) $\{w == 0,\ x\#init > 10\}$ then
        if the comment has the pattern ([a-zA-Z0-9_]+)#init then
            dict_variniteloc[line] ← the variable suffixed #init
        end
    end
end
```

Translation algorithm of invariants



```
// Part 2 - code generation
foreach line of PIPSCode do
    NewCodeInv ← line
    if is the beginning of a function then
        if has some line number of this function ∈ dict_variniteloc then
            foreach variable ∈ dict_variniteloc do
                NewCodeInv ← Declare a variable with this pattern type var_init = var;
            end
        end
    end
end
```

Translation algorithm of invariants



```
// Part 3 - correct the invariant format
foreach line of NewCodeInv do
    listinvpips ← { }
    NewCodeInv ← line
    if is a PIPS comment in this pattern // P(w,x) {$w == 0,\ x\#init > 10$} then
        foreach expression ∈ {$w == 0,\ x\#init > 10$} do
            listinvpips ← Reformulate the expression according to the C programs syntax and replace $\#init$ by $\_init$
        end
        NewCodeInv ← __ESBMC_assume(concatenate the invariants in listinvpips with &&)
    end
end
```

Agenda

1. Introduction
2. Background
3. Proposed Method
4. Experimental Evaluation
5. Related Work
6. Conclusions and Future Work

Planning and Designing the Experiments

Goal: Analyzing the ability of DepthK to **verify** a wide variety of safety properties in C programs.

✓ The experiments are conducted on an Intel Xeon CPU E5 – 2670 CPU, 2.60GHz, 115GB RAM with Linux OS

✓ The time limit to the verification is 15 min

✓ Memory consumption limit to 15 GB

DepthK tool is available at https://github.com/hbgit/depthk

Planning and Designing the Experiments

- ✓ **142 ANSI-C** programs of the **SV-COMP 2015** (Beyer, 2015);
- ✓ **34 ANSI-C** programs used in **embedded systems:**
  - Powerstone (Scott et al., 1998)
  - SNU real-time (SNU, 2012)
  - WCET (MRTC, 2012)
- ✓ Comparison with the tools:
  - CPAChecker SVN v15596 (Beyer e Keremoglu, 2011)
  - CBMC v5.0 with *k*-induction (Clarke et al., 2004a)
  - ESBMC v1.25.2 with *k*-induction (Cordeiro et al., 2012)

Planning and Designing the Experiments

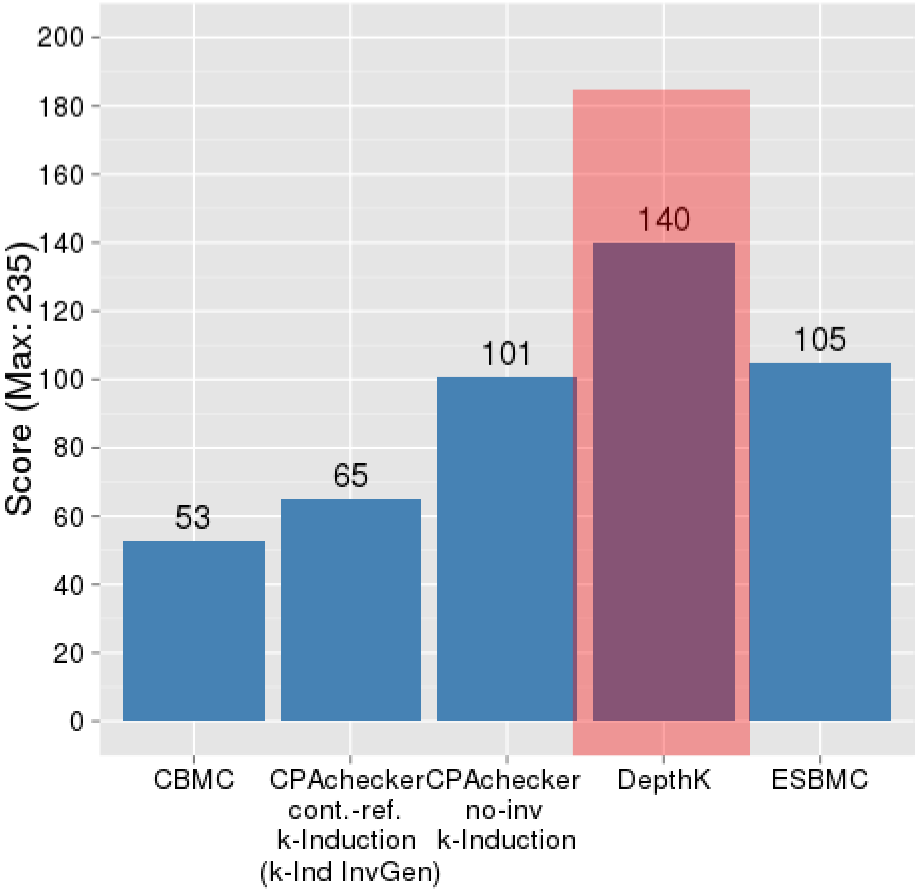
In the experiments, we collect the data:

- ✓ Correct Results
- ✓ False Incorrect
- ✓ True Incorrect
- ✓ Unknown
- ✓ Time

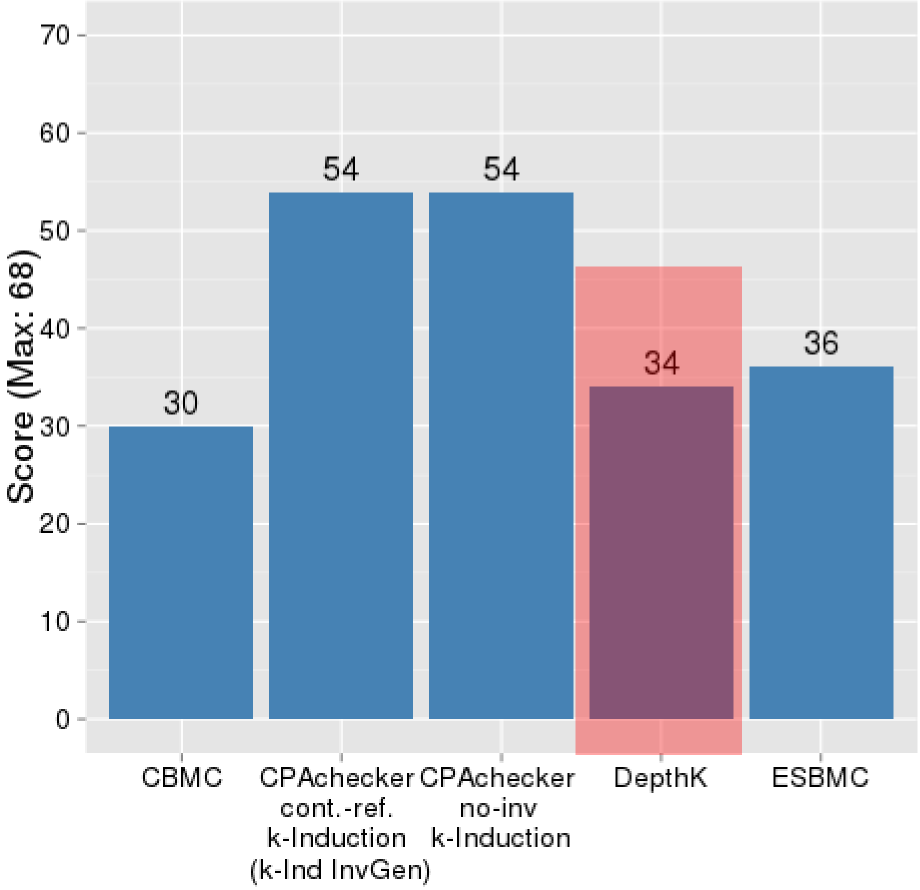
To evaluate we adopted the same scoring scheme that is used in SVCOMP 2015, e.g., **12 scores are subtracted** for every wrong safety proof (True Incorrect).

