Software Verification

An Overview of the State of the Art

Dirk Beyer

LMU Munich, Germany

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

CPA✓

SoSy-Lab Software Systems

Outline: Three Parts

- Overview
- SMT-Based Automatic Software Verification
- Cooperative Verification

Some Historical Landmarks

- **70 years ago: Assertions and Proof Decomposition**, Alan Turing, 1949 [28]
  "In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."

Some Historical Landmarks

- 70 years ago: Assertions and Proof Decomposition
- **45 years ago: Data-Flow Analysis and Abstract States**,
  Gary Kildall, POPL 1973 [26]
  "A Unified Approach to Global Program Optimization"
  Defines algorithm, meet operation, lattice, etc.
  Extended and made popular for program analysis by
  F. Nielson, H. R. Nielson, C. Hankin, P. Cousot, R. Cousot

Some Historical Landmarks

- 70 years ago: Assertions and Proof Decomposition
- 45 years ago: Data-Flow Analysis and Abstract States
- **40 years ago: LTL and Model Checking**,
  Pnueli, Clarke, Emerson, Sifakis, 1981 [17]
  Specification languages, modeling languages, algorithms, theory, tools
  LTL, CTL, automata, Kripke structures, model checking,
  $\rightarrow$ software model checking

"Handbook of Model Checking", Springer, 2018 [17]

Some Historical Landmarks

- 70 years ago: Assertions and Proof Decomposition
- 45 years ago: Data-Flow Analysis and Abstract States
- 40 years ago: LTL and Model Checking
- **20 years ago: Predicate Abstraction**,
  Graf, Saïdi, 1997 [18]
  Enabling idea to project software to
  a (smaller) finite state space
  Accelerated development of tools:
  BLAST, SLAM, SATABS ("1st generation")

Some Historical Landmarks

- 70 years ago: Assertions and Proof Decomposition
- 45 years ago: Data-Flow Analysis and Abstract States
- 40 years ago: LTL and Model Checking
- **15 years ago: Satisfiability Modulo Theory**
  Enormous breakthrough, many tools, ...
  See Cesare Tinelli's tutorial yesterday

Some Historical Landmarks

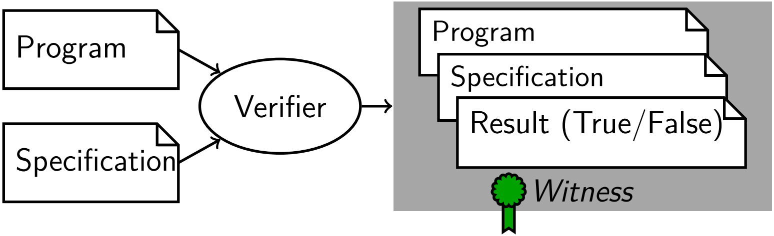
- 70 years ago: Assertions and Proof Decomposition
- 45 years ago: Data-Flow Analysis and Abstract States
- 40 years ago: LTL and Model Checking
- 15 years ago: Satisfiability Modulo Theory
- **today:** From **lack** of tools to **abundance** of tools
  Problem: missing standard interfaces
  Solution: competitions to establish standards
  (input, exchange, comparability, reproducibility)

Competitions in Software Verification and Testing

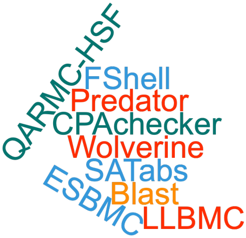
- RERS: off-site, tools, free-style [21]
- SV-COMP: off-site, automatic tools, controlled [1]
- Test-Comp: off-site, automatic tools, controlled [3]
- VerifyThis: on-site, interactive, teams [22]

(alphabetic order)

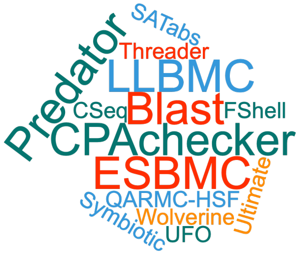
Automatic Software Verification



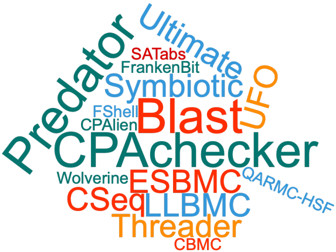
SV-COMP (Automatic Tools 2012)



SV-COMP (Automatic Tools 2013, cumulative)

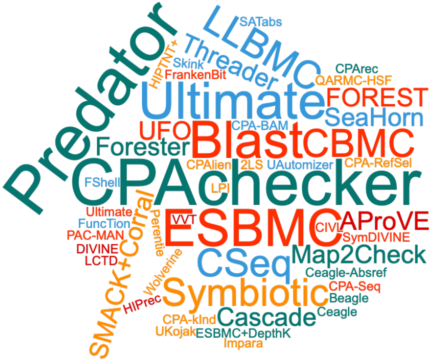


SV-COMP (Automatic Tools 2014, cumulative)

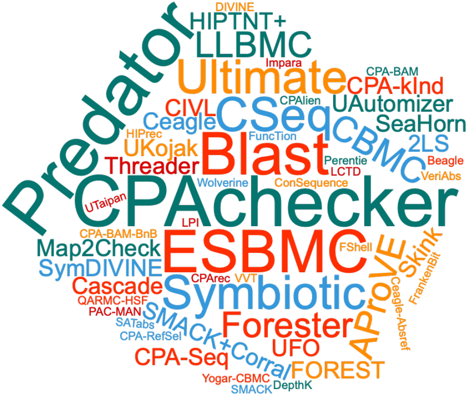


SV-COMP (Automatic Tools 2015, cumulative)

SV-COMP (Automatic Tools 2016, cumulative)

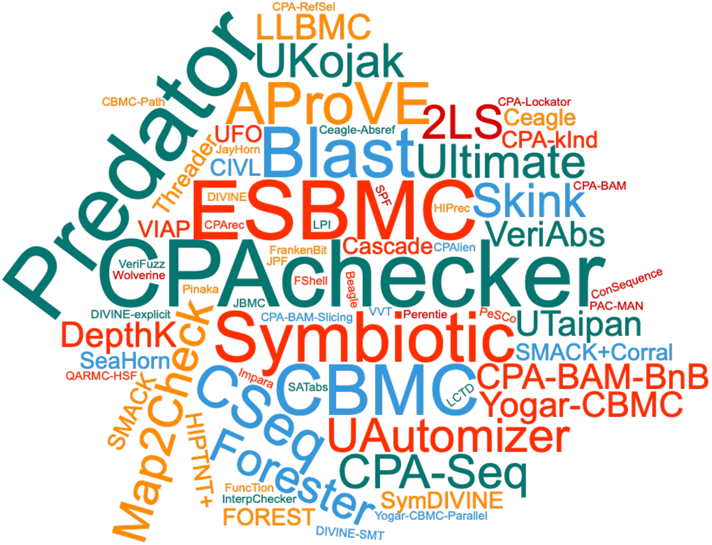


SV-COMP (Automatic Tools 2017, cumulative)



SV-COMP (Automatic Tools 2018, cumulative)

SV-COMP (Automatic Tools 2019, cumulative)

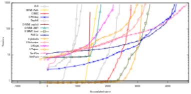


What is the best verifier?

