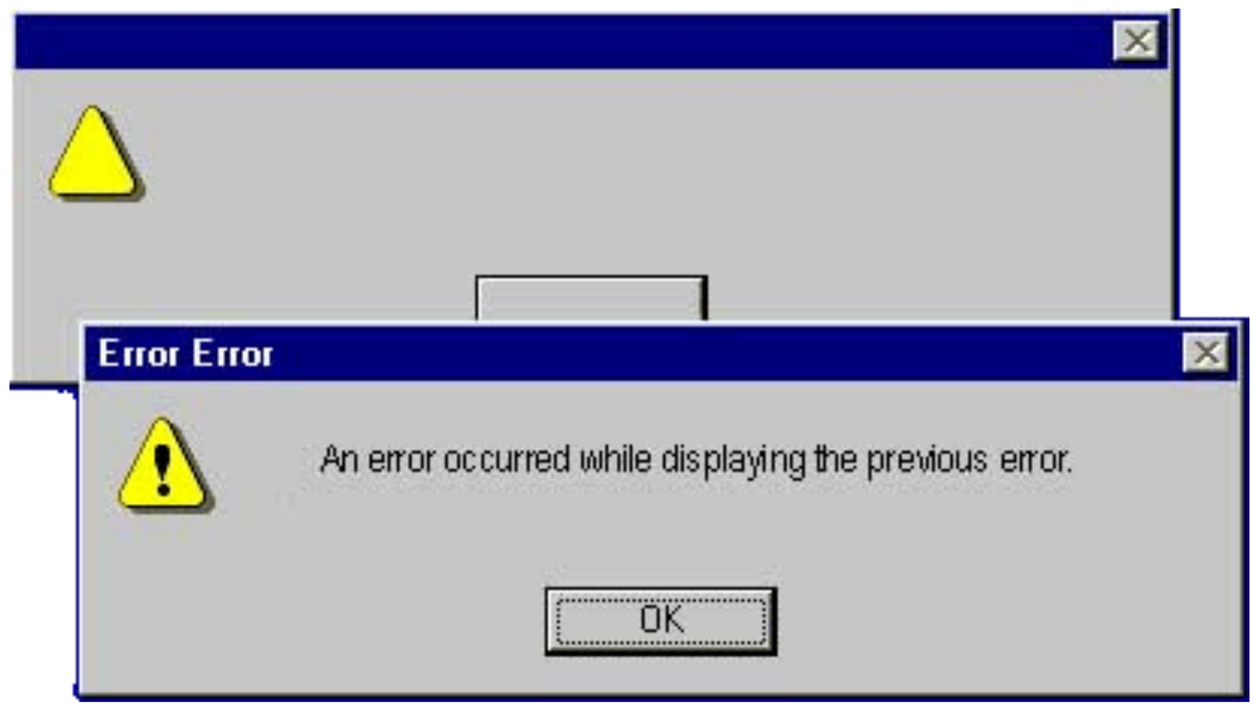


Towards Counterexample-guided *k*-Induction for Fast Bug Detection

Mikhail R. Gadelha
Felipe R. Monteiro
Lucas C. Cordeiro
Denis A. Nicole

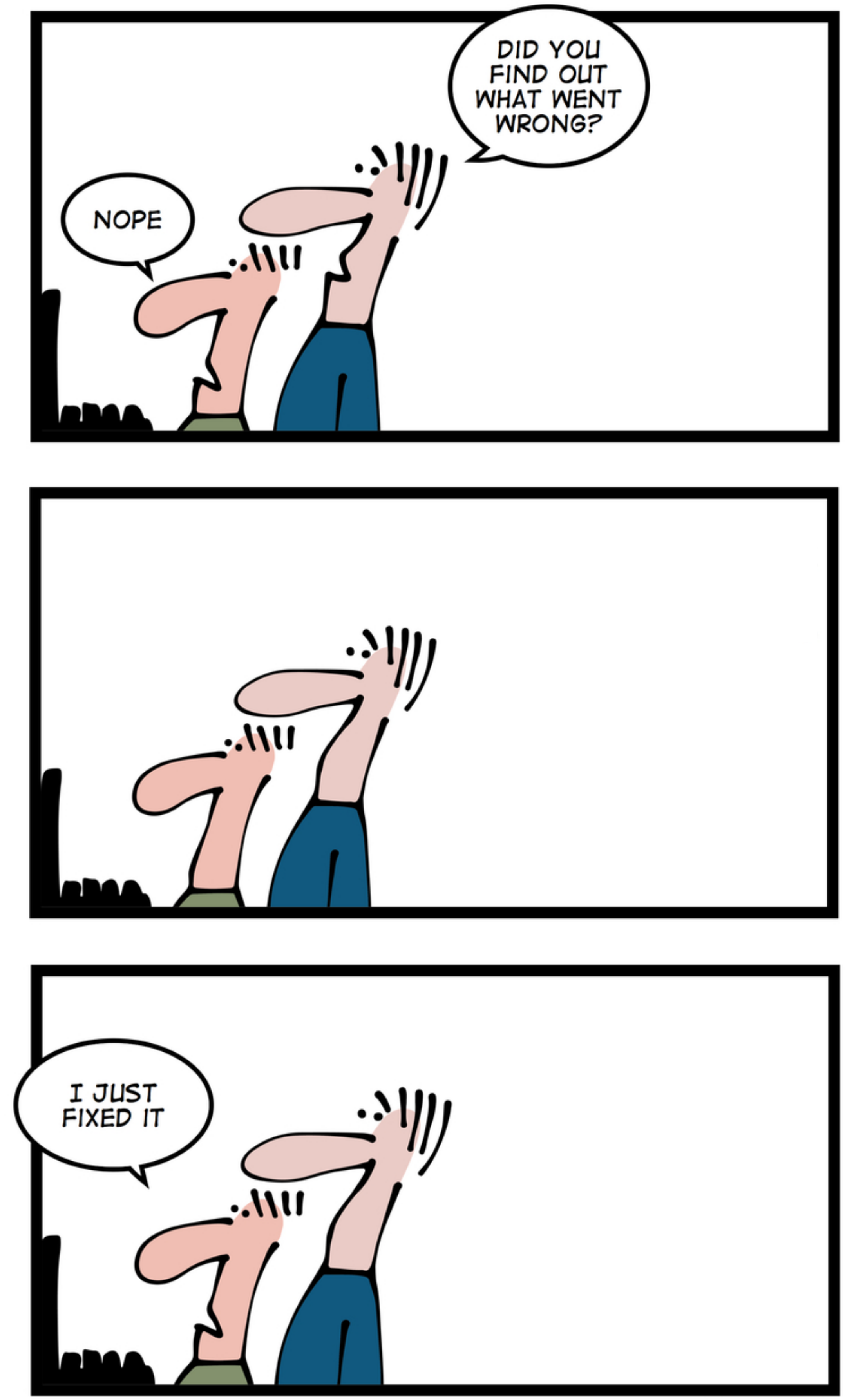
26th ACM Joint European Software Engineering Conference and Symposium
on the Foundations of Software Engineering

Towards Counterexample-guided *k*-Induction for Fast Bug Detection

Mikhail R. Gadelha, **Felipe R. Monteiro**, Lucas C. Cordeiro, and Denis A. Nicole

Motivation

Why should we invest in software reliability?



Why should we invest in software reliability?

- The ubiquity of embedded systems drives a need to **test and validate a system** before releasing it to the market, in order to protect against **system failures.**

"Formal automated reasoning is one of the investments that AWS is making in order to facilitate continued simultaneous growth in both functionality and security."

- Byron Cook, FLoC, 2018.

Why should we invest in software reliability?

- Embedded software must be as robust and bug-free as possible, given that **even subtle system bugs can have drastic consequences:**

    - In April 2014, the **Heartbleed** was publicly disclosed, a security bug in the OpenSSL cryptography library, which is a widely used implementation of the Transport Layer Security (TLS) protocol.

