Configurable Software Model Checking

Part 2: A Unifying View on SMT-Based Software Verification

Dirk Beyer

LMU Munich, Germany

Based on:

Dirk Beyer, Matthias Dangl, Philipp Wendler:

A Unifying View on SMT-Based Software Verification

Journal of Automated Reasoning, Volume 60, Issue 3, 2018

SMT-based Software Model Checking

- Predicate Abstraction
  (BLAST, CPACHECKER, SLAM, ...)
- IMPACT
  (CPACHECKER, IMPACT, WOLVERINE, ...)
- Bounded Model Checking
  (CBMC, CPACHECKER, ESBMC, ...)
- *k*-Induction
  (CPACHECKER, ESBMC, 2LS, ...)

Motivation

- Theoretical comparison difficult:
  - different conceptual optimizations
    (e.g., large-block encoding)
  - different presentation

$\rightarrow$ What are their core concepts and key differences?

Motivation

- Theoretical comparison difficult:
  - different conceptual optimizations (e.g., large-block encoding)
  - different presentation

$\rightarrow$ What are their core concepts and key differences?

- Experimental comparison difficult:
  - implemented in different tools
  - different technical optimizations (e.g., data structures)
  - different front-end and utility code
  - different SMT solver

$\rightarrow$ Where do performance differences actually come from?

Goals

- Provide a unifying framework for SMT-based algorithms
- Understand differences and key concepts of algorithms
- Determine potential of extensions and combinations
- Provide solid platform for experimental research

Approach

- Understand, and, if necessary, re-formulate the algorithms
- Design a configurable framework for SMT-based algorithms (based upon the CPA framework)
- Use flexibility of adjustable-block encoding (ABE)
- Express existing algorithms using the common framework
- Implement framework (in CPAchecker)

Base: Adjustable-Block Encoding

Originally for predicate abstraction:

- Abstraction computation is expensive
- Abstraction is not necessary after every transition
- Track precise path formula between abstraction states
- Reset path formula and compute abstraction formula at abstraction states
- Large-Block Encoding:
  abstraction only at loop heads (hard-coded)
- Adjustable-Block Encoding:
  introduce block operator "blk" to make it configurable

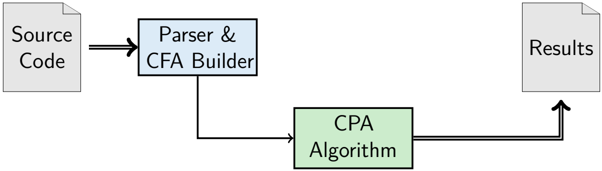
Base: Configurable Program Analysis

Configurable Program Analysis (CPA):

- Beyer, Henzinger, Théoduloz: [CAV'07]
- One single unifying algorithm for all algorithms based on state-space exploration
- Configurable components: abstract domain, abstract-successor computation, path sensitivity, ...

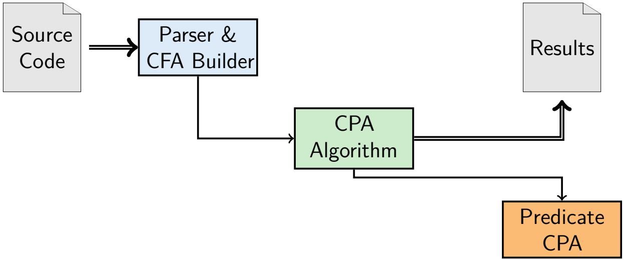
Using the CPA Framework

- CPA Algorithm is a configurable reachability analysis for arbitrary abstract domains



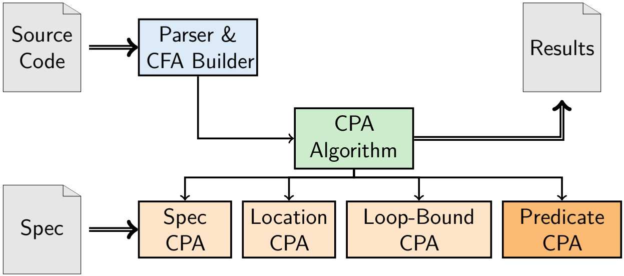
Using the CPA Framework

- CPA Algorithm is a configurable reachability analysis for arbitrary abstract domains
- Provide Predicate CPA for our predicate-based abstract domain



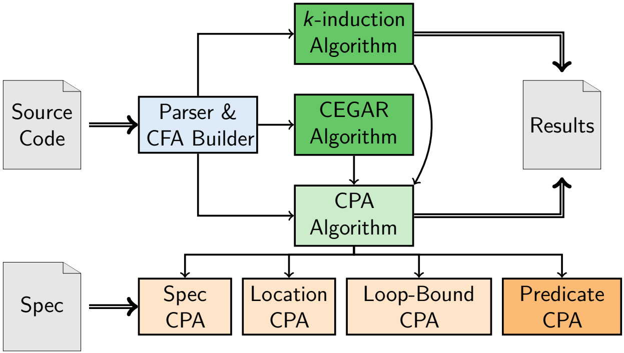
Using the CPA Framework

- CPA Algorithm is a configurable reachability analysis for arbitrary abstract domains
- Provide Predicate CPA for our predicate-based abstract domain
- Reuse other CPAs

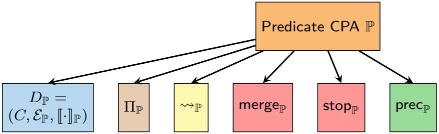


Using the CPA Framework

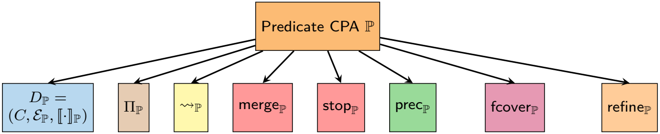
- CPA Algorithm is a configurable reachability analysis for arbitrary abstract domains
- Provide Predicate CPA for our predicate-based abstract domain
- Reuse other CPAs
- Built further algorithms on top that make use of reachability analysis



Predicate CPA



Predicate CPA



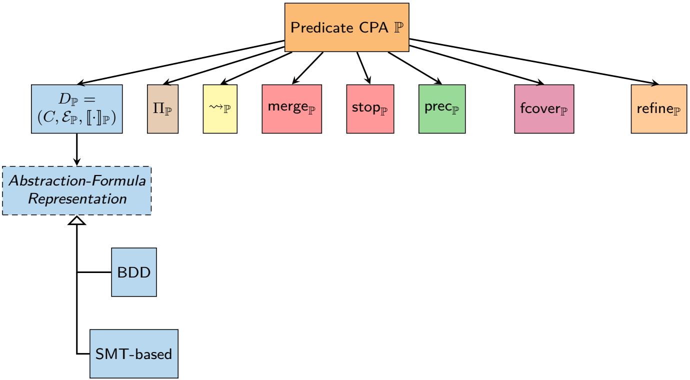
Predicate CPA: Abstract Domain

- Abstract state: (ψ, φ)
  - tuple of abstraction formula ψ and path formula φ (for ABE)
  - conjunctions represents state space
  - abstraction formula can be a BDD or an SMT formula
  - path formula is always SMT formula and concrete

Predicate CPA: Abstract Domain

- Abstract state: (ψ, φ)
  - tuple of abstraction formula ψ and path formula φ (for ABE)
  - conjunctions represents state space
  - abstraction formula can be a BDD or an SMT formula
  - path formula is always SMT formula and concrete
- Precision: set of predicates (per program location)

Predicate CPA



Predicate CPA: CPA Operators

- Transfer relation:
  - computes strongest post
  - changes only path formula, new abstract state is (ψ, φ')
  - purely syntactic, cheap
  - variety of encodings using different SMT theories possible
    (different approximations
    for arithmetic and heap operations)

Predicate CPA: CPA Operators

- Transfer relation:
  - computes strongest post
  - changes only path formula, new abstract state is (ψ, φ')
  - purely syntactic, cheap
  - variety of encodings using different SMT theories possible (different approximations for arithmetic and heap operations)
- Merge operator:
  - standard for ABE: create disjunctions inside block

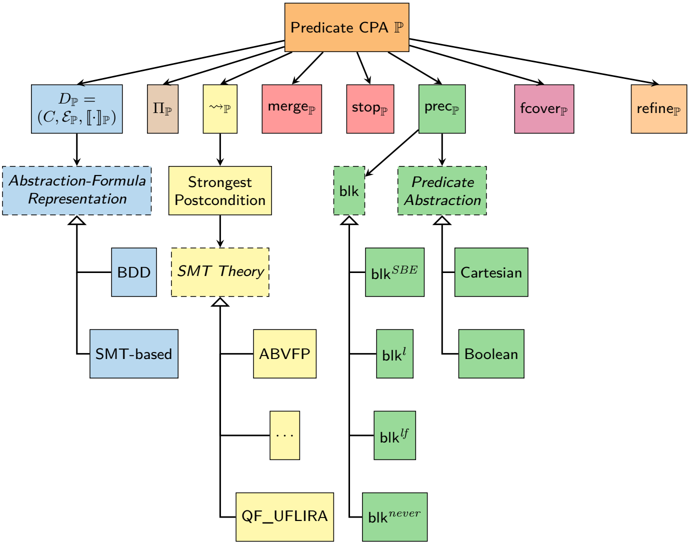
Predicate CPA: CPA Operators

- Transfer relation:
  - computes strongest post
  - changes only path formula, new abstract state is (ψ, φ')
  - purely syntactic, cheap
  - variety of encodings using different SMT theories possible (different approximations
    for arithmetic and heap operations)
- Merge operator:
  - standard for ABE: create disjunctions inside block
- Stop operator:
  - standard for ABE: check coverage only at block ends

Predicate CPA: CPA Operators

- Transfer relation:
  - computes strongest post
  - changes only path formula, new abstract state is (ψ, φ')
  - purely syntactic, cheap
  - variety of encodings using different SMT theories possible (different approximations for arithmetic and heap operations)
- Merge operator:
  - standard for ABE: create disjunctions inside block
- Stop operator:
  - standard for ABE: check coverage only at block ends
- Precision-adjustment operator:
  - only active at block ends (as determined by blk)
  - computes abstraction of current abstract state
  - new abstract state is $(\psi', true)$