Experimental Results

SV-COMP: Loops benchmarks

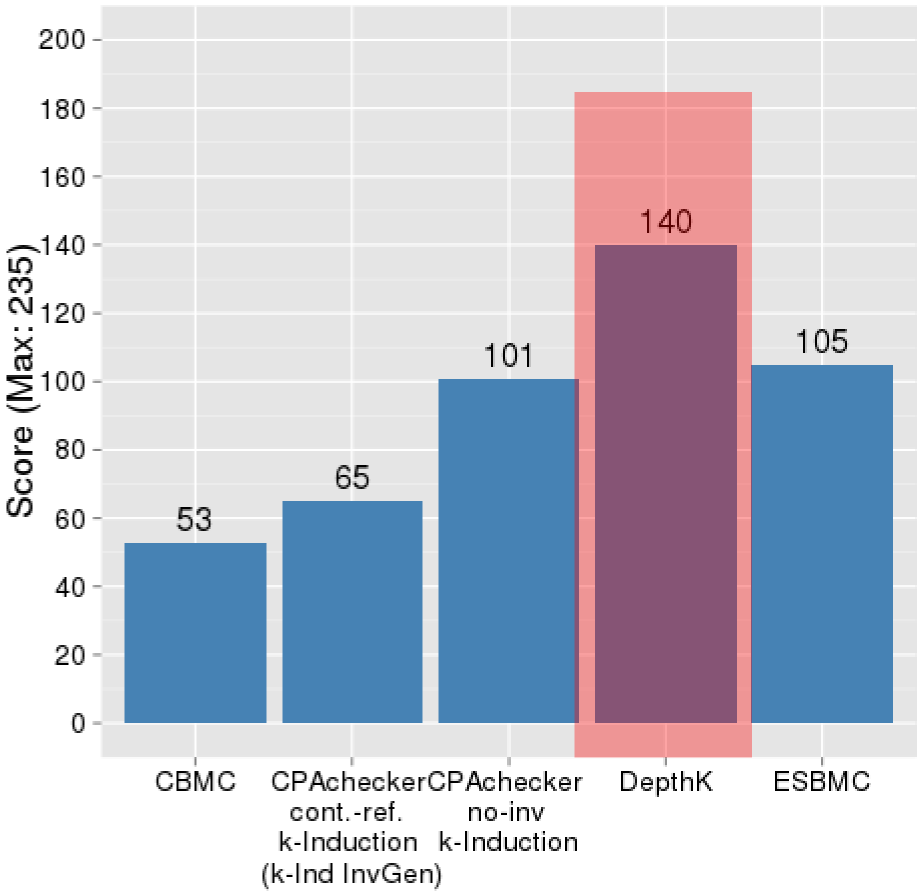


Embedded systems programs

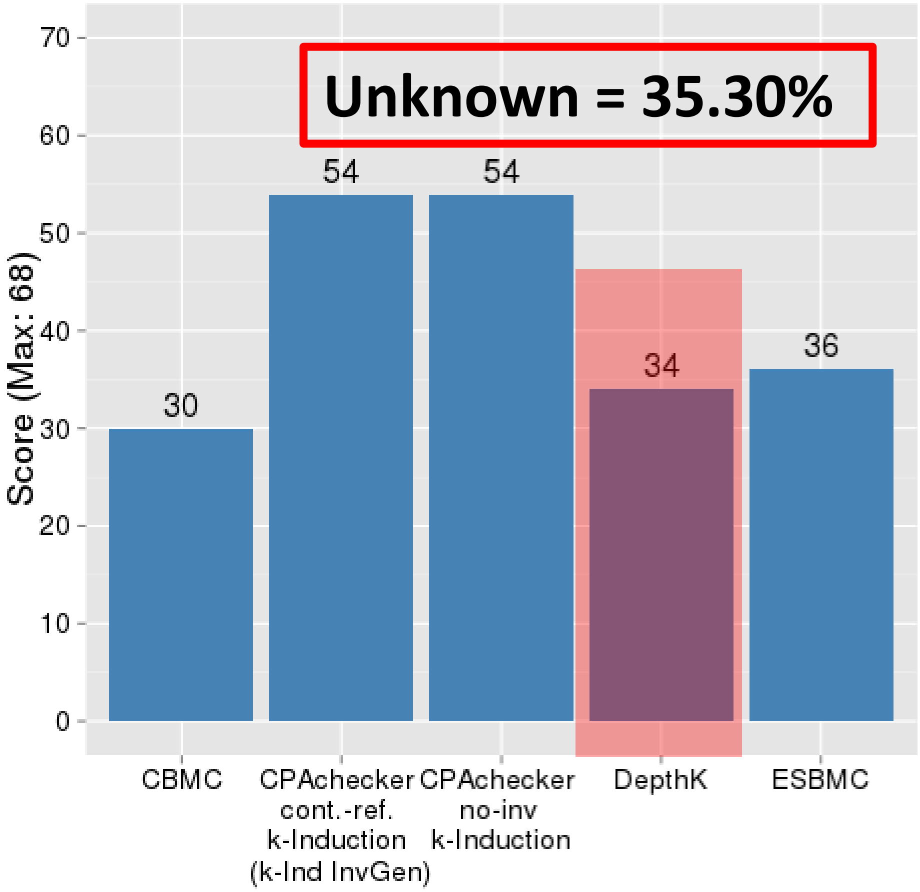


Experimental Results

SV-COMP: Loops benchmarks



Embedded systems programs

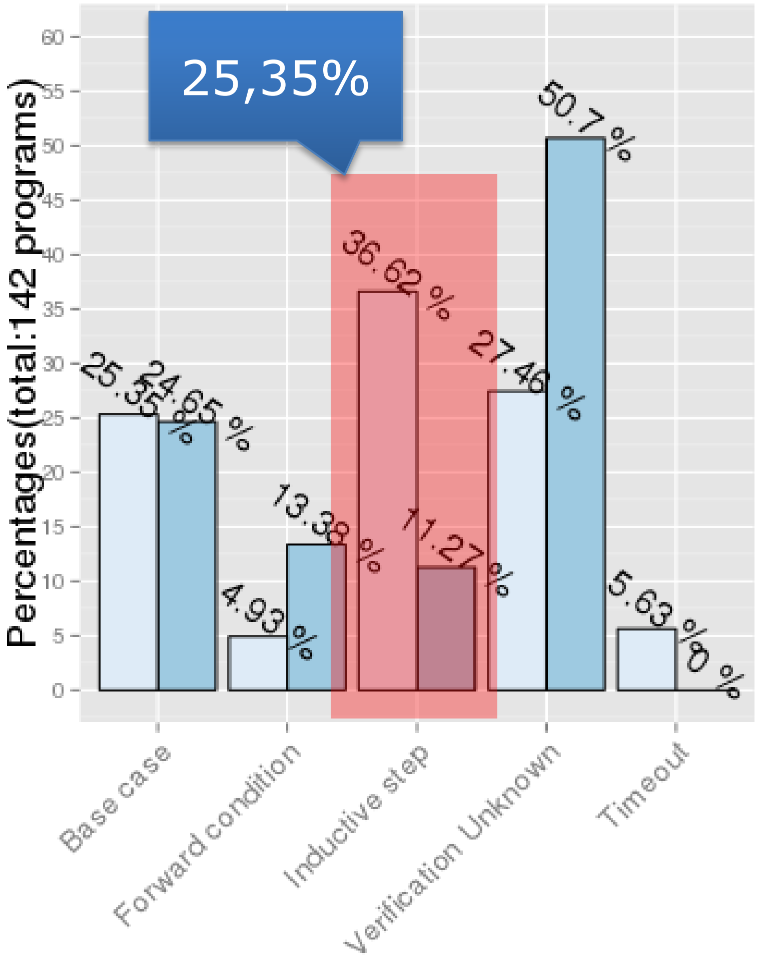


Unknown = 35.30%

Experimental Results

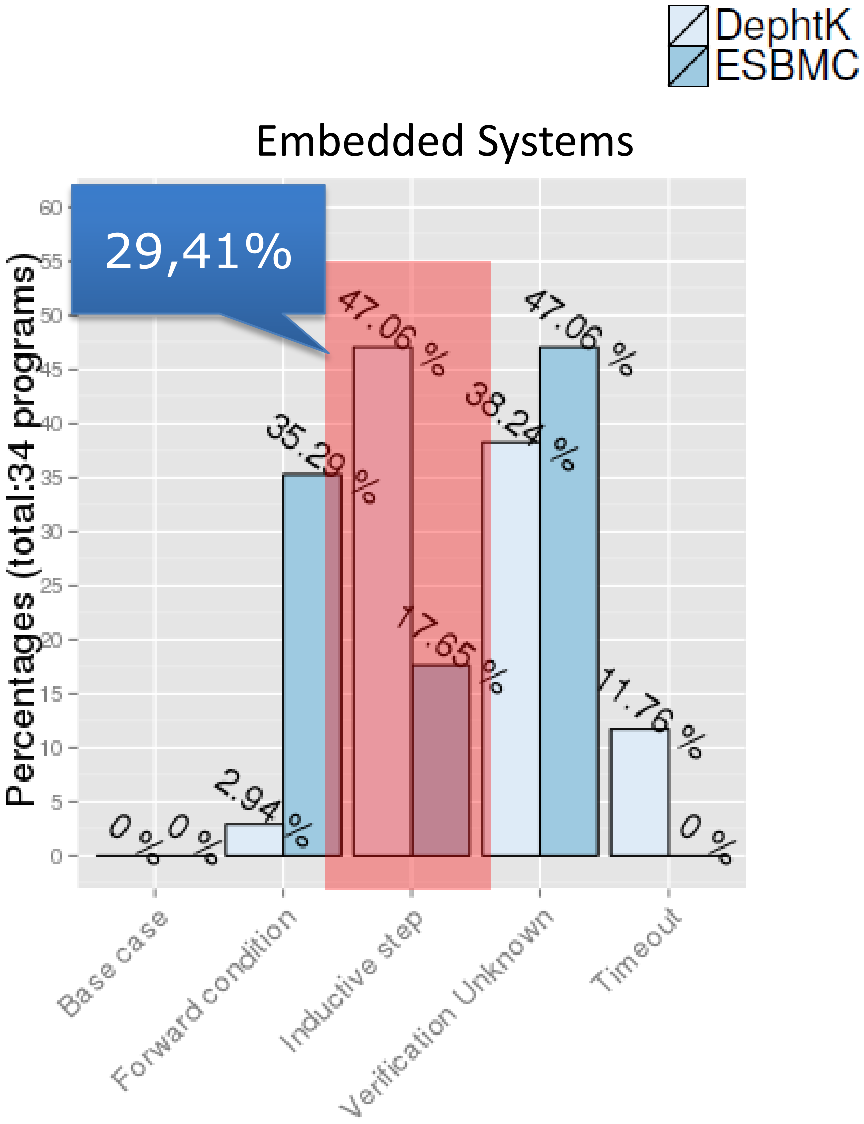
- ✓ *K*-induction steps

SV-COMP



25,35%

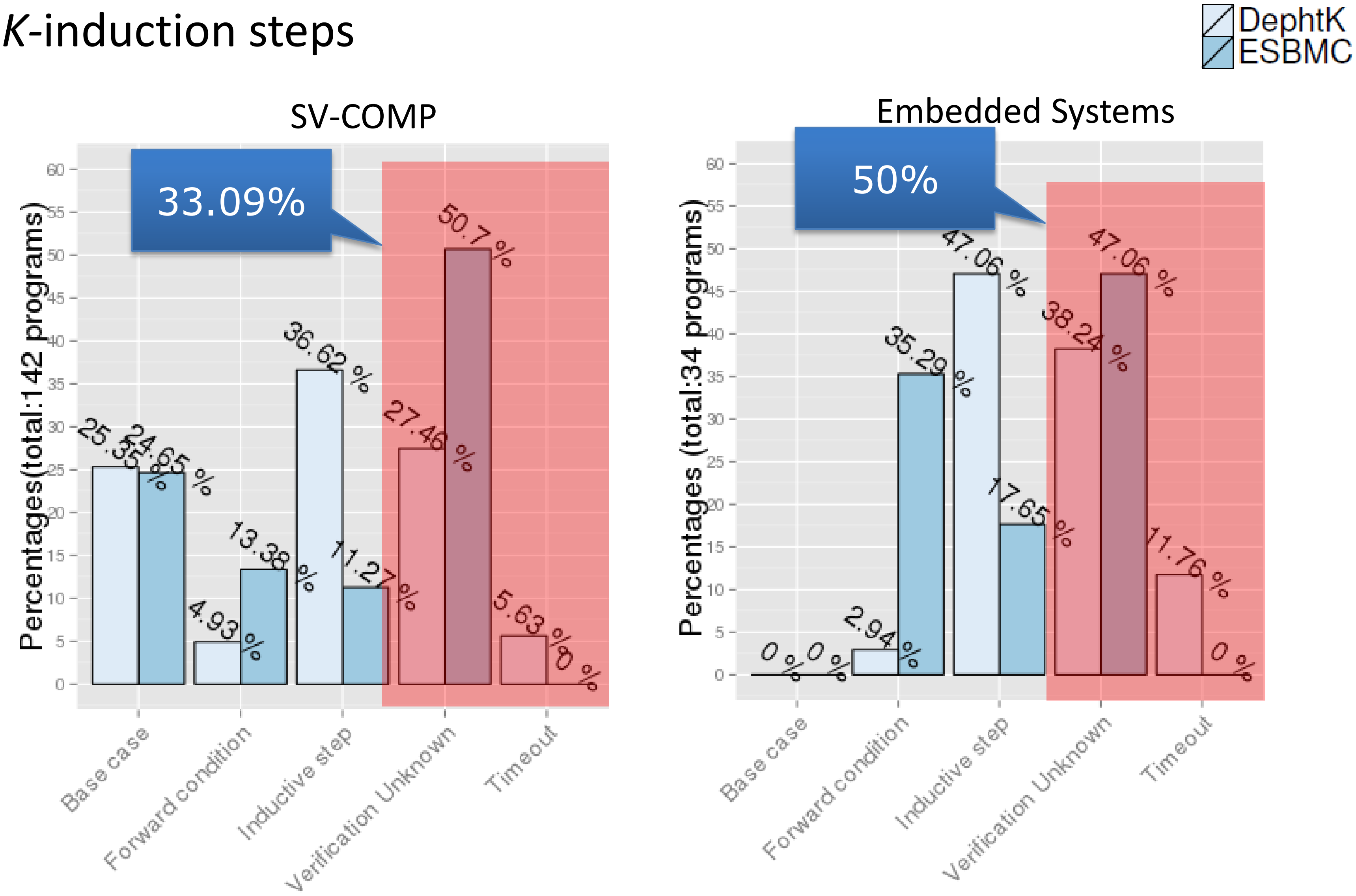
Embedded Systems



29,41%

Experimental Results

- ✓ *K*-induction steps