    - “*When it is exploited it leads to the leak of memory contents from the server to the client and from the client to the server.*”

      – Synopsys Inc., 2014.

Why should we invest in software reliability?

- Embedded software must be as robust and bug-free as possible, given that **even subtle system bugs can have drastic consequences:**

  - In April 2014, the **Heartbleed** was publicly disclosed, a security bug in the OpenSSL cryptography library, which is a widely used implementation of the Transport Layer Security (TLS) protocol.

  - *"When it is exploited it leads to the leak of memory contents from the server to the client and from the client to the server."*

    – Synopsys Inc., 2014.

  - In September 2018, attackers exploited three **Facebook** vulnerabilities and stole access tokens from as many as 50 million users, in order to take over their accounts.

Our main goal is to…

Propose a faster approach to detect bugs and prove correctness of a program

We demonstrate in this paper how to…

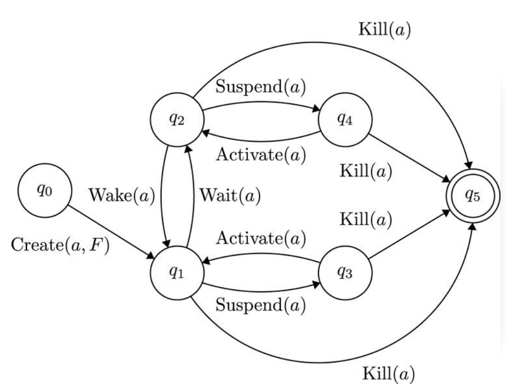
Improve the *k*-induction algorithm to work as a meet-in-the-middle bidirectional search by using the information from the counterexample