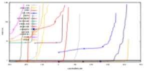
- Many different kinds of programs seem to require many different good tools with different strengths

SV-COMP (Automatic Tools)

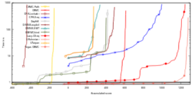
ReachSafety
1. **VeriAbs**
2. **CPA-Seq**
3. **PeSCo**



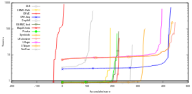
MemSafety
1. **Symbiotic**
2. **PredatorHP**
3. **CPA-Seq**



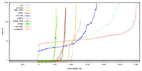
ConcurrencySafety
1. **Yogar-CBMC**
2. **Lazy-CSeq**
3. **CPA-Seq**



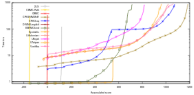
NoOverflows
1. **UAutomizer**
2. **UTaipan**
3. **CPA-Seq**



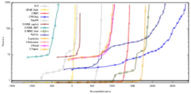
Termination
1. **UAutomizer**
2. **AProVE**
3. **CPA-Seq**



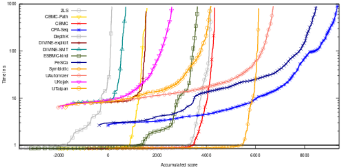
SoftwareSystems
1. **CPA-BAM-BnB**
2. **CPA-Seq**
3. **VeriAbs**



FalsificationOverall
1. **CPA-Seq**
2. **PeSCo**
3. **ESBMC-kind**



Overall
1. **CPA-Seq**
2. **PeSCo**
3. **UAutomizer**



https://sv-comp.sosy-lab.org/2019/results

Which techniques are used?

Participant	CEGAR	Predicate Abstraction	Symbolic Execution	Bounded Model Checking	k-Induction	Property-Directed Reach.	Explicit-Value Analysis	Numeric. Interval Analysis	Shape Analysis	Separation Logic	Bit-Precise Analysis	ARG-Based Analysis	Lazy Abstraction	Interpolation	Automata-Based Analysis	Concurrency Support	Ranking Functions	Evolutionary Algorithms
2LS				✓	✓			✓			✓						✓	
AProVE			✓				✓	✓		✓	✓						✓	
CBMC				✓							✓					✓		
CBMC-Path				✓							✓					✓		
CPA-BAM-BnB	✓	✓					✓				✓	✓	✓	✓				
CPA-Lockator	✓	✓					✓				✓	✓	✓	✓		✓		
CPA-Seq	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓	
DepthK				✓	✓						✓					✓		
DIVINE-explicit							✓				✓					✓		
DIVINE-SMT							✓				✓					✓		
ESBMC-kind				✓	✓						✓					✓		
JayHorn	✓	✓				✓		✓					✓	✓				
JBMC				✓							✓					✓		
JPF				✓			✓	✓			✓					✓		
Lazy-CSeq				✓							✓					✓		
Map2Check				✓							✓							
PeSCo	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓	
Pinaka			✓	✓							✓							
PredatorHP									✓									
Skink	✓						✓							✓	✓			
Smack	✓			✓		✓					✓		✓			✓		
SPF			✓						✓							✓		
Symbiotic			✓					✓			✓							
UAutomizer	✓	✓									✓		✓	✓	✓		✓	
UKojak	✓	✓									✓		✓	✓				

Competition Report [2]
https://doi.org/10.1007/978-3-030-17502-3_9

Algorithms

17	Bounded Model Checking
13	CEGAR
8	Predicate Abstraction
5	k-Induction
4	Symbolic Execution
3	Automata-Based Analysis
2	Property-Directed Reachability (IC3)

Abstract Domains

24	Bit-Precise Analysis
10	Explicit-Value Analysis
9	Numerical Interval Analysis
4	Shape Analysis
1	Separation Logic

Testing

- Fuzzing (VeriFuzz [16], based on AFL)
- Symbolic execution (KLEE [15])
- Software model checking (CoVeriTest [12], $\rightarrow$ Poster)

Part 2

Unified View on SMT-Based Software Verification

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, ...)
- Property-Directed Reachability (PDR, also known as IC3)
  (CPACHECKER, SEAHORN, VVT, ...)
- Trace Abstraction
  (ULTIMATE AUTOMIZER, CPACHECKER in progress, ...)

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

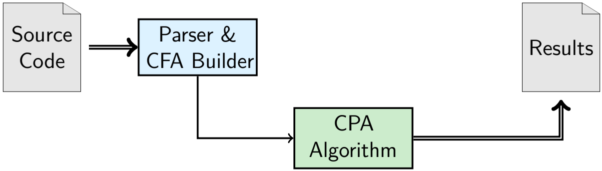
Base: Configurable Program Analysis

Configurable Program Analysis (CPA):

- Beyer, Henzinger, Théoduloz: Proc. CAV 2007, [11]
- One single unifying algorithm for all algorithms based on state-space exploration
- Configurable components: abstract domain, abstract-successor computation, path sensitivity, ...