SV-COMP

33.09%

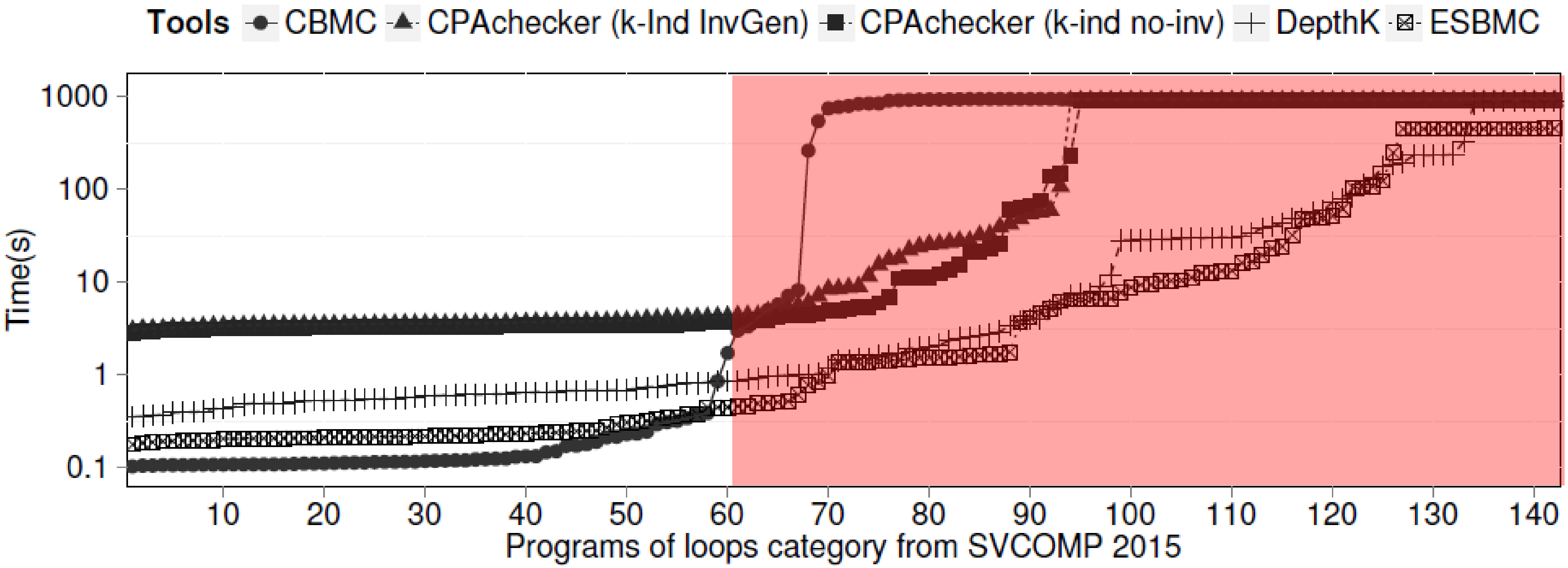
Embedded Systems

50%

Experimental Results

The verification time:

✓ DepthK is usually **faster** than the other tools, **except for ESBMC**

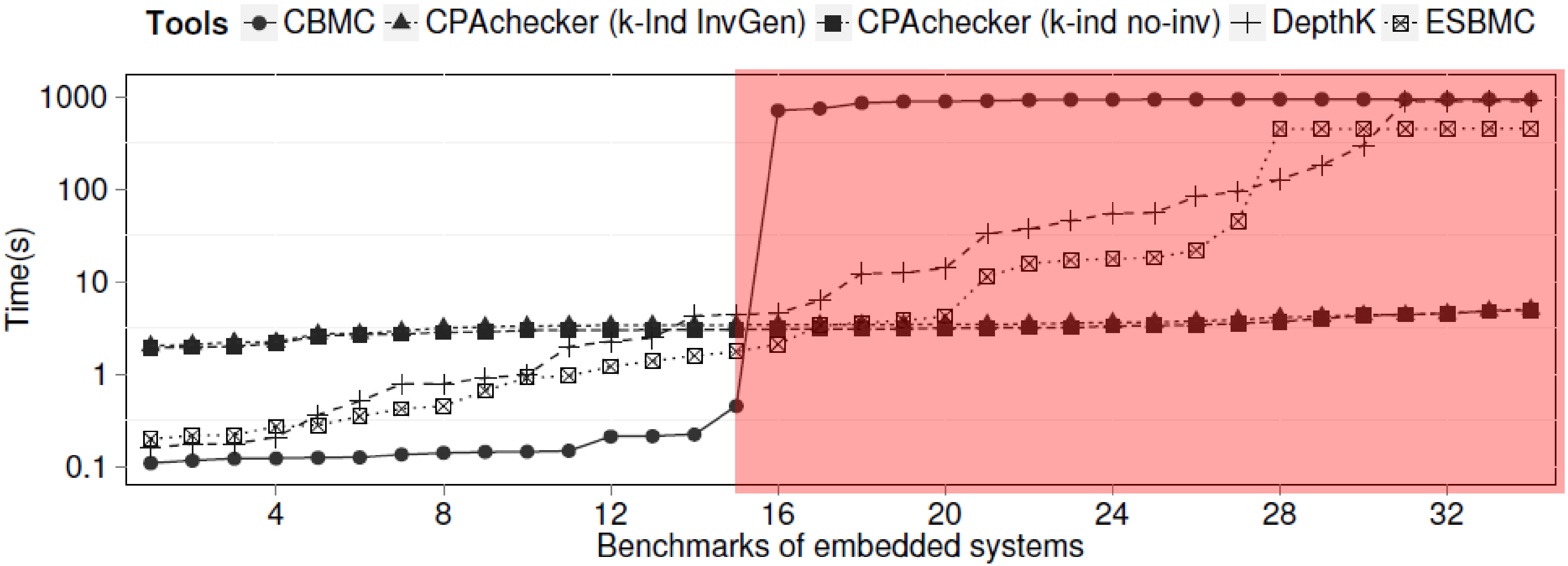


Experimental Results

The verification time:

✓ DepthK is **only faster** than CBMC

35.30% de Unknown

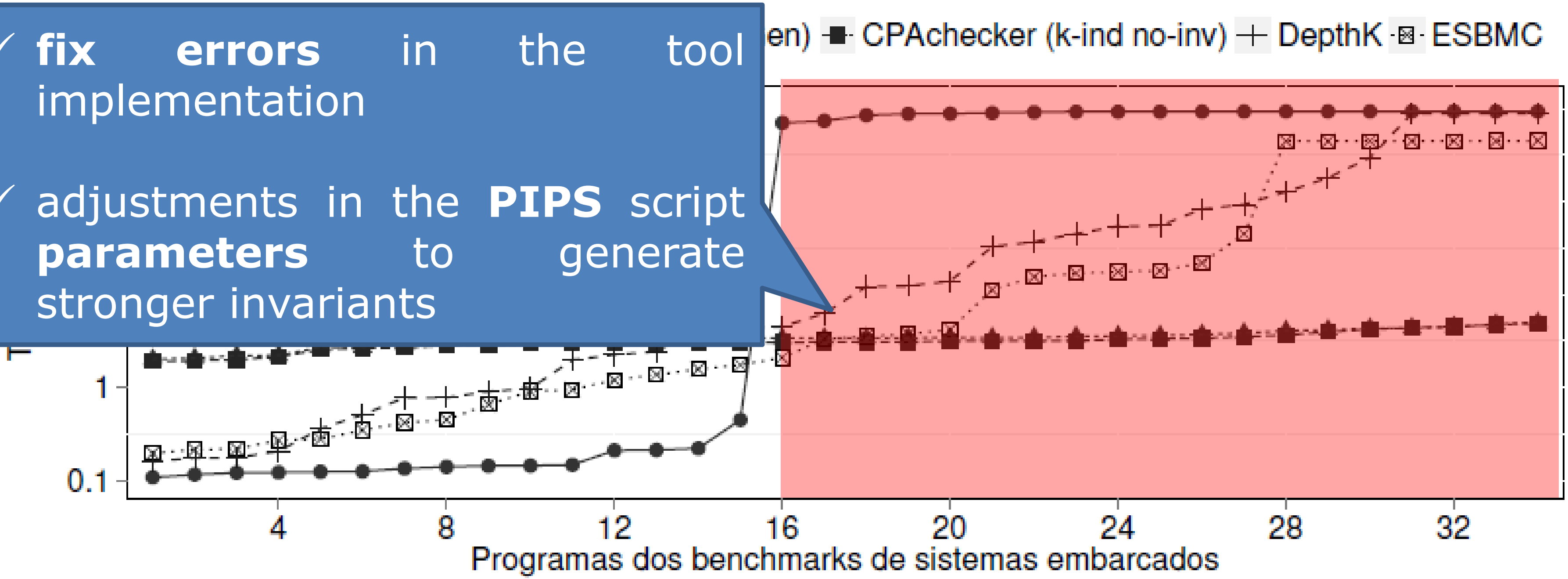


Experimental Results

The verification time:

✓ DepthK is **only faster** than CBMC

35.30% de Unknown