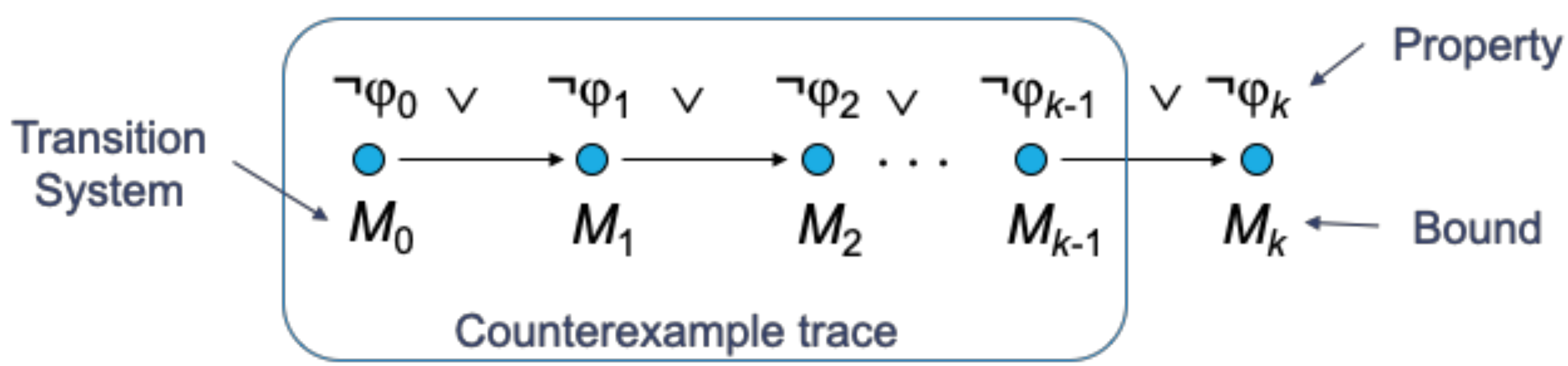
Background

The *k*-Induction Algorithm

Bounded Model Checking

- Basic Idea: given a transition system *M*, check negation of a given property ϕ up to given depth *k:*





- Translated into a VC ψ such that: **ψ is satisfiable iff ϕ has counterexample of max. depth *k*.**
- BMC tools are aimed at finding bugs; they cannot prove correctness, unless the bound *k* safely reaches all program states.

```c
int main() {
    uint32_t n;
    uint64_t sn = 0;
    for (uint64_t i = 1; i <= n; i++) {
        sn = sn + 2;
        assert(sn == i * 2);
    }
    assert(sn == n*2 || sn == 0);
}
```

```c
1  int main() {
2      uint32_t n;
3      uint64_t sn = 0;
4      for (uint64_t i = 1; i <= n; i++) {
5          sn = sn + 2;
6          assert(sn == i * 2);
7      }
8      assert(sn == n*2 || sn == 0);
9  }
```

```c
1  uint64_t i = 1;
2  /* n copies of this if statement */
3  if(i <= n) {
4    sn = sn + 2;
5    assert(sn == i * 2);
6    i++;
7  }
8  /* unwinding assertion */
9  assert(!(i<=n));
```

BMC tools such as CBMC, ESBMC or LLBMC typically reproduce the loop *k* times (lines 4 – 7) and are unable to verify that program unless the loop is fully unrolled, *i.e.*, the unwinding assertion fails if *k < (2^32 – 1)*

The *k*-induction Algorithm

I. Base case: usual BMC algorithm, tries to find a property violation.

- Explores all states up to a bound *k*. Cannot prove correctness.

II. Forward Condition: checks the completeness threshold (if all loops were unrolled).

- Cannot find bugs.

III. Inductive Step: over-approximates loops so all states can be checked without unrolling them completely.

- Might return spurious counterexamples.

Approach and Uniqueness

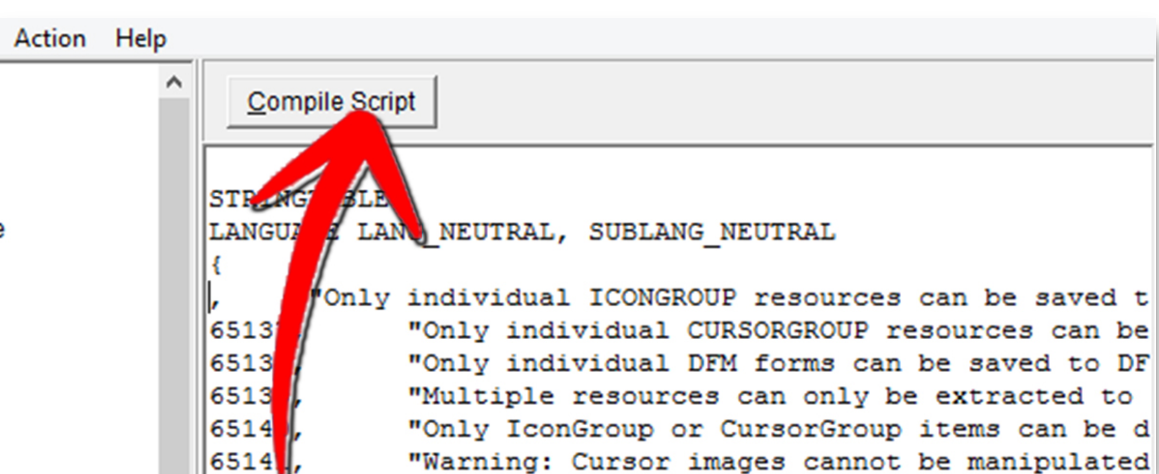
Counterexample-Guided
k-Induction

Counterexample-Guided *k*-Induction

- The biggest limitation of *k*-induction is the fact that it performs three checks for each *k* (*i.e.*, base case, forward condition and inductive step).

