Boosting *k*-Induction with Continuously-Refined Invariants

Accepted to CAV'15

Matthias Dangl

Joint work with Dirk Beyer and Philipp Wendler

University of Passau, Germany

UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

CPA✓

SoSy-Lab
Software Systems

k-Induction for Software Verification

- Bounded Model Checking (BMC) is successful for finding bugs
- But not all loop bounds are small enough or even known/computable
- BMC is good for falsification, but often cannot prove absence of bugs

k-Induction for Software Verification

- Bounded Model Checking (BMC) is successful for finding bugs
- But not all loop bounds are small enough or even known/computable
- BMC is good for falsification, but often cannot prove absence of bugs
- (*k*-)Induction extends BMC towards unbounded safety proofs

1-Induction

- 1-Induction:
  - Check that the safety property holds in the first loop iteration: $P(1)$
  - Equivalent to BMC with loop bound 1
  - Check that the safety property is 1-inductive: $\forall n : P(n) \implies P(n+1)$

k-Induction

- *k*-Induction generalizes the induction principle:
  - Check that the property holds in the first k iterations:
    $$\bigwedge_{i=1}^{k} P(i)$$
  - Equivalent to BMC with loop bound k
  - Check that the safety property is k-inductive:
    $$\forall n : \left(\bigwedge_{i=1}^{k} P(n+i-1) \right) \implies P(n+k)$$
  - Stronger hypothesis is more likely to succeed [Wahl'13]
- Iteratively increase k

k-Induction

- k-Induction generalizes the induction principle:
  - Check that the property holds in the first k iterations: $\bigwedge_{i=1}^{k} P(i)$
  - Equivalent to BMC with loop bound k
  - Check that the safety property is k-inductive: $\forall n : \left(\bigwedge_{i=1}^{k} P(n+i-1) \right) \implies P(n+k)$
  - Stronger hypothesis is more likely to succeed [Wahl'13]
- Iteratively increase k
- Done, next talk?

k-Induction

- *k*-Induction generalizes the induction principle:
  - Check that the property holds in the first *k* iterations:
    $$\bigwedge_{i=1}^{k} P(i)$$
  - Equivalent to BMC with loop bound *k*
  - Check that the safety property is *k*-inductive:
    $$\forall n : \left(\bigwedge_{i=1}^{k} P(n+i-1) \right) \implies P(n+k)$$
  - Stronger hypothesis is more likely to succeed [Wahl'13]
- Iteratively increase *k*
- Done, next talk?
- No!

Example

```
int main() {
  unsigned int x1 = 0, x2 = 0;        → data variables
  int s = 1;                          → state variable

  while (nondet()) {                  → unbounded loop
    if (s == 1) x1++;
    else if (s == 2) x2++;            → some calculations

    s++;
    if (s == 5) s = 1;                → state computation

    if (s == 1) assert(x1 == x2);     → safety property
  }
  return 0;
}
```

Example

```
int main() {
  unsigned int x1 = 0, x2 = 0;
  int s = 1;

  while (nondet()) {
    if (s == 1) x1++;
    else if (s == 2) x2++;

    s++;
    if (s == 5) s = 1;

    if (s == 1) assert(x1 == x2);
  }
  return 0;
}
```

- Explicit state analysis?

Example

```
int main() {
  unsigned int x1 = 0, x2 = 0;
  int s = 1;

  while (nondet()) {
    if (s == 1) x1++;
    else if (s == 2) x2++;

    s++;
    if (s == 5) s = 1;

    if (s == 1) assert(x1 == x2);
  }
  return 0;
}
```

- Explicit state analysis?
  Too many states.
- Predicate analysis?

Example

```
int main() {
  unsigned int x1 = 0, x2 = 0;
  int s = 1;

  while (nondet()) {
    if (s == 1) x1++;
    else if (s == 2) x2++;

    s++;
    if (s == 5) s = 1;

    if (s == 1) assert(x1 == x2);
  }
  return 0;
}
```

- Explicit state analysis?
  Too many states.
- Predicate analysis?
  “Interpolants suck”
- Intervals, Octagons?

Example

```
int main() {
  unsigned int x1 = 0, x2 = 0;
  int s = 1;

  while (nondet()) {
    if (s == 1) x1++;
    else if (s == 2) x2++;

    s++;
    if (s == 5) s = 1;

    if (s == 1) assert(x1 == x2);
  }
  return 0;
}
```

- Explicit state analysis?
  Too many states.
- Predicate analysis?
  "Interpolants suck"
- Intervals, Octagons?
  Too imprecise.
- BMC?

Example

```
int main() {
  unsigned int x1 = 0, x2 = 0;
  int s = 1;

  while (nondet()) {
    if (s == 1) x1++;
    else if (s == 2) x2++;

    s++;
    if (s == 5) s = 1;

    if (s == 1) assert(x1 == x2);
  }
  return 0;
}
```

- Explicit state analysis?
  Too many states.
- Predicate analysis?
  “Interpolants suck”
- Intervals, Octagons?
  Too imprecise.
- BMC?
  Unbounded loop.
- 1-Induction?