Predicate CPA



Predicate CPA: Refinement

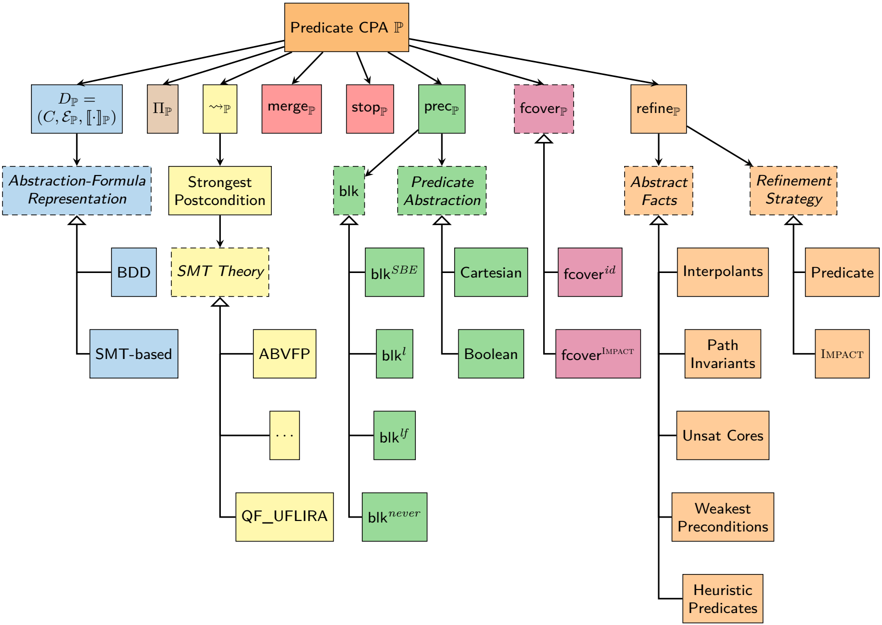
Four steps:

1. Reconstruct ARG path to abstract error state
2. Check feasibility of path
3. Discover abstract facts, e.g.,
   - interpolants
   - weakest precondition
   - heuristics
4. Refine abstract model
   - add predicates to precision, cut ARG

   or

   - conjoin interpolants to abstract states, recheck coverage relation

Predicate CPA



Predicate Abstraction

- Predicate Abstraction
  - [CAV'97, POPL'02, J. ACM'03, POPL'04]
  - Abstract-interpretation technique
  - Abstract domain constructed from a set of predicates π
  - Use CEGAR to add predicates to π (refinement)
  - Derive new predicates using Craig interpolation
  - Abstraction formula as BDD

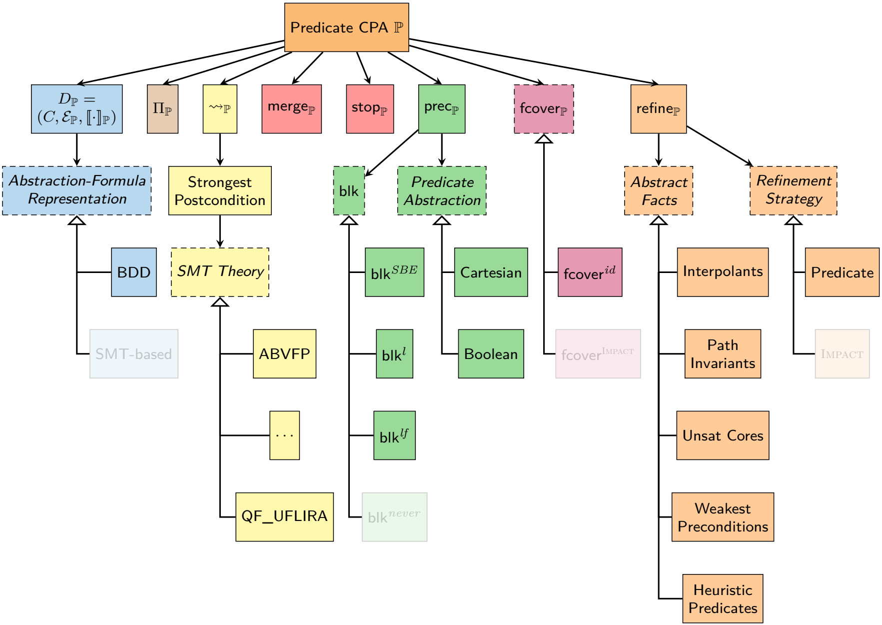
Expressing Predicate Abstraction

- Abstraction Formulas: BDDs
- Block Size (blk): e.g. blk^{SBE} or blk^{l} or blk^{lf}
- Refinement Strategy: add predicates to precision, cut ARG

Use CEGAR Algorithm:

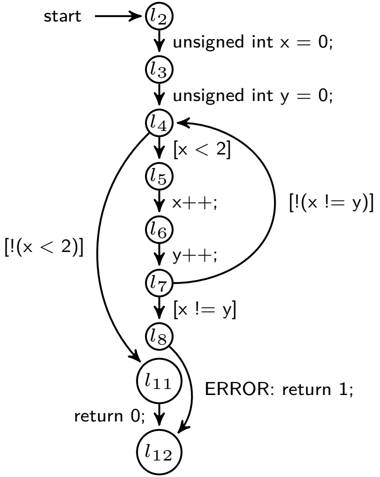
```
while true do
    run CPA Algorithm
    if target state found then
        call refine
        if target state reachable then
            return false
    else
        return true
```

Predicate CPA

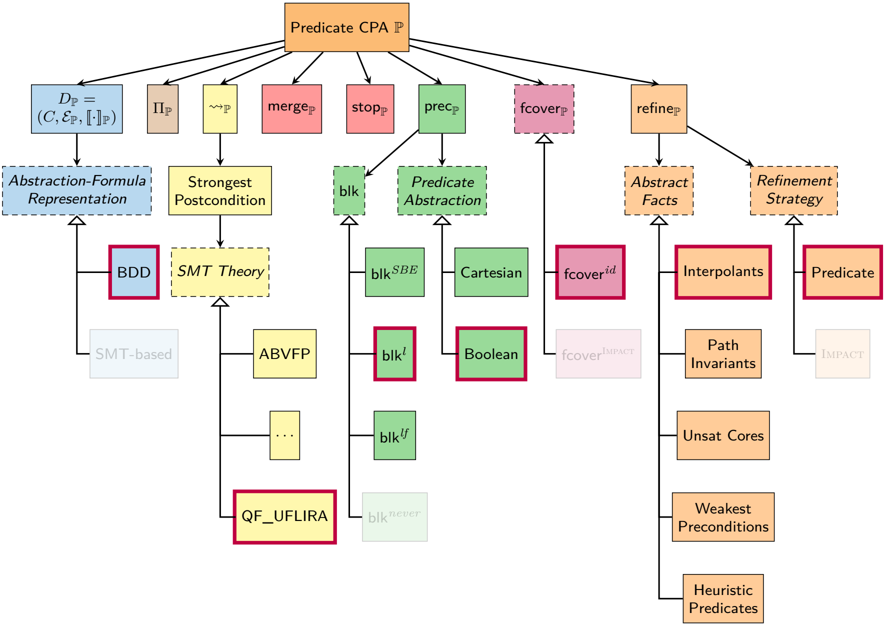


Example Program

```
int main() {
  unsigned int x = 0;
  unsigned int y = 0;
  while (x < 2) {
    x++;
    y++;
    if (x != y) {
      ERROR: return 1;
    }
  }
  return 0;
}
```

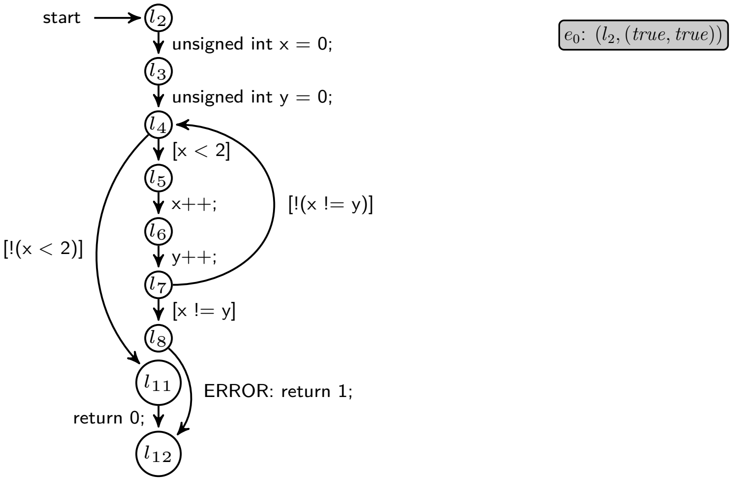


Predicate CPA



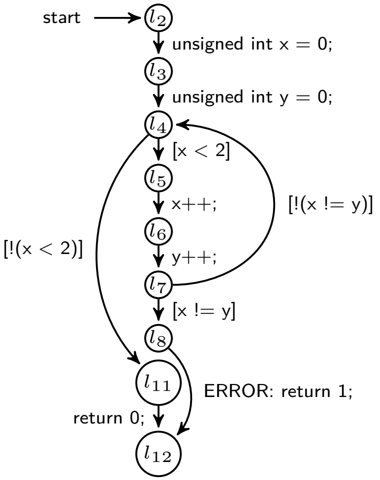
Predicate Abstraction: Example

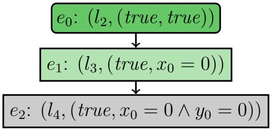
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$



Predicate Abstraction: Example

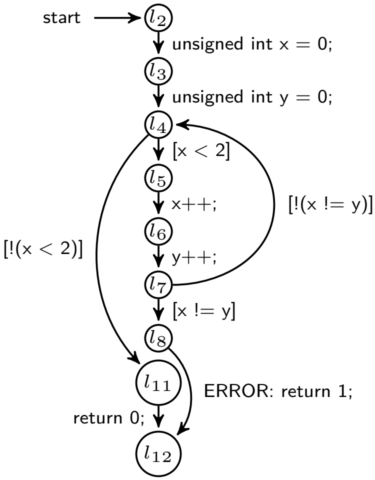
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

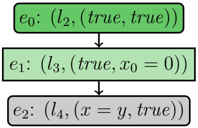




Predicate Abstraction: Example

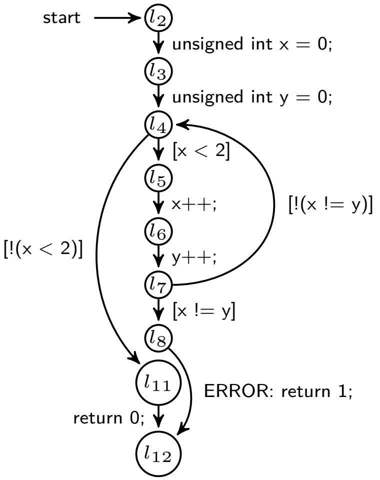
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