- The **inductive step is the most computationally expensive one**; it is an over-approximation, forcing the SMT solver to find a set of assignments in a larger state space than the original program.

- Moreover, the computation is ***wasted*** if a counterexample is found by the inductive step, as it is assumed to be spurious.



Counterexample-Guided *k*-Induction

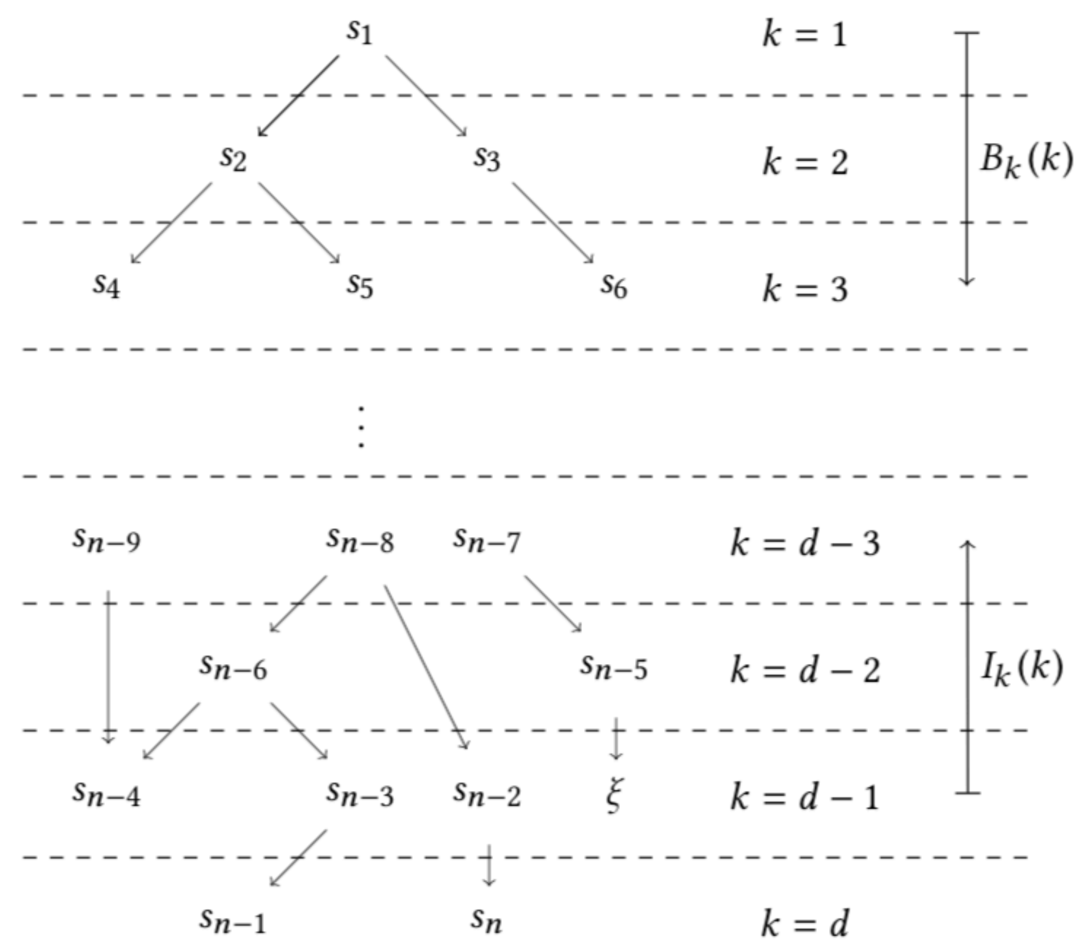
- We propose to use the ***counterexample*** generated by the ***inductive step*** to ***speed up*** the bug finding check (*i.e.*, the base case).

- Our extension converts the *k*-induction algorithm into a **bidirectional search approach** by searching simultaneously:

  i. both forward (*i.e.*, from the initial state);

  ii. backward (*i.e.*, from the error state ξ detected in the inductive step);

  iii. stop if both searches meet in the middle.