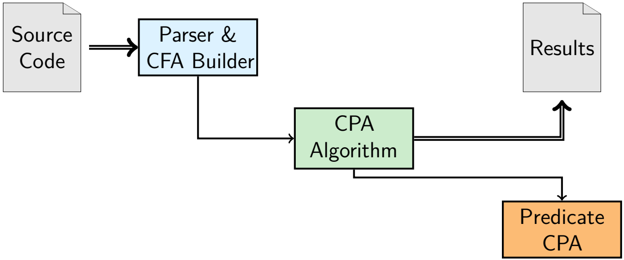
Using the CPA Framework

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



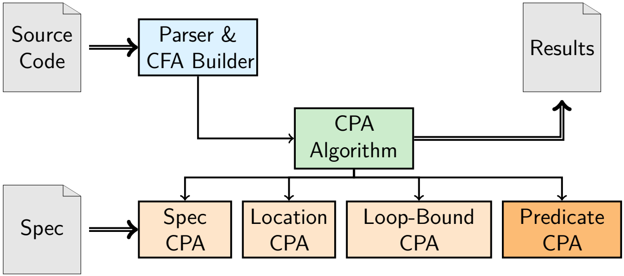
Using the CPA Framework

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



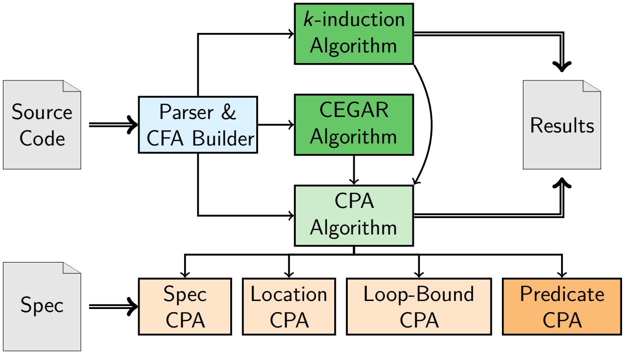
Using the CPA Framework

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

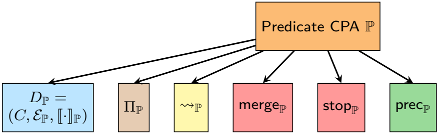


Using the CPA Framework

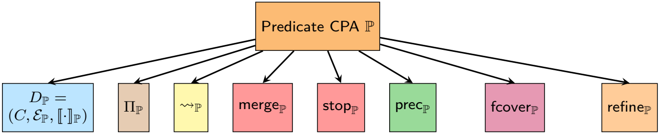
- CPA Algorithm is a configurable reachability analysis for arbitrary abstract domains
- Provide Predicate CPA for our predicate-based abstract domain
- Reuse other CPAs
- Built further algorithms on top that make use of reachability analysis



Predicate CPA



Predicate CPA



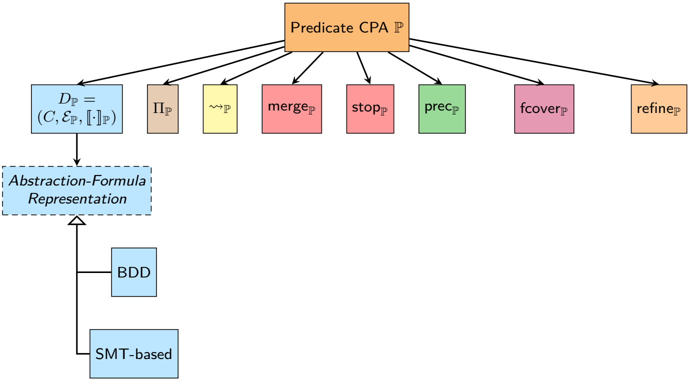
Predicate CPA: Abstract Domain

- 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

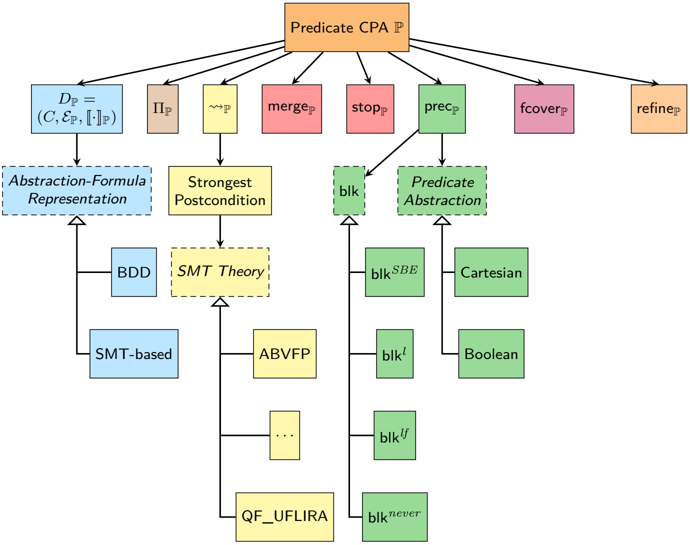
Predicate CPA: CPA Operators

- 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 (ψ', true)

Predicate CPA



Predicate CPA: Refinement

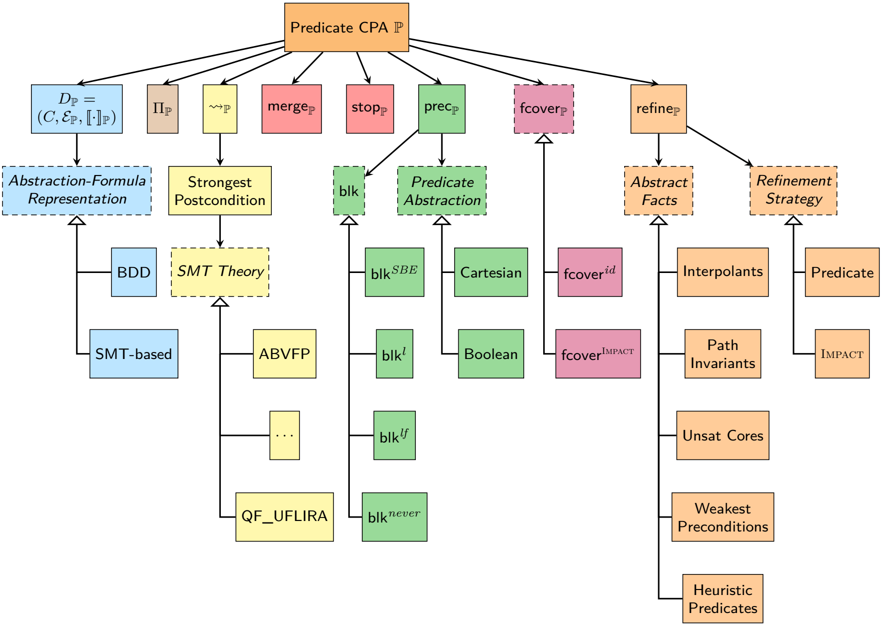
Four steps:

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
  - [18, 20, 23, 19, 24]
  - 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