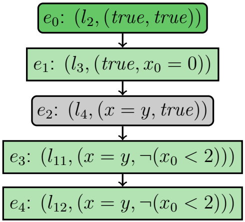




Predicate Abstraction: Example

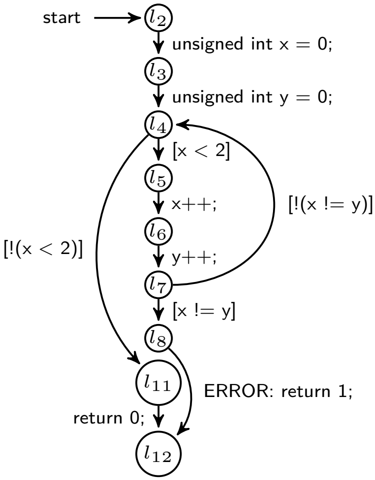
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

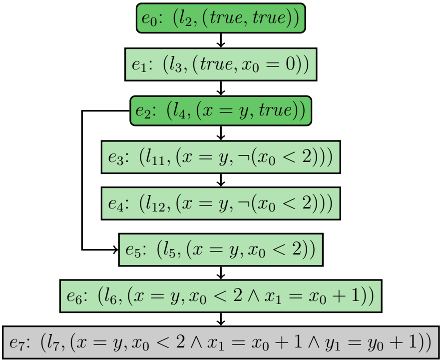




Predicate Abstraction: Example

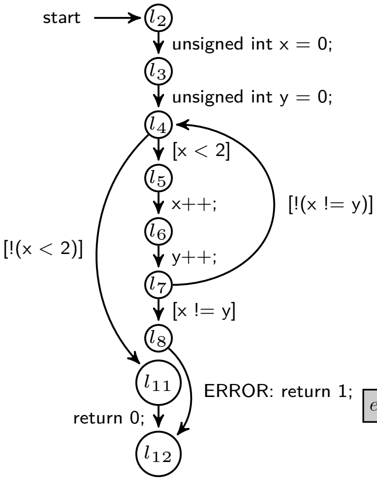
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

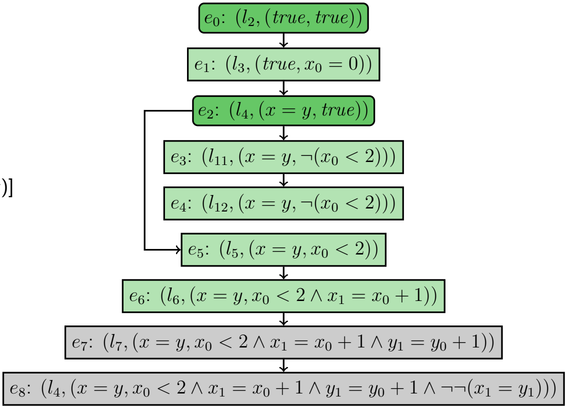




Predicate Abstraction: Example

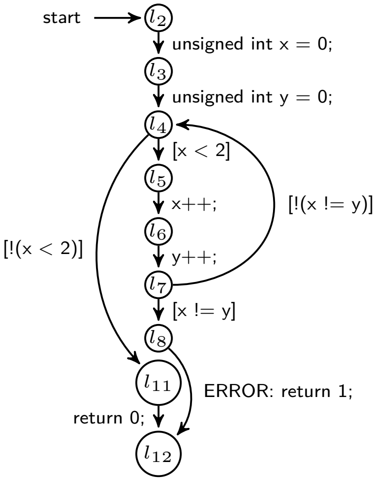
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

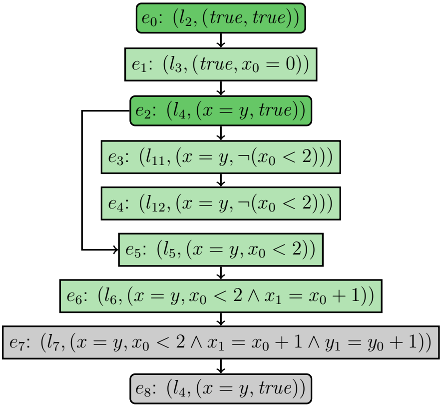




Predicate Abstraction: Example

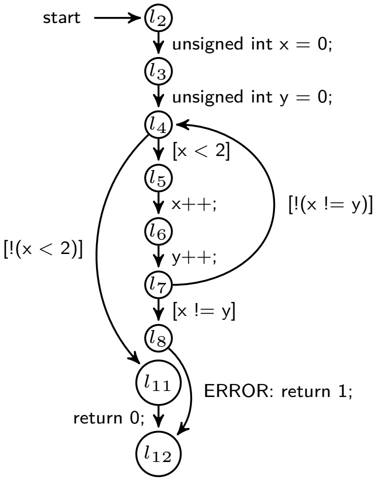
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

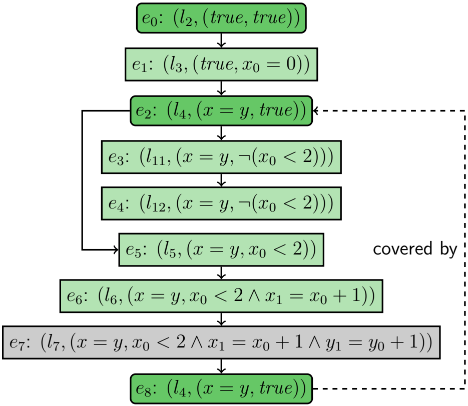




Predicate Abstraction: Example

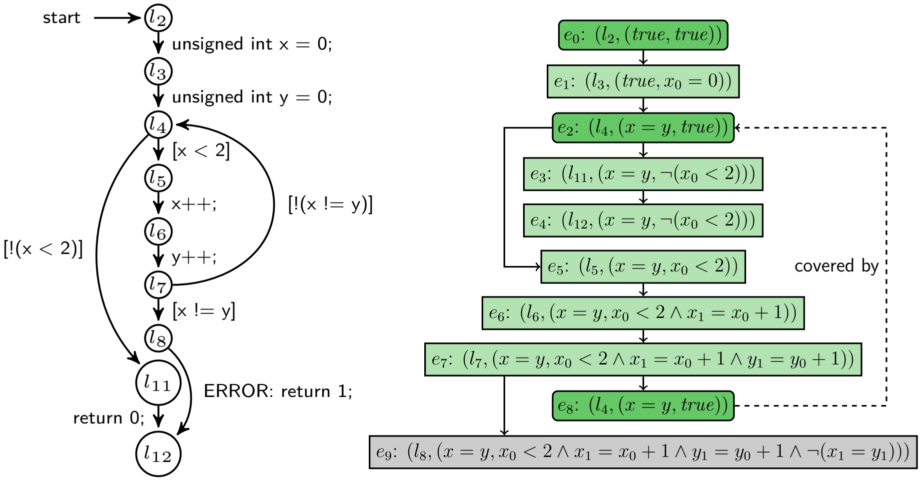
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$





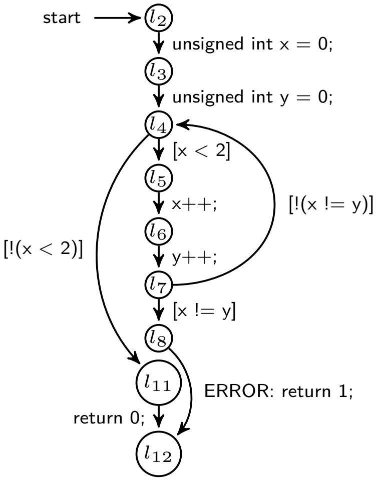
Predicate Abstraction: Example

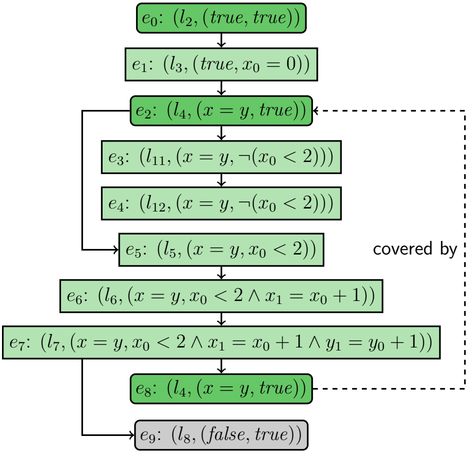
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$



Predicate Abstraction: Example

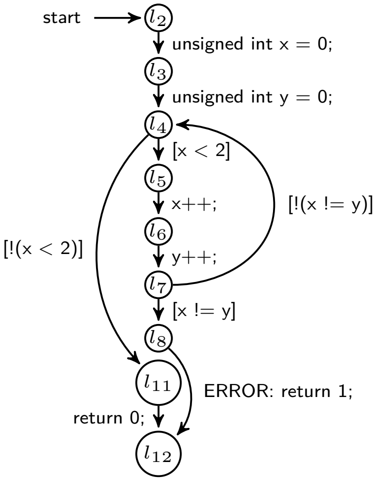
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

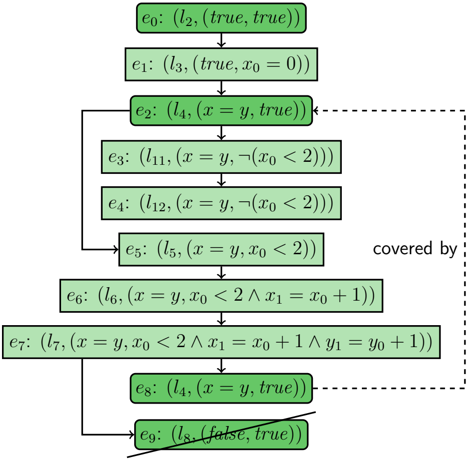




Predicate Abstraction: Example

with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$





IMPACT

- IMPACT
  - "Lazy Abstraction with Interpolants" [CAV'06]
  - Abstraction is derived dynamically/lazily
  - Solution to avoiding expensive abstraction computations
  - Compute fixed point over three operations
    - Expand
    - Refine
    - Cover
  - Abstraction formula as SMT formula
  - Optimization: forced covering