Running Example

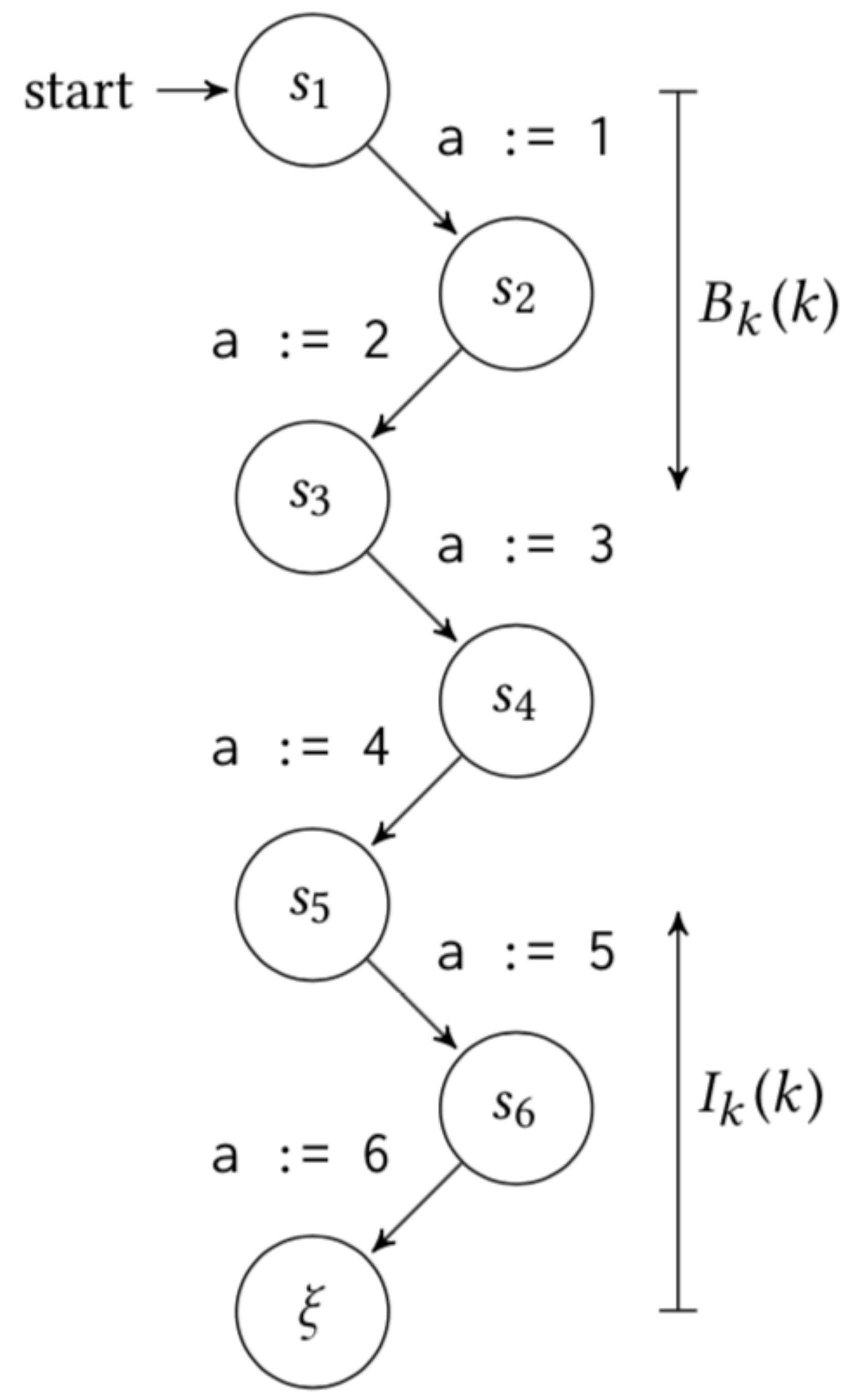
Original Program

```c
unsigned int a = 1;
while(1)
{
  if(a == 6)
    assert(0);
  a++;
}
```

Running Example

Original Program

```
unsigned int a = 1;
while(1)
{
  if(a == 6)
    assert(0);
  a++;
}
```