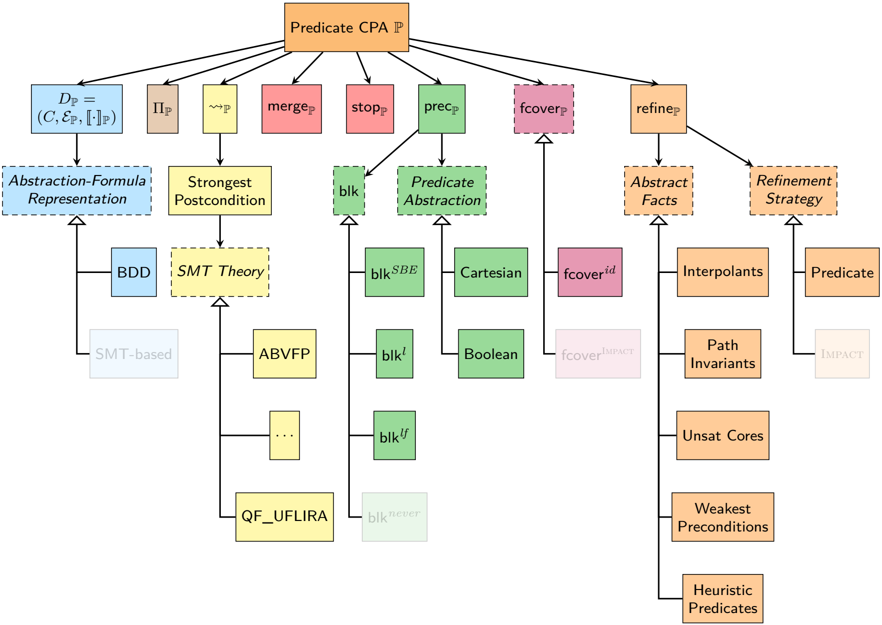
Expressing Predicate Abstraction

- 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:

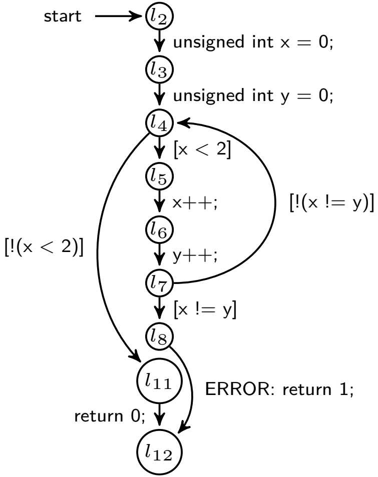
1: **while** $true$ **do**
2: run CPA Algorithm
3: **if** target state found **then**
4: call refine
5: **if** target state reachable **then**
6: **return** $false$
7: **else**
8: **return** $true$

Predicate CPA

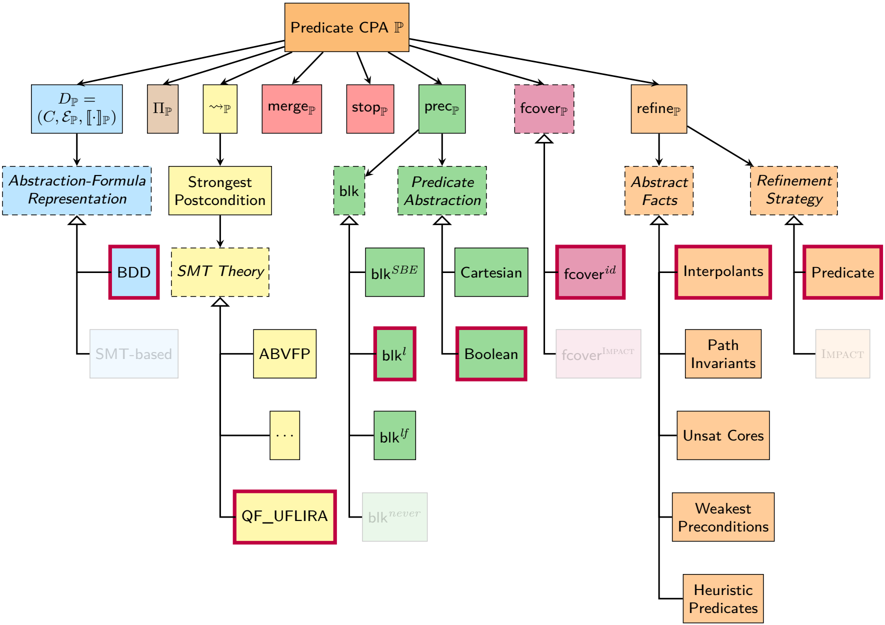


Example Program

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

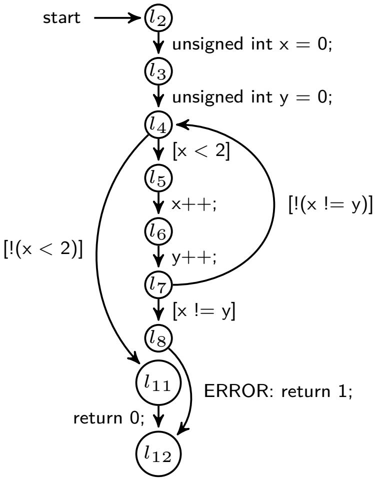


Predicate CPA



Predicate Abstraction: Example

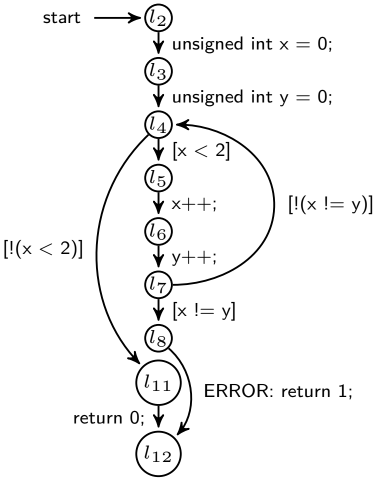
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

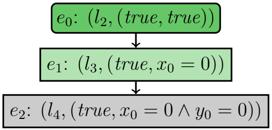


e_0: $(l_2, (true, true))$

Predicate Abstraction: Example

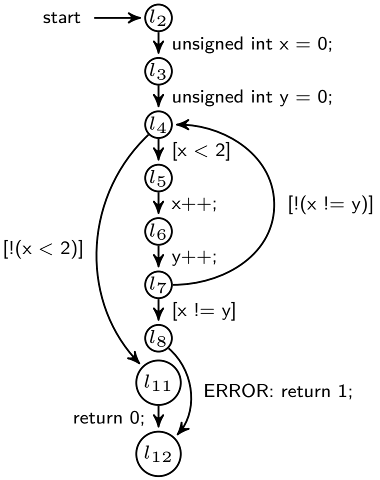
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$