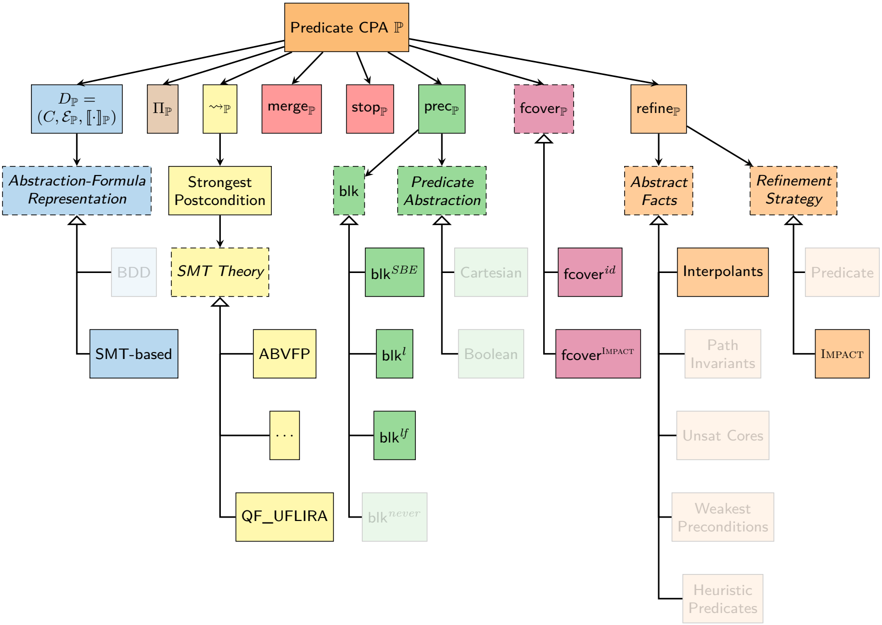
Expressing IMPACT

- Abstraction Formulas: SMT-based
- Block Size (blk): blk^{SBE} or other (new!)
- Refinement Strategy:
  conjoin interpolants to abstract states,
  recheck coverage relation

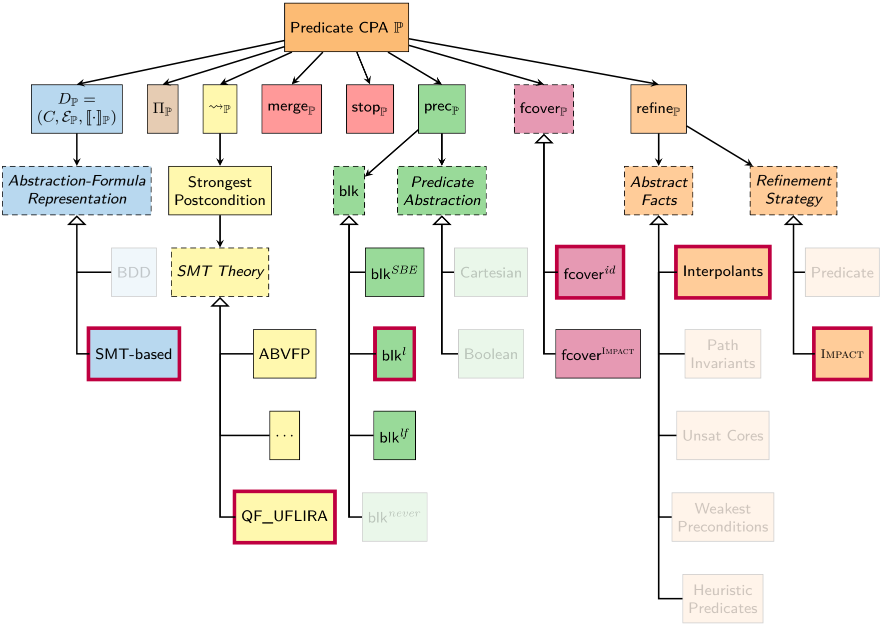
Furthermore:

- Use CEGAR Algorithm
- Precision stays empty
  $\rightarrow$ predicate abstraction never computed