Example

```
int main() {
  unsigned int x1 = 0, x2 = 0;
  int s = 1;

  while (nondet()) {
    if (s == 1) x1++;
    else if (s == 2) x2++;

    s++;
    if (s == 5) s = 1;

    if (s == 1) assert(x1 == x2);
  }
  return 0;
}
```

- Explicit state analysis?
  Too many states.
- Predicate analysis?
  "Interpolants suck"
- Intervals, Octagons?
  Too imprecise.
- BMC?
  Unbounded loop.
- 1-Induction?
  Hypothesis too weak.
- *k*-Induction

Example

```
int main() {
  unsigned int x1 = 0, x2 = 0;
  int s = 1;

  while (nondet()) {
    if (s == 1) x1++;
    else if (s == 2) x2++;

    s++;
    if (s == 5) s = 1;

    if (s == 1) assert(x1 == x2);
  }
  return 0;
}
```

- Explicit state analysis?
  Too many states.
- Predicate analysis?
  “Interpolants suck”
- Intervals, Octagons?
  Too imprecise.
- BMC?
  Unbounded loop.
- 1-Induction?
  Hypothesis too weak.
- *k*-Induction
  Hypothesis too weak!
  Needs $s > 0$

Further Strengthening

- Proofs still fail too often
- Introduce auxiliary invariants to strengthen the hypothesis:

$$\forall n : \left(\mathbf{Inv(n)} \wedge \bigwedge_{i=1}^{k} P(n+i-1) \right) \implies P(n+k)$$

- Auxilary invariants must hold
- Auxiliary invariants must be inductive
- Where do these invariants come from?

Auxiliary Invariants

- An additional component provides auxiliary invariants: The invariant generator
- Should be strong enough so that the proof succeeds
- Should not waste more resources than necessary

Experimental Results for *k*-Induction with static Invariant Generation by Abstract Interpretation

2 814 verification tasks taken from SV-COMP'15

Approach	KI		KI←AI	
		weakest	weak	strongest
Correct results	1 082	1 900	**1 934**	1 861
CPU time (h)	380	190	**180**	200
k-Values for correct safe results only:				
Max. final *k*	101	101	100	**86**

Powered by **BenchExec**

Continuously-Refined Invariants

- An additional component provides auxiliary invariants: The invariant generator
- Should be strong enough so that the proof succeeds
- Should not waste more resources than necessary

Continuously-Refined Invariants

- An additional component provides auxiliary invariants: The invariant generator
- Should be strong enough so that the proof succeeds
- Should not waste more resources than necessary
- But no single fixed-precision configuration can provide this!

Continuously-Refined Invariants

- An additional component provides auxiliary invariants: The invariant generator
- Should be strong enough so that the proof succeeds
- Should not waste more resources than necessary
- But no single fixed-precision configuration can provide this!
- Invariant generator can be run in parallel and provide invariants continuously
- Invariant generator improves invariants continuously over time
- Pick up current set of auxiliary invariants in each k-Induction iteration

Algorithm

Induction:

```
$k = 0$
while !finished do
    BMC(k)
    Induction($k$, invariants)
    $k$++
```

Invariant generation:

```
prec = <weak>
invariants = $\emptyset$
while !finished do
    invariants = GenInv(prec)
    prec = RefinePrec(prec)
```

Invariant Generation

How to generate invariants?

Invariant Generation

How to generate invariants?

- Option 1: Abstract Interpretation

Invariant Generation

How to generate invariants?

- Option 1: Abstract Interpretation
- Option 2: Candidate-based approaches
  e.g. Kahsai, Tinelli: PKind [PDMC'11]

Invariant Generation

How to generate invariants?

- Option 1: Abstract Interpretation
- Option 2: Candidate-based approaches
  e.g. Kahsai, Tinelli: PKind [PDMC'11]
- Option 3: Policy Iteration (see next talk)

Invariant Generation

How to generate invariants?

- Option 1: Abstract Interpretation
- Option 2: Candidate-based approaches
  e.g. Kahsai, Tinelli: PKind [PDMC'11]
- Option 3: Policy Iteration (see next talk)
- ...

Experimental Results for *k*-Induction with Continuously-Refined Invariants

- 2 814 verification tasks taken from SV-COMP'15
- Best static configuration solved **1 934** tasks in **180** CPU hours

Approach	KI	KI←⟲−AI	KI←⟲−KI	KI←⟲−KI←⟲−AI
Correct Results	1 082	1 984	1 690	**2 005**
CPU Time (h)	380	170	240	**170**

Powered by **BenchExec**

k-Induction in Other Tools: Comparison

Tool	CBMC	ESBMC		CPACHECKER
Configuration		sequential	parallel	KI←⟲–KI←⟲–AI
Correct results	1 216	2 214	2 137	2 005
Wrong proofs	261	184	137	4
Wrong alarms	4	28	24	25
CPU time (h)	350	100	130	170

Powered by **BenchExec**

Conclusion

- *k*-Induction for software verification is feasible

Conclusion

- *k*-Induction for software verification is feasible
- *k*-Induction for software verification requires auxiliary invariants

Conclusion

- *k*-Induction for software verification is feasible
- *k*-Induction for software verification requires auxiliary invariants
- Auxiliary invariants should be continously refined

Conclusion

- *k*-Induction for software verification is feasible
- *k*-Induction for software verification requires auxiliary invariants
- Auxiliary invariants should be continously refined
- Combinations of KI and AI techniques are successful

Conclusion

- *k*-Induction for software verification is feasible
- *k*-Induction for software verification requires auxiliary invariants
- Auxiliary invariants should be continously refined
- Combinations of KI and AI techniques are successful
- Unsound approaches are not worth their trouble
- Bounded model checkers can easily be extended to provide proofs

Conclusion

- *k*-Induction for software verification is feasible
- *k*-Induction for software verification requires auxiliary invariants
- Auxiliary invariants should be continously refined
- Combinations of KI and AI techniques are successful
- Unsound approaches are not worth their trouble
- Bounded model checkers can easily be extended to provide proofs
- Read the upcoming paper:
  Boosting *k*-Induction with Continuously-Refined Invariants [CAV'15]
  ... or email me at `dangl@fim.uni-passau.de`