Predicate Abstraction: Example

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



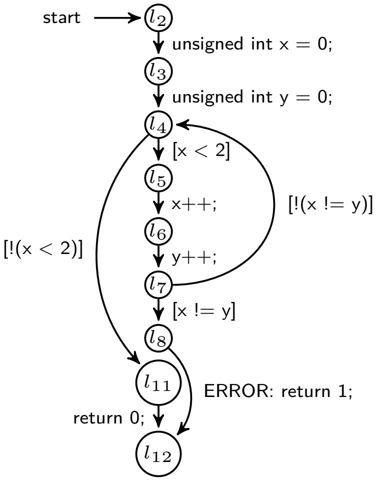
e_0: $(l_2, (true, true))$

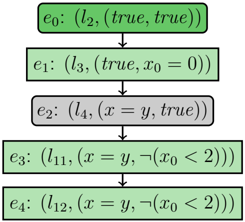
e_1: $(l_3, (true, x_0 = 0))$

e_2: $(l_4, (x = y, true))$

Predicate Abstraction: Example

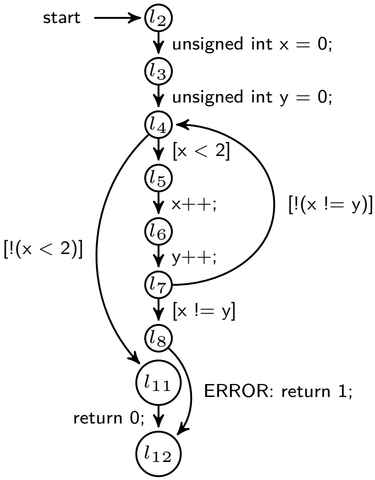
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

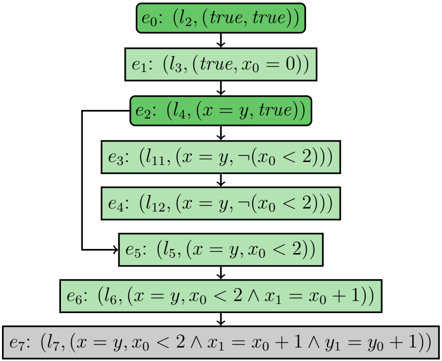




Predicate Abstraction: Example

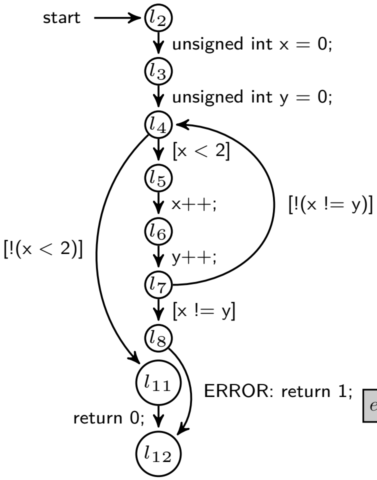
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

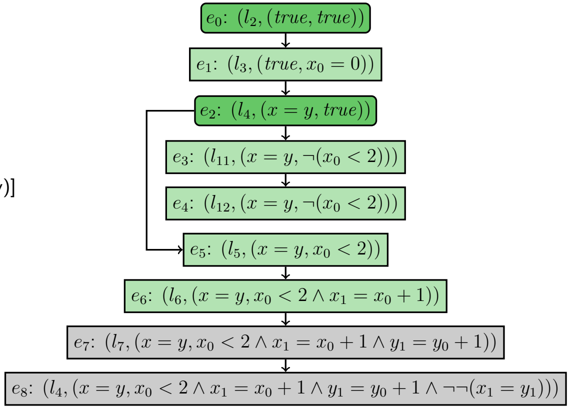




Predicate Abstraction: Example

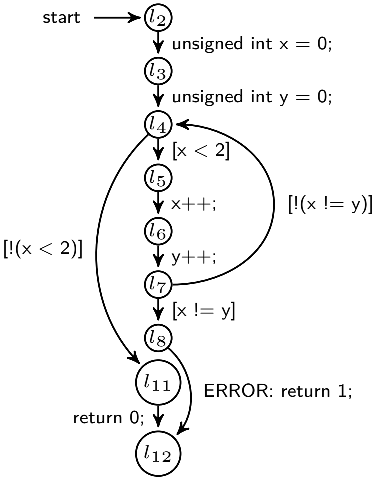
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

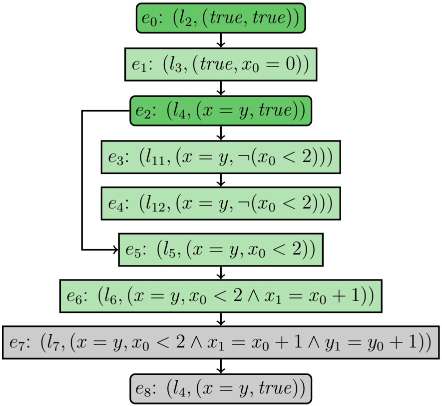




Predicate Abstraction: Example

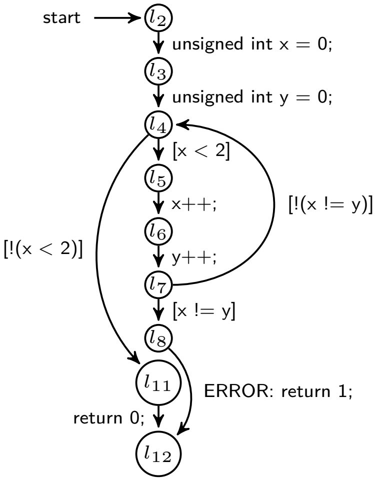
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

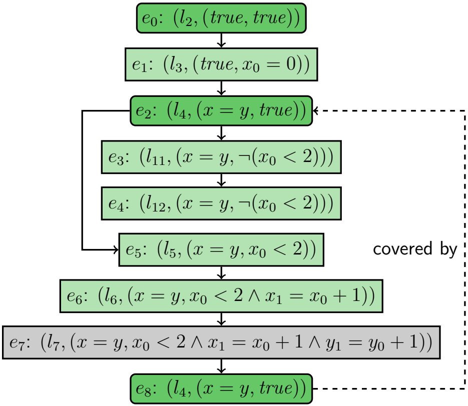




Predicate Abstraction: Example

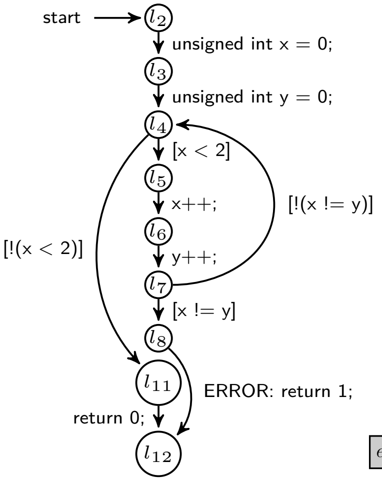
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$