Predicate CPA

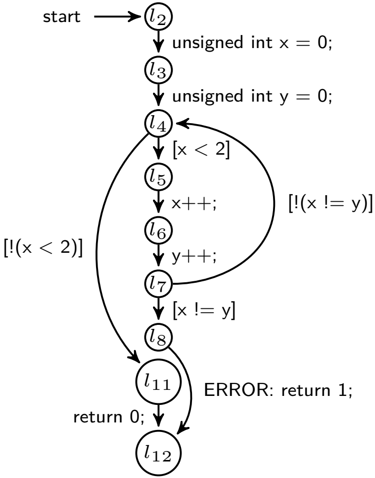


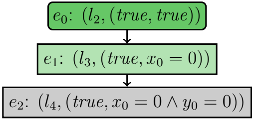
Predicate CPA



Impact: Example

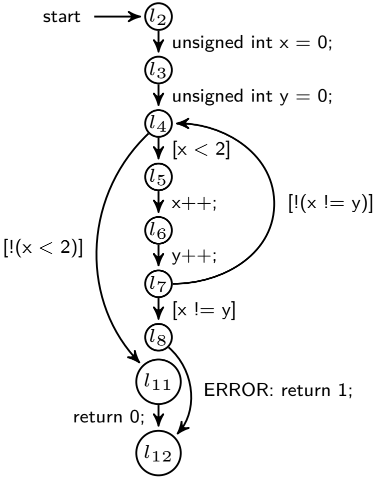
with blk^l

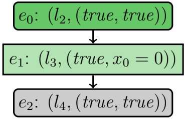




IMPACT: Example

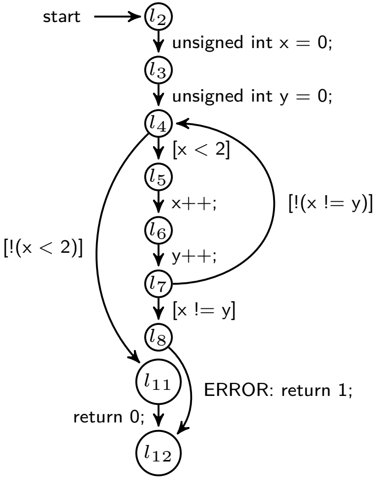
with blk^l

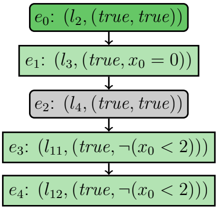




Impact: Example

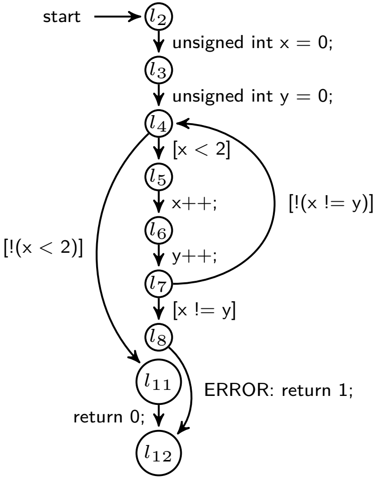
with blk^l

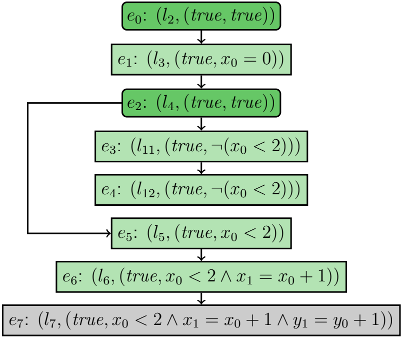




IMPACT: Example

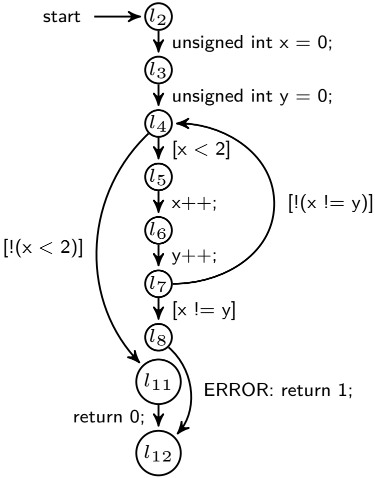
with blk^l

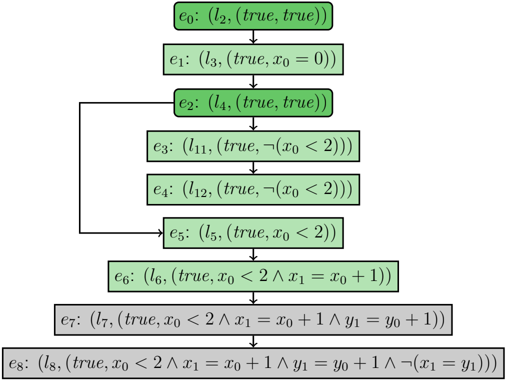




Impact: Example

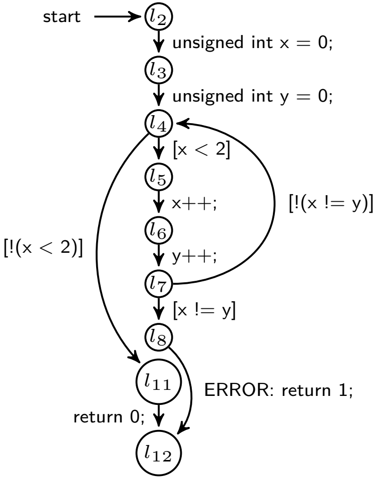
with blk^l

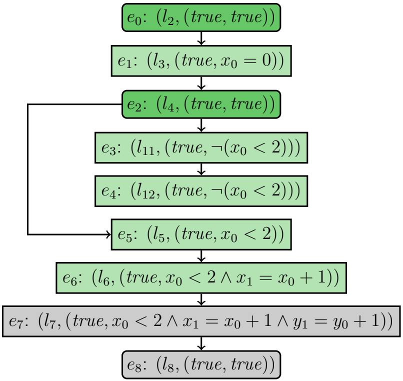




Impact: Example

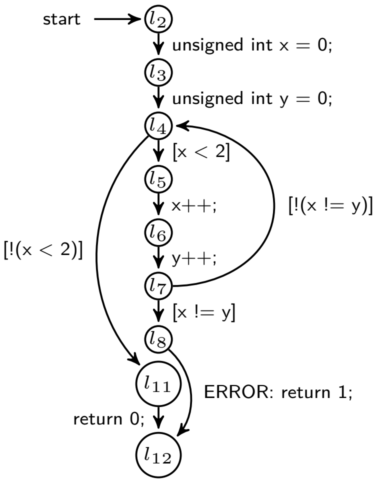
with blk^l

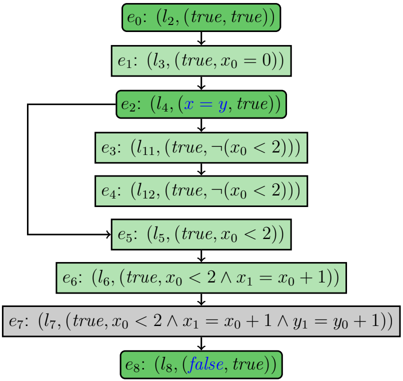




Impact: Example

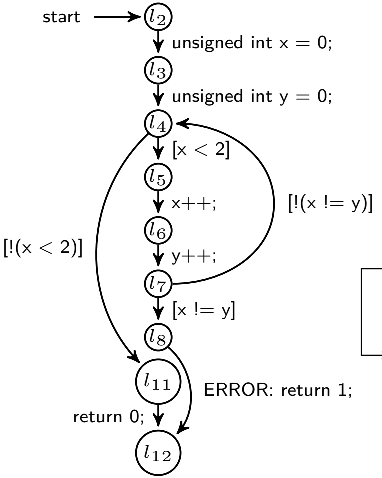
with blk^l

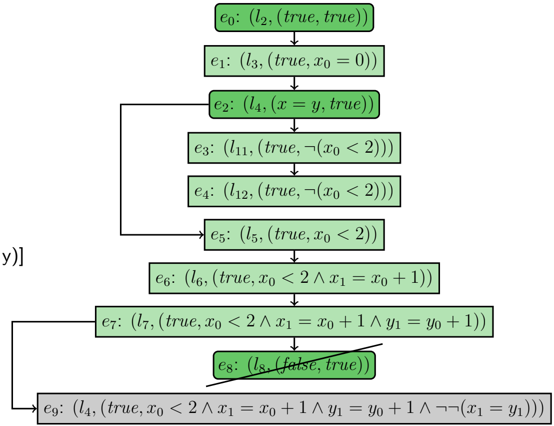




IMPACT: Example

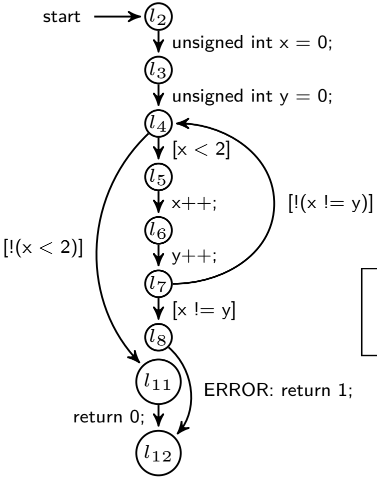
with blk^l

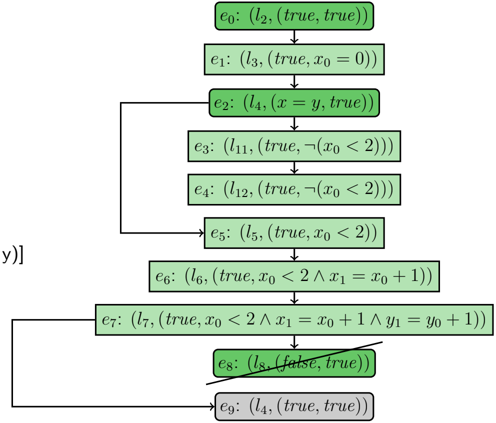




IMPACT: Example

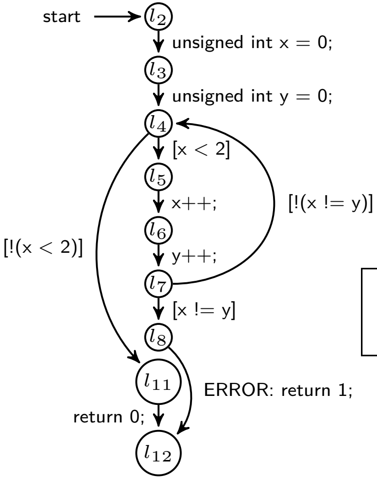
with blk^{l}

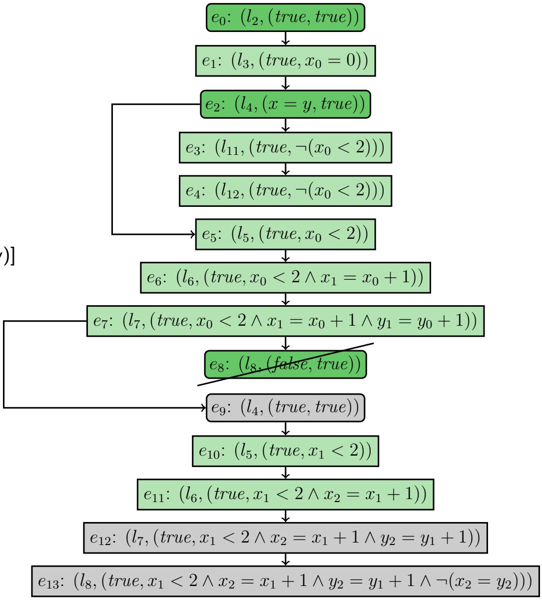




IMPACT: Example

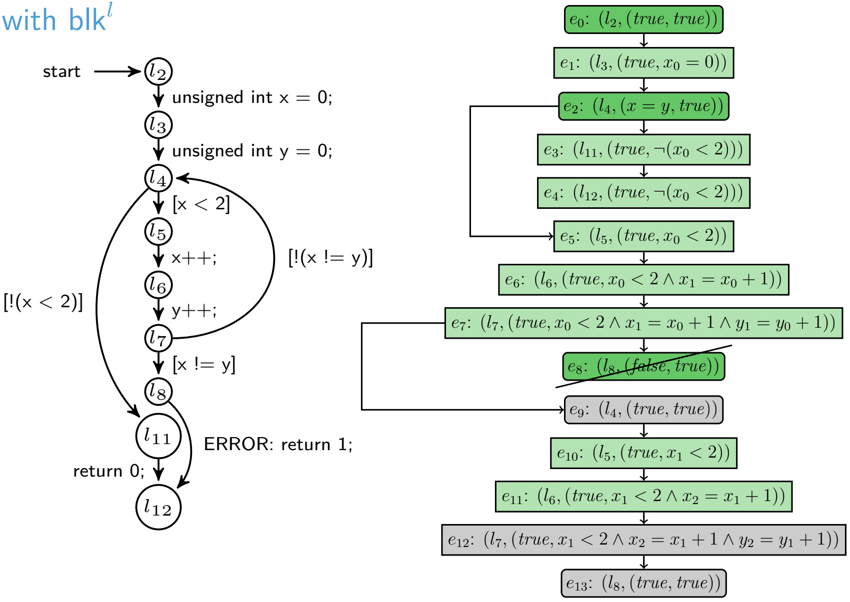
with blk^l





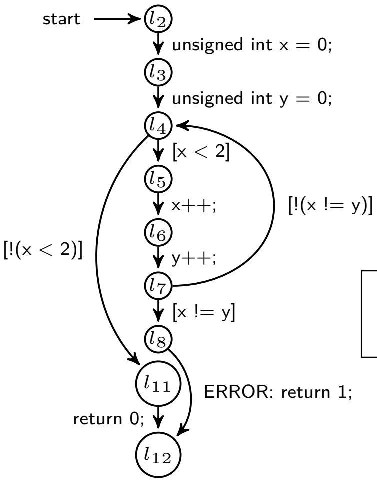
Impact: Example

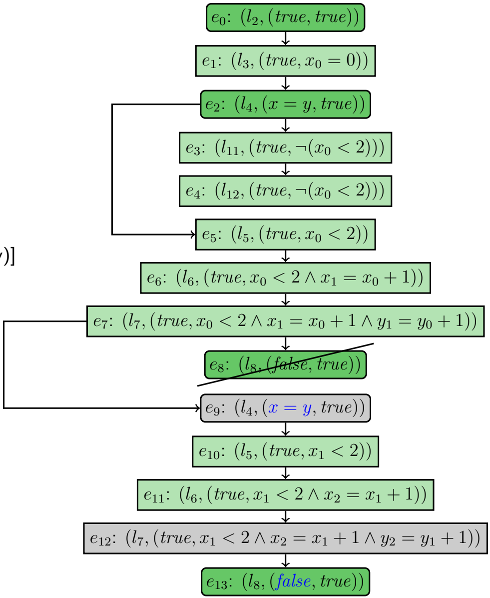
with blk^l



IMPACT: Example

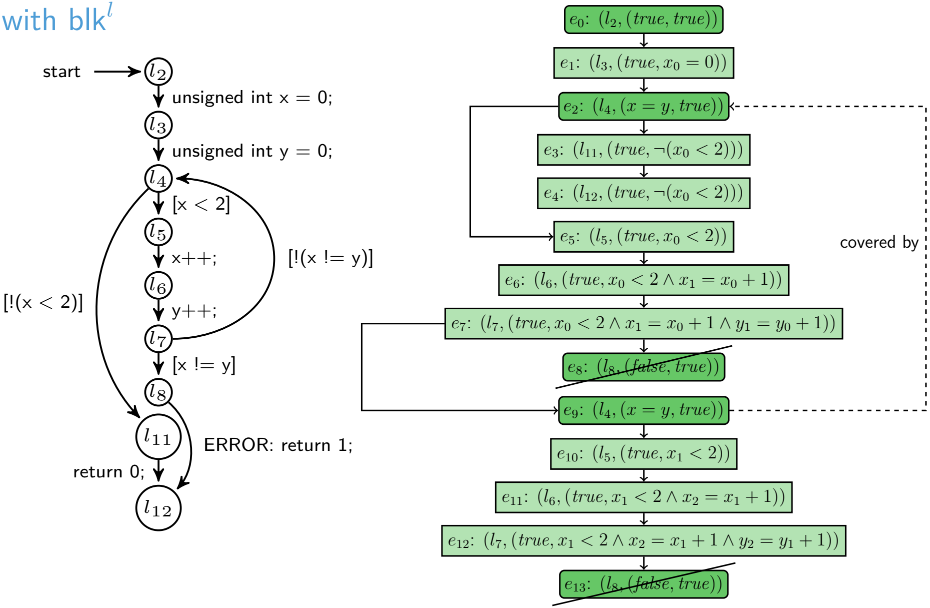
with blk^l





Impact: Example

with blk^l



Bounded Model Checking

- Bounded Model Checking:
  - Biere, Cimatti, Clarke, Zhu: [TACAS'99]
  - No abstraction
  - Unroll loops up to a loop bound k
  - Check that P holds in the first k iterations:

$$\bigwedge_{i=1}^{k} P(i)$$

Expressing BMC

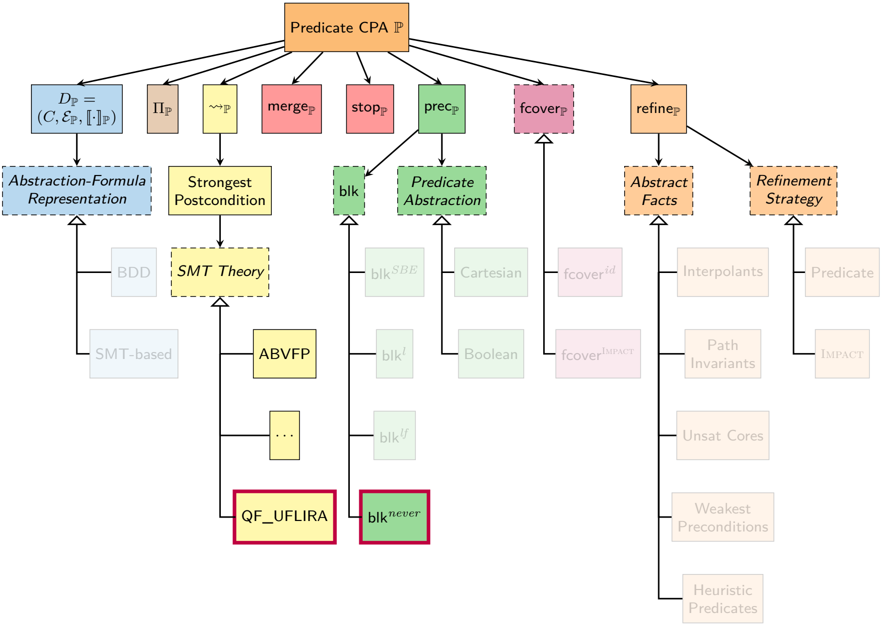
- Block Size (blk): blk^{never}

Furthermore:

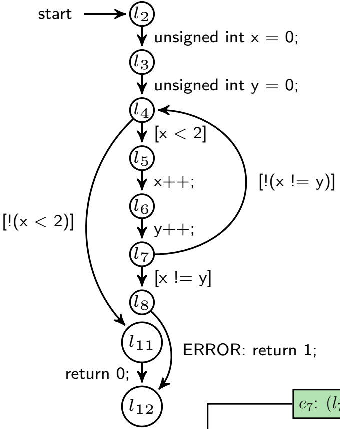
- Add CPA for bounding state space (e.g., loop bounds)
- Choices for abstraction formulas and refinement irrelevant because block end never encountered
- Use Algorithm for iterative BMC:

  $k = 1$
  while !finished **do**
      run CPA Algorithm
      check feasibility of each abstract error state
      $k{+}{+}$

Predicate CPA



Bounded Model Checking: Example with $k = 1$



e_0: $(l_2, (\mathit{true}, \mathit{true}), \{l_4 \mapsto -1\})$

e_1: $(l_3, (\mathit{true}, x_0 = 0), \{l_4 \mapsto -1\})$

e_2: $(l_4, (\mathit{true}, x_0 = 0 \wedge y_0 = 0), \{l_4 \mapsto 0\})$

e_3: $(l_{11}, (\mathit{true}, x_0 = 0 \wedge y_0 = 0 \wedge \neg(x_0 < 2)), \{l_4 \mapsto 0\})$

e_4: $(l_{12}, (\mathit{true}, x_0 = 0 \wedge y_0 = 0 \wedge \neg(x_0 < 2)), \{l_4 \mapsto 0\})$

e_5: $(l_5, (\mathit{true}, x_0 = 0 \wedge y_0 = 0 \wedge x_0 < 2), \{l_4 \mapsto 0\})$

e_6: $(l_6, (\mathit{true}, x_0 = 0 \wedge y_0 = 0 \wedge x_0 < 2 \wedge x_1 = x_0 + 1), \{l_4 \mapsto 0\})$

e_7: $(l_7, (\mathit{true}, x_0 = 0 \wedge y_0 = 0 \wedge x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1), \{l_4 \mapsto 0\})$

e_8: $(l_8, (\mathit{true}, x_0 = 0 \wedge y_0 = 0 \wedge x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(x_1 = y_1)), \{l_4 \mapsto 0\})$

e_9: $(l_{12}, (\mathit{true}, x_0 = 0 \wedge y_0 = 0 \wedge x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(x_1 = y_1)), \{l_4 \mapsto 0\})$

e_{10}: $(l_4, (\mathit{true}, x_0 = 0 \wedge y_0 = 0 \wedge x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(\neg(x_1 = y_1))), \{l_4 \mapsto 1\})$

1-Induction

- 1-Induction:
  - Base case: Check that the safety property holds in the first loop iteration:

    $$P(1)$$

    $\rightarrow$ Equivalent to BMC with loop bound 1
  - Step case: Check that the safety property is 1-inductive:

    $$\forall n : (P(n) \Rightarrow P(n+1))$$

k-Induction

- k-Induction generalizes the induction principle:
  - No abstraction
  - Base case: Check that P holds in the first k iterations:
    $\rightarrow$ Equivalent to BMC with loop bound k
  - Step case: Check that the safety property is k-inductive:

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

  - Stronger hypothesis is more likely to succeed
  - Add auxiliary invariants
  - Kahsai, Tinelli: [PDMC'11]

k-Induction with Auxiliary Invariants

Induction:

$k = 1$
while !finished **do**
  BMC(k)
  Induction(k, invariants)
  k++

Invariant generation:

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

k-Induction: Example

e_0: $(l_4, (\mathit{true}, \mathit{true}), \{l_4 \mapsto 0\})$

e_1: $(l_{11}, (\mathit{true}, \neg(x_0 < 2)), \{l_4 \mapsto 0\})$

e_2: $(l_{12}, (\mathit{true}, \neg(x_0 < 2)), \{l_4 \mapsto 0\})$

e_3: $(l_5, (\mathit{true}, x_0 < 2), \{l_4 \mapsto 0\})$

e_4: $(l_6, (\mathit{true}, x_0 < 2 \wedge x_1 = x_0 + 1), \{l_4 \mapsto 0\})$

e_5: $(l_7, (\mathit{true}, \wedge x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1), \{l_4 \mapsto 0\})$

e_6: $(l_8, (\mathit{true}, x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(x_1 = y_1)), \{l_4 \mapsto 0\})$

e_7: $(l_{12}, (\mathit{true}, x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(x_1 = y_1)), \{l_4 \mapsto 0\})$

e_8: $(l_4, (\mathit{true}, x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(\neg(x_1 = y_1))), \{l_4 \mapsto 1\})$

e_9: $(l_{11}, (\mathit{true}, x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(\neg(x_1 = y_1)) \wedge \neg(x_1 < 2)), \{l_4 \mapsto 1\})$

e_{10}: $(l_{12}, (\mathit{true}, x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(\neg(x_1 = y_1)) \wedge \neg(x_1 < 2)), \{l_4 \mapsto 1\})$

e_{11}: $(l_5, (\mathit{true}, x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(\neg(x_1 = y_1)) \wedge x_1 < 2), \{l_4 \mapsto 1\})$

e_{12}: $(l_6, (\mathit{true}, x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(\neg(x_1 = y_1)) \wedge x_1 < 2 \wedge x_2 = x_1 + 1), \{l_4 \mapsto 1\})$

e_{13}: $(l_7, (\mathit{true}, x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(\neg(x_1 = y_1)) \wedge x_1 < 2 \wedge x_2 = x_1 + 1 \wedge y_2 = y_1 + 1), \{l_4 \mapsto 1\})$

e_{14}: $(l_8, (\mathit{true}, x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(\neg(x_1 = y_1)) \wedge x < 2 \wedge x_2 = x_1 + 1 \wedge y_2 = y_1 + 1 \wedge \neg(x_2 = y_2)), \{l_4 \mapsto 1\})$

e_{15}: $(l_{12}, (\mathit{true}, x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(\neg(x_1 = y_1)) \wedge x < 2 \wedge x_2 = x_1 + 1 \wedge y_2 = y_1 + 1 \wedge \neg(x_2 = y_2)), \{l_4 \mapsto 1\})$

e_{16}: $(l_4, (\mathit{true}, x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge \neg(\neg(x_1 = y_1)) \wedge x < 2 \wedge x_2 = x_1 + 1 \wedge y_2 = y_1 + 1 \wedge \neg(\neg(x_2 = y_2))), \{l_4 \mapsto 2\})$

Insights

- BMC naturally follows by increasing block size to whole (bounded) program

Insights

- BMC naturally follows by increasing block size to whole (bounded) program
- Difference between predicate abstraction and IMPACT:
  - BDDs vs. SMT-based formulas: costly abstractions vs. costly coverage checks
  - Recompute ARG vs. rechecking coverage
  - We know that only these differences are relevant!
  - Predicate abstraction pays for creating more general abstract model
  - IMPACT is lazier but this can lead to many refinements $\rightarrow$ forced covering or large blocks help

Evaluation: Usefulness of Framework

- 4 existing approaches successfully integrated
- Ongoing projects for integration of further approaches
- Interesting insights learned about these approaches
- High configurability allows new combinations and hybrid approaches
- Already used as base for other successful research projects

Evaluation: Usefulness of Implementation

- Used in other research projects

- Used as part of many SV-COMP submissions, 48 medals

- Also competitive stand-alone

- Awarded Gödel medal by Kurt Gödel Society

Comparison with SV-COMP'17 Verifiers

- 5 594 verification tasks from SV-COMP'17 (only reachability, without recursion and concurrency)
- 15 min time limit per task (CPU time)
- 15 GB memory limit
- Measured with BenchExec
- Comparison of
  - 4 configurations of CPAchecker with Predicate CPA: BMC, *k*-induction, Impact, predicate abstraction
  - 16 participants of SV-COMP'17

Comparison with SV-COMP'17 Verifiers: Results

Number of correctly solved tasks:

- Each configuration of Predicate CPA beats other tools with same approach
- Only 3 tools beat Predicate CPA with *k*-induction:
  - SMACK: guesses results
  - CPA-BAM-BnB, CPA-Seq: based on Predicate CPA as well