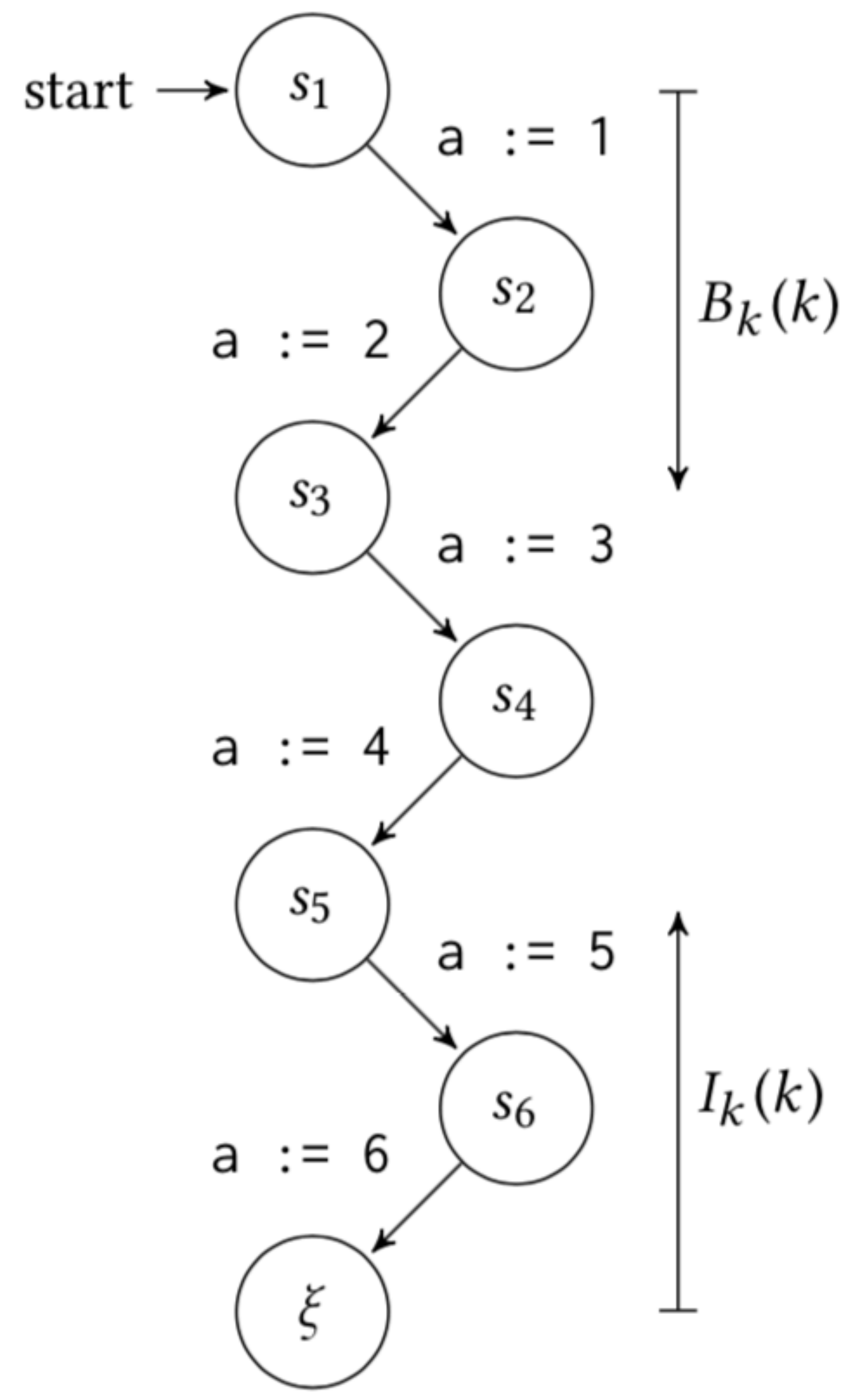
Running Example

Original Program

```c
unsigned int a = 1;
while(1)
{
  if(a == 6)
    assert(0);
  a++;
}
```

Modified Program

```c
unsigned int a = 1;
while(1)
{
  if(a == 6)
    assert(0);
  a++;
  // added assertion
  assert(a != 5);
}
```



Preliminary Results

Evaluate our *k*-induction algorithm extension

Experiments

- In order to evaluate our *k*-induction algorithm extension, we selected a number of benchmarks from the **International Competition on Software Verification 2018.**
- We compare the results from the original k-induction and our extended version.

Experimental setup. All experiments were conducted on a computer with an Intel Core i7-2600 running at 3.40GHz and 24GB of RAM under Fedora 25 64-bit. We used **ESBMC v5.0** and no time or memory limit was set for the verification tasks.

Availability of data & tools. Our experiments are based on a set of publicly available benchmarks. All tools, benchmarks, and the results of our evaluation are available on our web page http://esbmc.org/

Preliminary Results

- Preliminary evaluation over the SV-COMP 2018 benchmarks.

Benchmark	LOC	k-induction T(s)	k-induction M(MB)	k-induction k	Extended k-induction T(s)	Extended k-induction M(MB)	Extended k-induction k
sum04.c	19	**1**	38.7	9	**1**	38.7	5
sum01.c	18	**1**	38.9	11	**1**	38.8	6
sum03.c	25	3	39.1	11	**1**	38.8	6
diamond1.c	24	13	43.6	51	**6**	39.1	26
rangesum.c	64	7	66.2	4	**1**	39.0	2
rangesum05.c	59	11	72.3	6	**1**	65.4	3
rangesum10.c	59	28	78.2	11	**16**	47.5	6
Problem01_label15.c	594	7	87.3	5	**5**	70.3	4
rangesum20.c	59	101	99.9	21	**26**	78.2	12
rangesum40.c	59	847	269.5	41	**90**	113.9	22
const.c	20	2606	796.6	1025	**890**	253.2	513
rangesum60.c	59	80272	1106.9	61	**159**	134.6	32
Average	88	6991	228.1	104	**99**	79.8	53
Total	1059	83897	2737.2	1255	**1197**	957.5	638