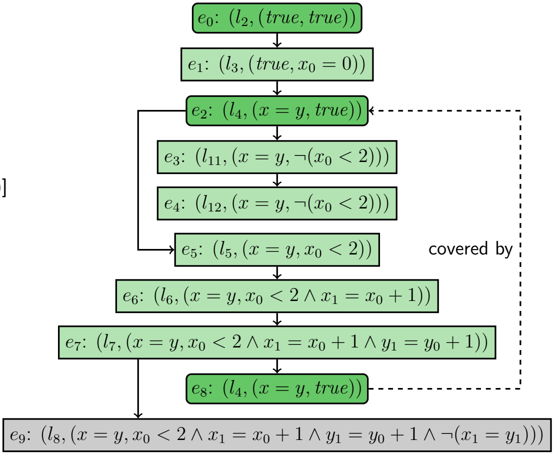




Predicate Abstraction: Example

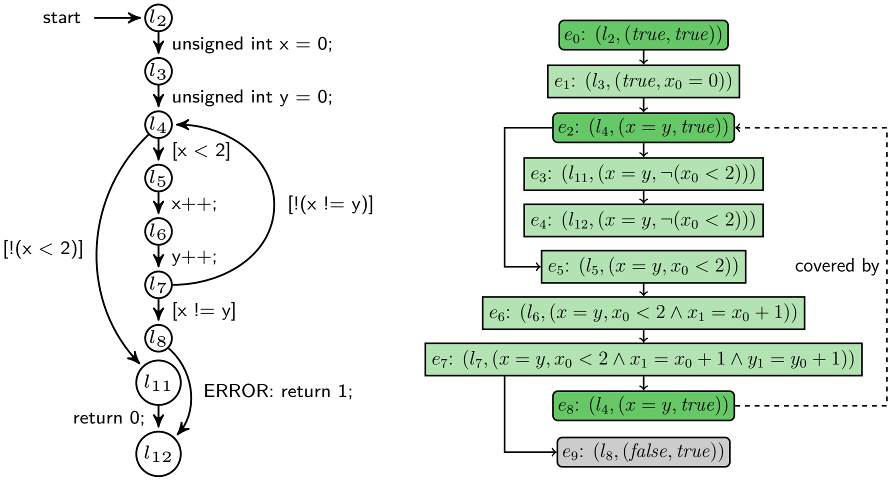
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$





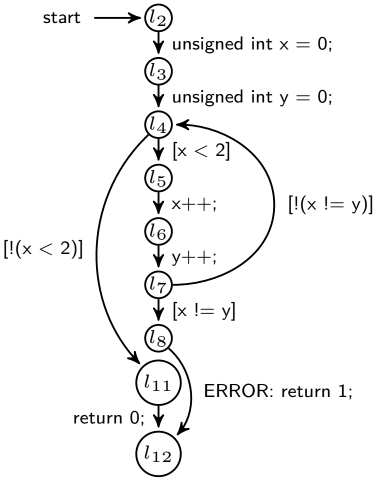
Predicate Abstraction: Example

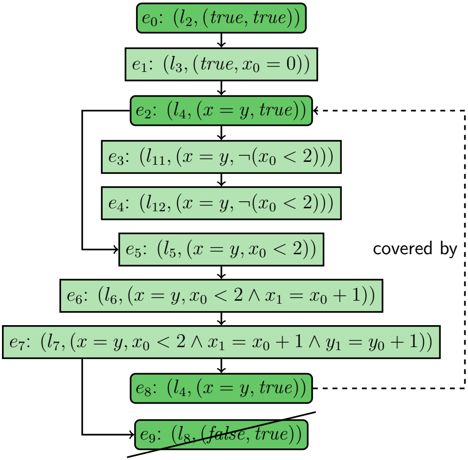
with blk^l, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{false\}$



Predicate Abstraction: Example

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





Impact

- Impact
  - "Lazy Abstraction with Interpolants" Proc. CAV 2006 [27]
  - 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

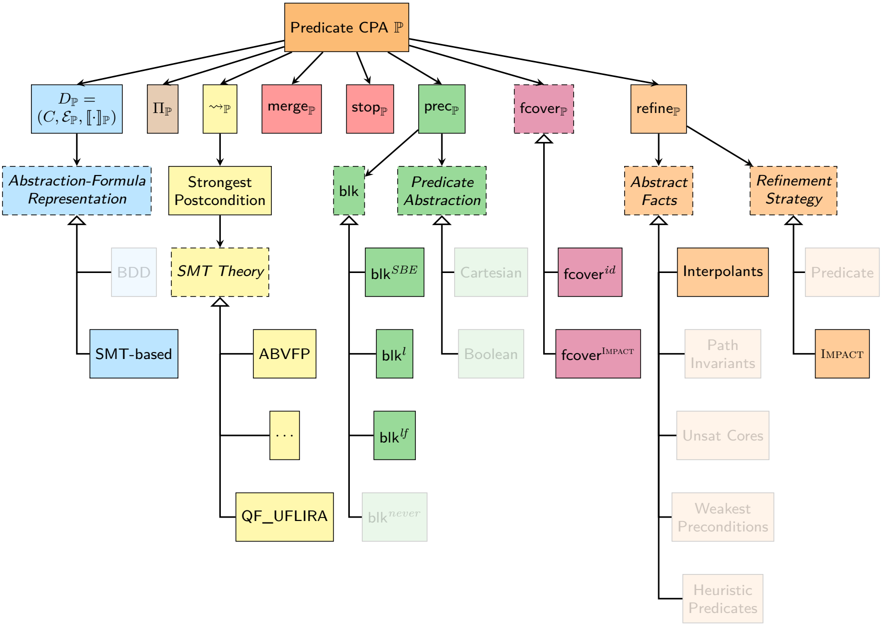
Expressing IMPACT

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

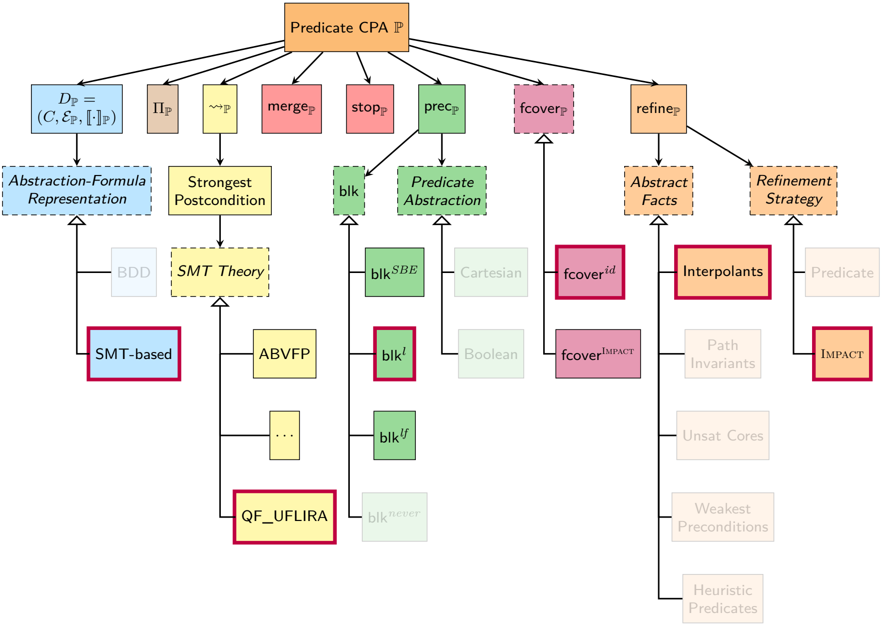
Furthermore:

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

Predicate CPA

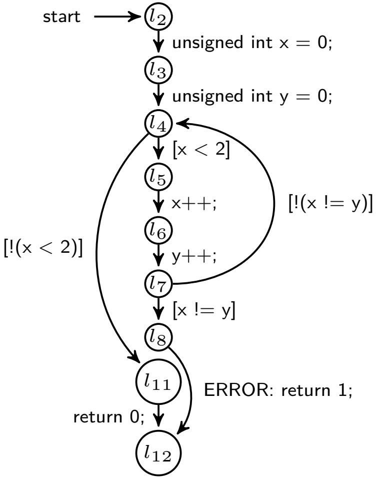


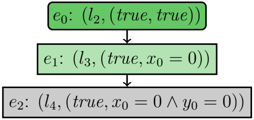
Predicate CPA



IMPACT: Example

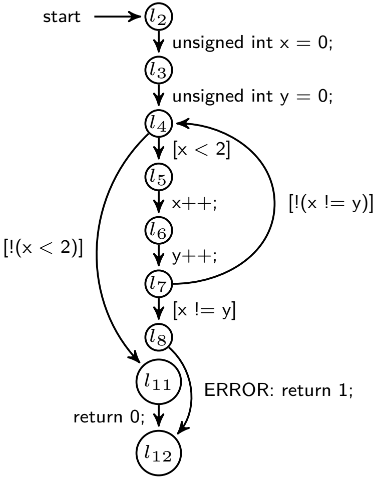
with blk^l

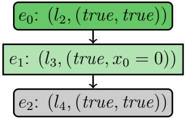




IMPACT: Example

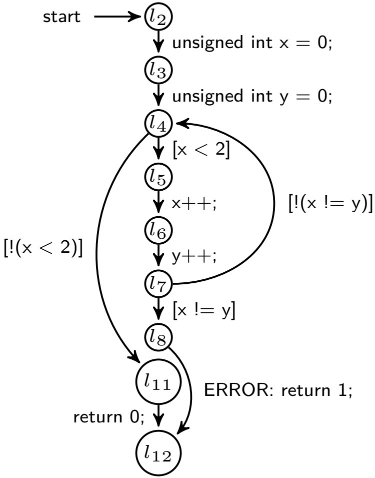
with blk^l

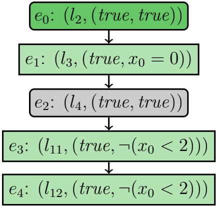




IMPACT: Example

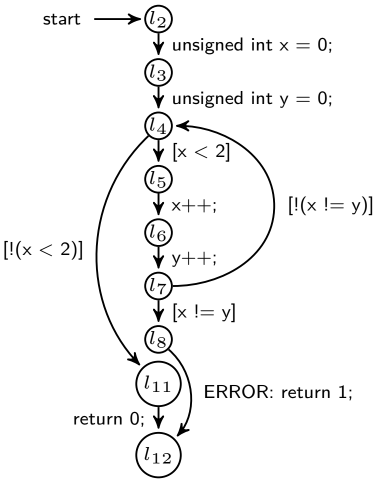
with blk^l

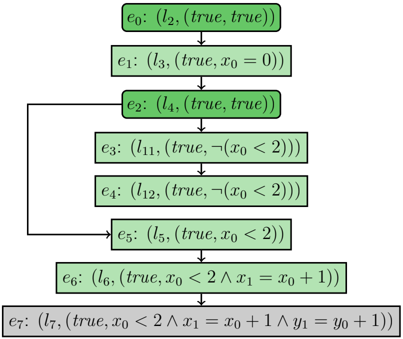




Impact: Example

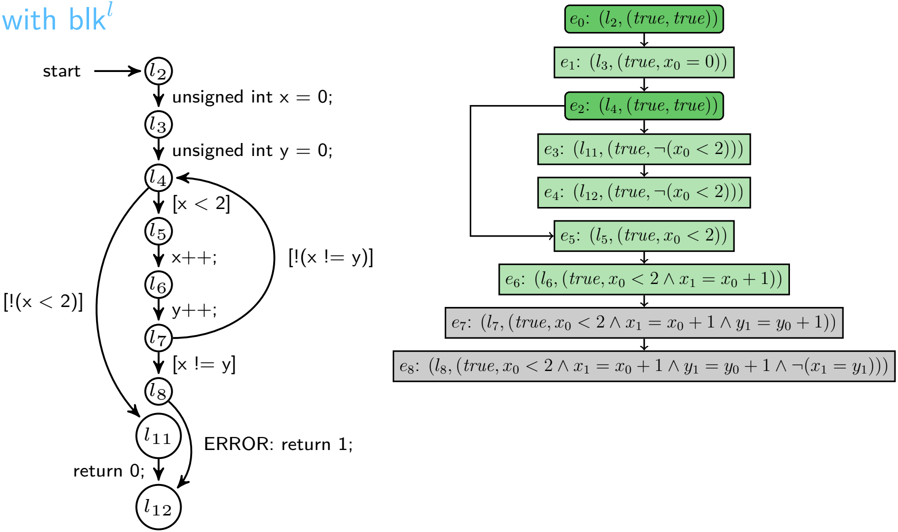