Comparison with SV-COMP'17 Verifiers: Results

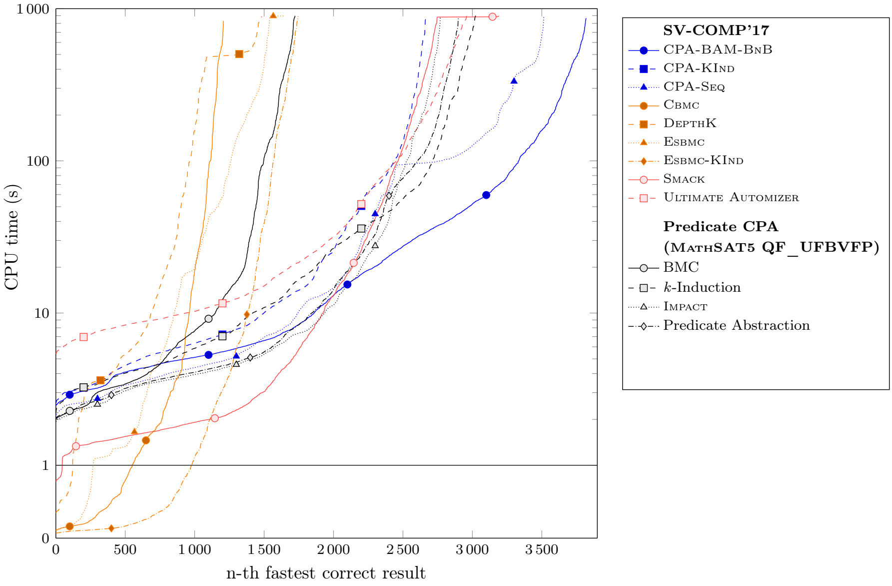
Number of correctly solved tasks:

- Each configuration of Predicate CPA beats other tools with same approach
- Only 3 tools beat Predicate CPA with *k*-induction:
  - SMACK: guesses results
  - CPA-BAM-BNB, CPA-SEQ: based on Predicate CPA as well

Number of wrong results:

- Comparable with other tools
- No wrong proofs (sound)

Comparison with SV-COMP'17 Verifiers



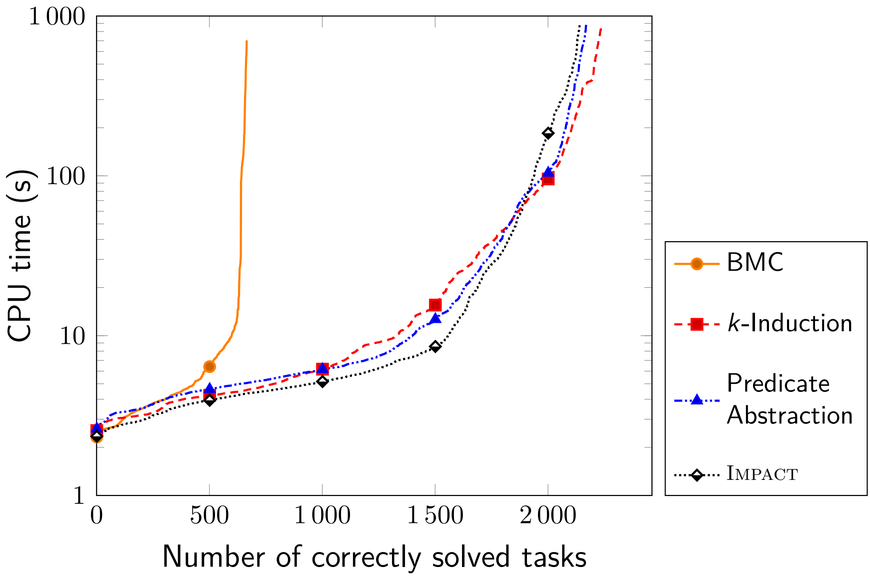
Evaluation: Enabling Experimental Studies

- Comparison of algorithms
  across different program categories
  [VSTTE'16, JAR]
- SMT solvers for various theories and algorithms

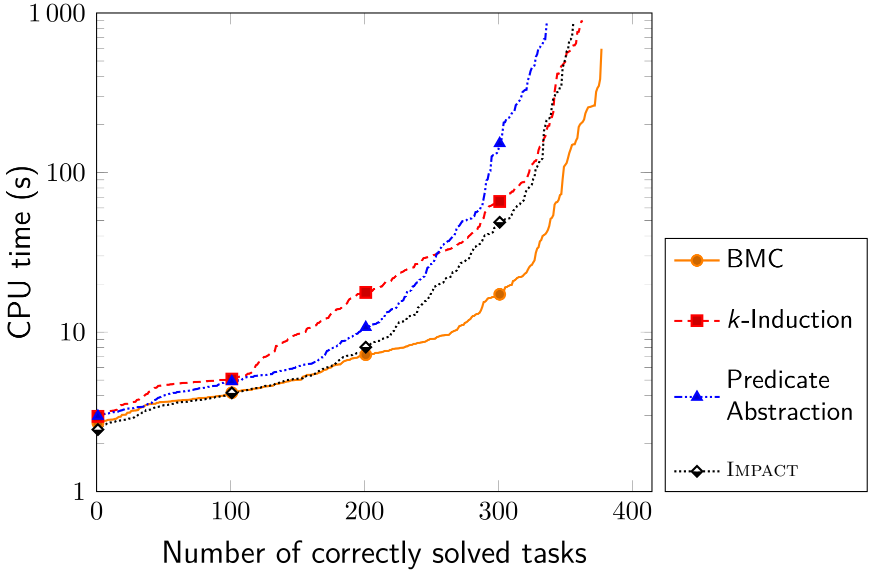
Experimental Comparison of Algorithms

- 5 287 verification tasks from SV-COMP'17
- 15 min time limit per task (CPU time)
- 15 GB memory limit
- Measured with BenchExec

All 3 913 bug-free tasks



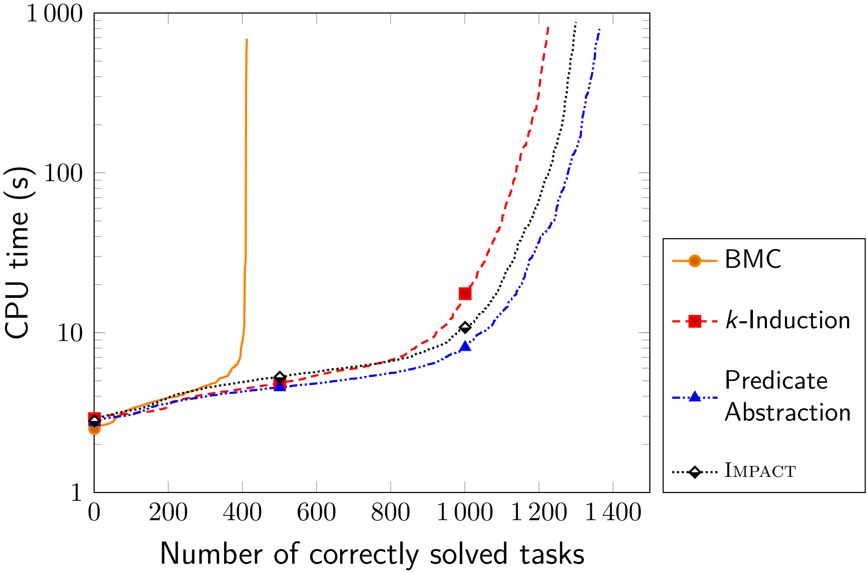
All 1 374 tasks with known bugs



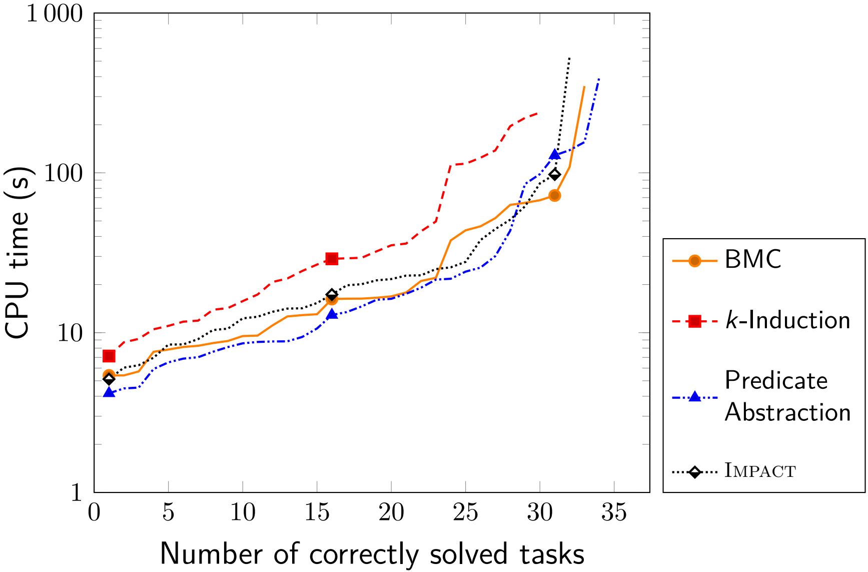
Category *Device Drivers*

- Several thousands LOC per task
- Complex structures
- Pointer arithmetics

Category *Device Drivers*: 2 440 bug-free tasks



Category *Device Drivers*: 355 tasks with known bugs



Category *Event Condition Action Systems (ECA)*

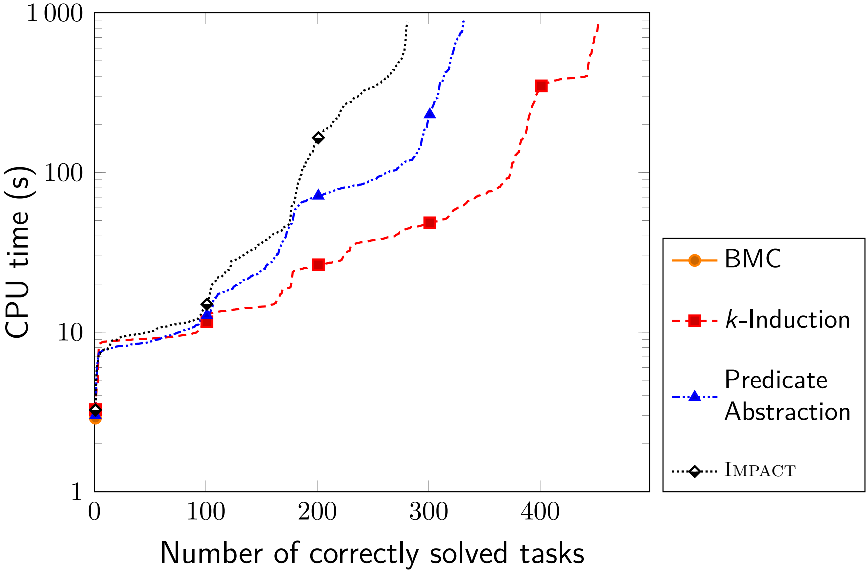
- Several thousand LOC per task
- Auto-generated
- Only integer variables
- Linear and non-linear arithmetics
- Complex and dense control structure