- ✓ **fix errors** in the tool implementation
- ✓ adjustments in the **PIPS** script **parameters** to generate stronger invariants

Agenda

1. Introduction
2. Background
3. Proposed Method
4. Experimental Evaluation
5. **Related Work**
6. Conclusions and Future Work

Related Works

VS.

(Groβe, 2012)

- ✓ Explored proofs by mathematical induction of hardware and software systems
- ✓ Required changes in the code to introduce loop invariants

(Hagen et al., 2018 and Donaldson et al., 2011)

- ✓ The approach is similar
- ✓ Our method is completely automatic and does not require the user to provide loops invariants
- ✓ Not only as bug-finding tools
- ✓ DepthK aims prove correctness of C programs

Related Works

VS.

(Donaldson et al., 2010) - Scratch tool

- ✓ It is restricted to verify a specific class of problems for a particular type of hardware
- ✓ It requires annotations in the code to introduce loops invariants

(Beyer et al., 2015) - CPAChecker

- ✓ Invariant generation from predefined templates (**interval**) and constantly feeds the inductive step process
- ✓ DepthK adopts PIPS (polyhedral)

Agenda

1. Introduction
2. Background
3. Proposed Method
4. Experimental Evaluation
5. Related Work
6. Conclusions and Future Work

Conclusions

The experimental results are promising:

- ✓ DepthK determined **11.27% more accurate results** than that obtained by **CPAChecker** in the SVCOMP 2015 loops subcategory
- ✓ DepthK determined **3.45% more correct results** to analyze all 176 C programs
- ✓ DepthK determined **17% more accurate results** than the *k*-induction algorithm without invariant

Conclusions

Improvements in the DepthK tool - In embedded systems benchmarks:

- ✓ DepthK only obtained better results than CBMC tool

For future works:

- ✓ We will improve the robustness of DepthK and tune the PIPS parameters to produce stronger invariants

Overview:

- ✓ In comparison to other state of the art tools, showed promising results indicating its effectiveness.

Questions ?

Thank you for your attention!

herberthb12@gmail.com