Preliminary Results

verification time is not related to the number of steps or the program size

- Preliminary evaluation over the SV-CO

Benchmark	LOC	k-induction			Extended k-induction		
		T(s)	M(MB)	k	T(s)	M(MB)	k
sum04.c	19	1	38.7	9	**1**	38.7	5
sum01.c	18	1	38.9	11	**1**	38.8	6
sum03.c	25	3	39.1	11	**1**	38.8	6
diamond1.c	24	13	43.6	51	**6**	39.1	26
rangesum.c	64	7	66.2	4	**1**	39.0	2
rangesum05.c	59	11	72.3	6	**1**	65.4	3
rangesum10.c	59	28	78.2	11	**16**	47.5	6
Problem01_label15.c	594	7	87.3	5	**5**	70.3	4
rangesum20.c	59	101	99.9	21	**26**	78.2	12
rangesum40.c	59	847	269.5	41	**90**	113.9	22
const.c	20	2606	796.6	1025	**890**	253.2	513
rangesum60.c	59	80272	1106.9	61	**159**	134.6	32
Average	88	6991	228.1	104	**99**	79.8	53
Total	1059	83897	2737.2	1255	**1197**	957.5	638

Preliminary Results

- Preliminary evaluation over

our extension to the *k*-induction algorithm potentially cuts the verification time considerably in cases where the state space explored is large

Benchmark	*LOC*	*k*-induction			Extended *k*-induction		
		T(s)	*M*(MB)	*k*	*T*(s)	*M*(MB)	*k*
sum04.c	19	**1**	38.7	9	**1**	38.7	5
sum01.c	18	**1**	38.9	11	**1**	38.8	6
sum03.c	25	3	39.1	11	**1**	38.8	6
diamond1.c	24	13	43.6	51	**6**	39.1	26
rangesum.c	64	7	66.2	4	**1**	39.0	2
rangesum05.c	59	11	72.3	6	**1**	65.4	3
rangesum10.c	59	28	78.2	11	**16**	47.5	6
Problem01_label15.c	594	7	87.3	5	**5**	70.3	4
rangesum20.c	59	101	99.9	21	**26**	78.2	12
rangesum40.c	59	847	269.5	41	**90**	113.9	22
const.c	20	2606	796.6	1025	**890**	253.2	513
rangesum60.c	59	80272	1106.9	61	**159**	134.6	32
Average	88	6991	228.1	104	**99**	79.8	53
Total	1059	83897	2737.2	1255	**1197**	957.5	638

Preliminary Results

- Preliminary evaluation over

for small cases, our extension does not slow things down or use more memory than the original *k*-induction

Benchmark	*LOC*	*k*-induction *T*(s)	*M*(MB)	*k*	Extended *k*-induction *T*(s)	*M*(MB)	*k*
sum04.c	19	**1**	38.7	9	**1**	38.7	5
sum01.c	18	**1**	38.9	11	**1**	38.8	6
sum03.c	25	3	39.1	11	**1**	38.8	6
diamond1.c	24	13	43.6	51	**6**	39.1	26
rangesum.c	64	7	66.2	4	**1**	39.0	2
rangesum05.c	59	11	72.3	6	**1**	65.4	3
rangesum10.c	59	28	78.2	11	**16**	47.5	6
Problem01_label15.c	594	7	87.3	5	**5**	70.3	4
rangesum20.c	59	101	99.9	21	**26**	78.2	12
rangesum40.c	59	847	269.5	41	**90**	113.9	22
const.c	20	2606	796.6	1025	**890**	253.2	513
rangesum60.c	59	80272	1106.9	61	**159**	134.6	32
Average	88	6991	228.1	104	**99**	79.8	53
Total	1059	83897	2737.2	1255	**1197**	957.5	638

Preliminary Results

for large cases, the gains are substantial (*e.g.*, the verification time of `rangesum60.c` is 504x faster)

- Preliminary evaluation over

Benchmark	*LOC*	*k*-induction			Extended *k*-induction		
		T(s)	*M*(MB)	*k*	*T*(s)	*M*(MB)	*k*
sum04.c	19	**1**	38.7	9	**1**	38.7	5
sum01.c	18	**1**	38.9	11	**1**	38.8	6
sum03.c	25	3	39.1	11	**1**	38.8	6
diamond1.c	24	13	43.6	51	**6**	39.1	26
rangesum.c	64	7	66.2	4	**1**	39.0	2
rangesum05.c	59	11	72.3	6	**1**	65.4	3
rangesum10.c	59	28	78.2	11	**16**	47.5	6
Problem01_label15.c	594	7	87.3	5	**5**	70.3	4
rangesum20.c	59	101	99.9	21	**26**	78.2	12
rangesum40.c	59	847	269.5	41	**90**	113.9	22
const.c	20	2606	796.6	1025	**890**	253.2	513
rangesum60.c	59	80272	1106.9	61	**159**	134.6	32
Average	88	6991	228.1	104	**99**	79.8	53
Total	1059	83897	2737.2	1255	**1197**	957.5	638