Category *Event Condition Action Systems (ECA)*

- Several thousand LOC per task
- Auto-generated
- Only integer variables
- Linear and non-linear arithmetics
- Complex and dense control structure

```
if (((a24==3) && (((a18==10) && ((input == 6)
    && ((115 < a3) && (306 >= a3))))
    && (a15==4)))) {
  a3 = (((a3 * 5) + -583604) * 1);
  a24 = 0;
  a18 = 8;
  return -1;
}
```

Category *ECA*: 738 bug-free tasks



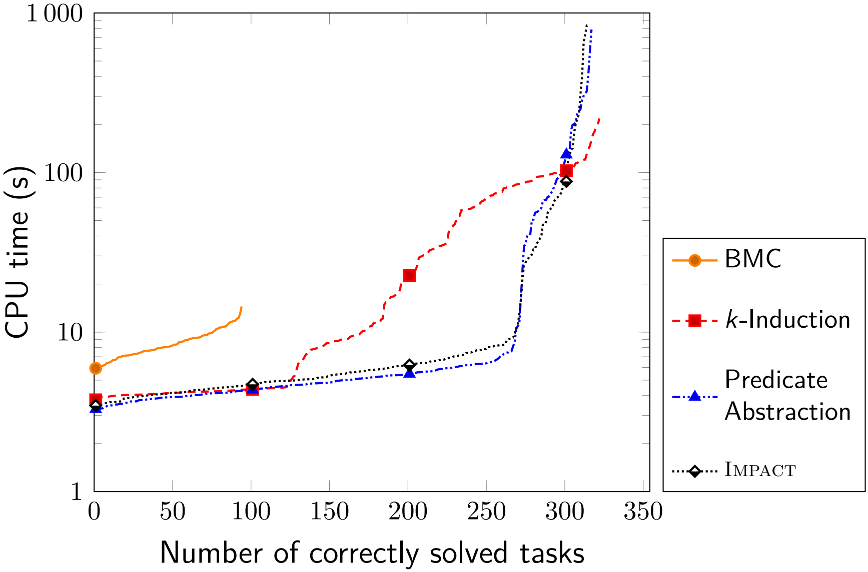
Category *ECA*: 411 tasks with known bugs

- Only BMC and *k*-Induction solve 1 task (the same one for both)
- IMPACT and Predicate Abstraction solve none

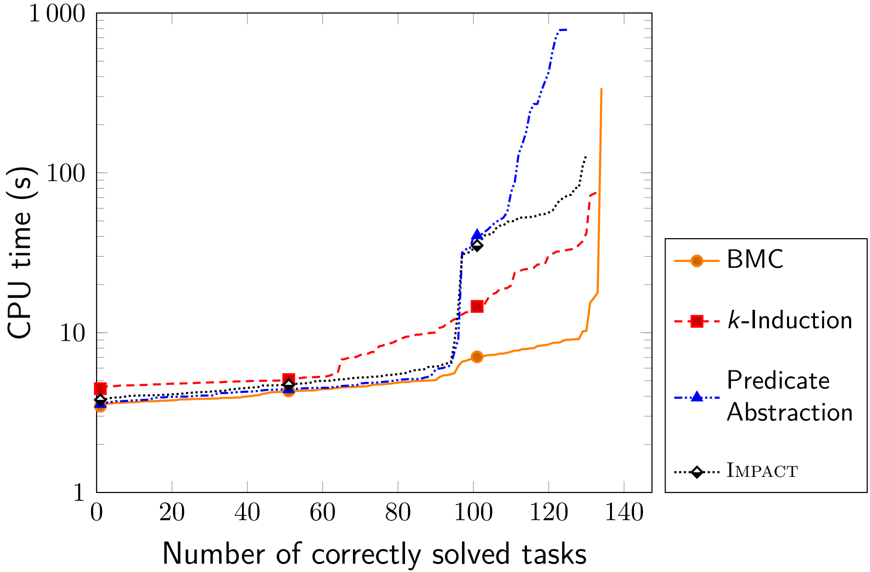
Category *Product Lines*

- Several hundred LOC
- Mostly integer variables, some structs
- Mostly simple linear arithmetics
- Lots of property-independent code

Category *Product Lines*: 332 bug-free tasks



Category *Product Lines*: 265 tasks with known bugs



Experimental Comparison of Algorithms: Summary

We reconfirm that

- BMC is a good bug hunter
- *k*-Induction is a heavy-weight proof technique:
  effective, but costly
- CEGAR makes abstraction techniques
  (Predicate Abstraction, IMPACT) scalable
- IMPACT is lazy:
  explores the state space and finds bugs quicker
- Predicate Abstraction is eager:
  prunes irrelevant parts and finds proofs quicker

SMT Study: Motivation

Now, which do you think is better, i.e., solves more tasks?

k-Induction

Predicate Abstraction

SMT Study: Motivation

Now, which do you think is better, i.e., solves more tasks?

(A)	(B)
k-Induction solves 29 % more tasks	Predicate Abstraction solves 3 % more tasks

SMT Study: Motivation

Now, which do you think is better, i.e., solves more tasks?

(A)	(B)
k-Induction solves 29 % more tasks	Predicate Abstraction solves 3 % more tasks
Z3 with bitprecise arithmetic	MathSAT5 with linear arithmetic

Depending on configuration, either (A) or (B) can be true!

Technical details (e.g., choice of SMT theory)
influence evaluation of algorithms

Comparison of SMT Solvers and Theories

- Which SMT solver should we use in a verifier?
- Which formula encoding?
- Which of these should we use for benchmarks in papers?

Comparison of SMT Solvers and Theories

- Which SMT solver should we use in a verifier?
- Which formula encoding?
- Which of these should we use for benchmarks in papers?

- Large study made possible by our framework
- Produced some interesting insights
- Resulted in change of default configuration of CPACHECKER

Comparison of SMT Solvers and Theories

- Which SMT solver should we use in a verifier?
- Which formula encoding?
- Which of these should we use for benchmarks in papers?

- Large study made possible by our framework
- Produced some interesting insights
- Resulted in change of default configuration of CPAchecker

- Comparison using CPAchecker and Predicate CPA
- 5 594 verification tasks from SV-COMP'17
- 15 min time limit (CPU time), 15 GB memory limit
- Measured with BenchExec

SMT Study: 120 Configurations

BMC | *k*-Induction | Impact | Pred. Abs

$\times$

MathSAT5 | Princess | SMTInterpol | Z3

$\times$

Bitprecise | Linear | Linear unsound

$\times$

with Quantifiers | Quantifier-free

$\times$

Arrays | UFs

Point of View: SMT Solvers

- Princess is never competitive
- Interpolation in Z3 is unmaintained since 2015
- Bitvector interpolation in Z3 produces up to 24 % crashes
- MathSAT5 has known interpolation problem for bitvectors, but problem occurs rarely

Point of View: Theories and Encodings

- Unsound linear encoding always the easiest (as expected)

Point of View: Theories and Encodings

- Unsound linear encoding always the easiest (as expected)
- Correctness as expected:
  BV $>$ sound LIRA $>$ unsound LIRA

Point of View: Theories and Encodings

- Unsound linear encoding always the easiest (as expected)
- Correctness as expected:
  BV $>$ sound LIRA $>$ unsound LIRA
- Effectivity for $\mathrm{Z3}$ as expected:
  BV $<$ sound LIRA $<$ unsound LIRA

Point of View: Theories and Encodings

- Unsound linear encoding always the easiest (as expected)
- Correctness as expected:
  BV > sound LIRA > unsound LIRA
- Effectivity for Z3 as expected:
  BV < sound LIRA < unsound LIRA
- Effectivity for MathSAT5:
  sound LIRA < BV ≈ unsound LIRA
  (but BV needs more CPU time)

⇒ MathSAT5 is really good with bitvectors.

Point of View: Theories and Encodings

- Unsound linear encoding always the easiest (as expected)
- Correctness as expected:
  BV $>$ sound LIRA $>$ unsound LIRA
- Effectivity for Z3 as expected:
  BV $<$ sound LIRA $<$ unsound LIRA
- Effectivity for MathSAT5:
  sound LIRA $<$ BV $\approx$ unsound LIRA
  (but BV needs more CPU time)
- Effectivity for SMTInterpol:
  sound LIRA $\ll$ unsound LIRA

$\Rightarrow$ MathSAT5 is really good with bitvectors.

Point of View: Theories and Encodings

- Unsound linear encoding always the easiest (as expected)
- Correctness as expected:
  BV $>$ sound LIRA $>$ unsound LIRA
- Effectivity for Z3 as expected:
  BV $<$ sound LIRA $<$ unsound LIRA
- Effectivity for MathSAT5:
  sound LIRA $<$ BV $\approx$ unsound LIRA
  (but BV needs more CPU time)
- Effectivity for SMTInterpol:
  sound LIRA $\ll$ unsound LIRA

$\Rightarrow$ MathSAT5 is really good with bitvectors.

$\Rightarrow$ Sound LIRA encoding rarely makes sense.

Point of View: Algorithms

- Mostly, the best configurations of MathSAT5, SMTInterpol, and Z3 are close for each algorithm
  - Gives confidence for valid comparison of algorithm
  - But outlier exists:
    Z3 is worse than others for predicate abstraction

Point of View: Algorithms

- Mostly, the best configurations of MathSAT5, SMTInterpol, and Z3 are close for each algorithm
  - Gives confidence for valid comparison of algorithm
  - But outlier exists:
    Z3 is worse than others for predicate abstraction
- Predicate abstraction and Impact suffer most from disjunctions of sound LIRA encoding.

Point of View: Arrays and Quantifiers

- Little difference with/without arrays/quantifiers

$\Rightarrow$ Arrays don't hurt
(though this might change
once more complex array predicates are used)

Point of View: Arrays and Quantifiers

- Little difference with/without arrays/quantifiers

$\Rightarrow$ Arrays don't hurt
(though this might change
once more complex array predicates are used)

- But quantifiers restrict solver choice
(PRINCESS and Z3)

SMT Study: Final Conclusions

- Choice of theories, solver, and encoding details affects comparisons of algorithms!
- For now:
  use MathSAT5 with bitvectors and arrays if possible
  - Possible problems for users: license, native binary
  - Next-best choice:
    SMTInterpol with unsound linear arithmetic
  - No improvement of situation in sight