Preliminary Results

the speed up comes from requiring roughly half the number of steps to find a property violation

- Preliminary evaluation over

Benchmark	*LOC*	*k*-induction			Extended *k*-induction		
		T(s)	*M*(MB)	*k*	*T*(s)	*M*(MB)	*k*
sum04.c	19	**1**	38.7	9	**1**	38.7	5
sum01.c	18	**1**	38.9	11	**1**	38.8	6
sum03.c	25	3	39.1	11	**1**	38.8	6
diamond1.c	24	13	43.6	51	**6**	39.1	26
rangesum.c	64	7	66.2	4	**1**	39.0	2
rangesum05.c	59	11	72.3	6	**1**	65.4	3
rangesum10.c	59	28	78.2	11	**16**	47.5	6
Problem01_label15.c	594	7	87.3	5	**5**	70.3	4
rangesum20.c	59	101	99.9	21	**26**	78.2	12
rangesum40.c	59	847	269.5	41	**90**	113.9	22
const.c	20	2606	796.6	1025	**890**	253.2	513
rangesum60.c	59	80272	1106.9	61	**159**	134.6	32
Average	88	6991	228.1	104	**99**	79.8	53
Total	1059	83897	2737.2	1255	**1197**	957.5	638

Contributions

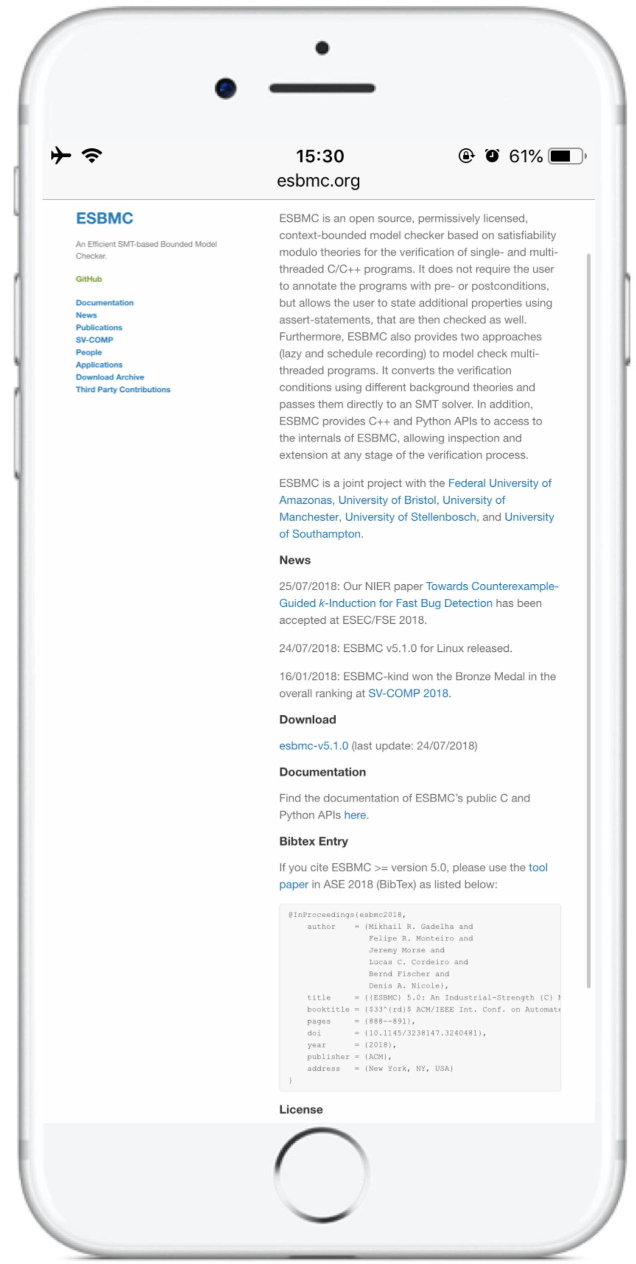
A novel extension to the *k*-induction algorithm

Contributions

- Our main contribution is **a novel extension to the *k*-induction algorithm**, to perform a bidirectional search instead of the conventional iterative deepening search:
  - the preliminary results show that the extension has the potential to substantially improve the verification time for problems with large state space, while maintaining a small verification time for small programs.
- As **future work**, we plan to expand our evaluation over the SV-COMP benchmarks, where the original k-induction algorithm already proved to be the state-of-art, if compared to other *k*-induction tools

"**The main challenge is scalability**: real-world software systems not only include complex control and data structure, but depend on much "context" such as libraries and interfaces to other code, including lower-level systems code. As a result, proving a software system correct requires much more effort, knowledge, training, and ingenuity than writing the software in trial-and-error style."

–E. M. Clarke et al., Handbook of Model Checking, 2018.



Scan me