with blk^l





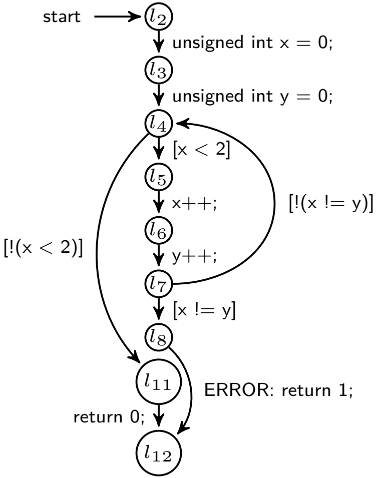
IMPACT: Example

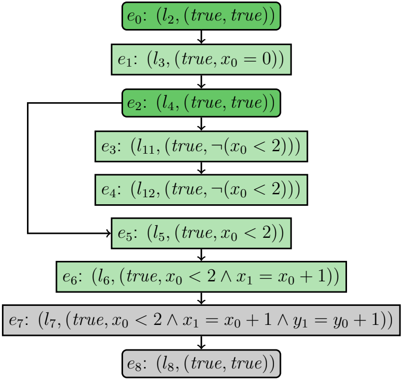
with blk^l



Impact: Example

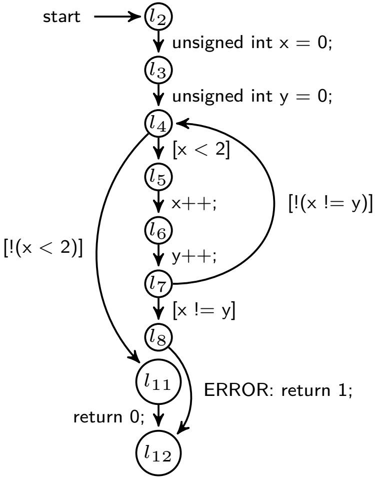
with blk^l

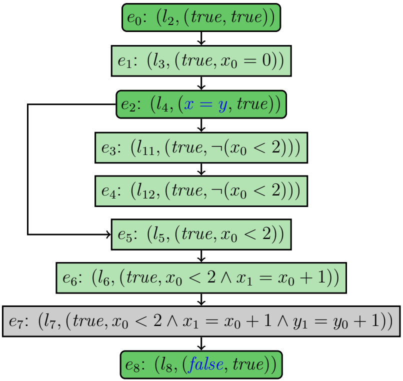




Impact: Example

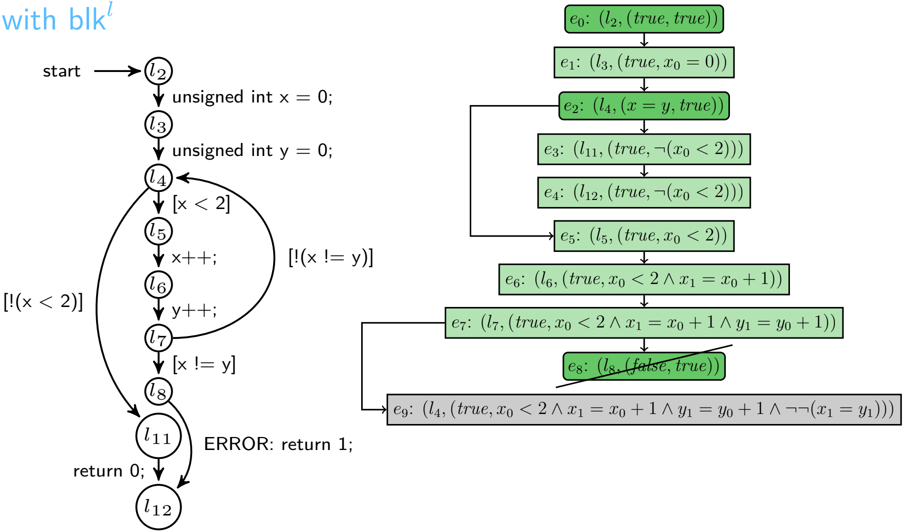
with blk^{l}





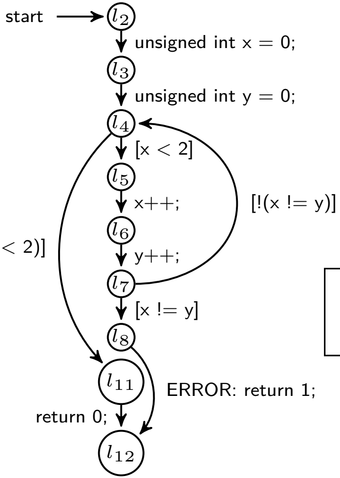
IMPACT: Example

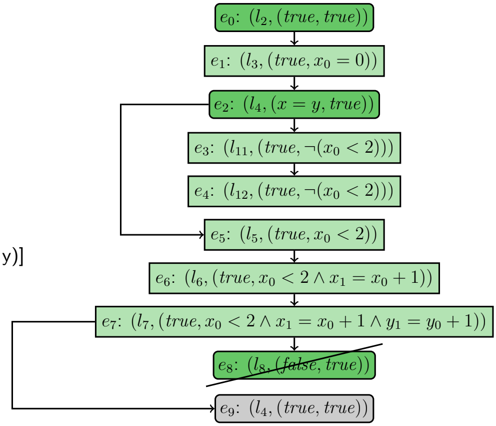
with blk^l



IMPACT: Example

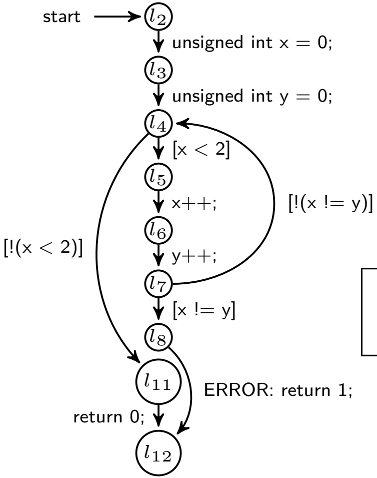
with blk^{l}

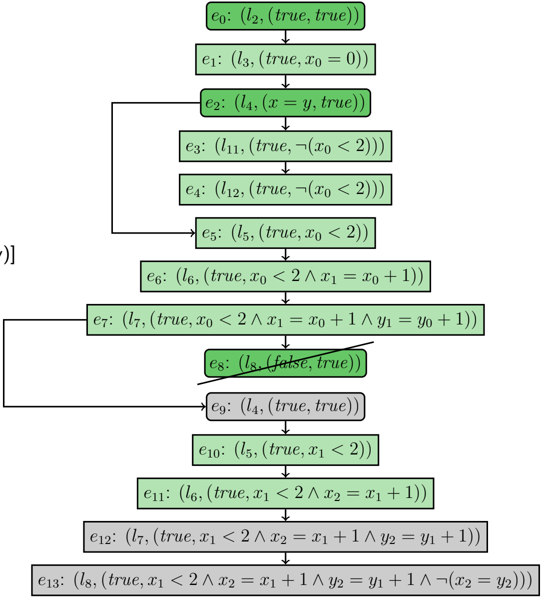




IMPACT: Example

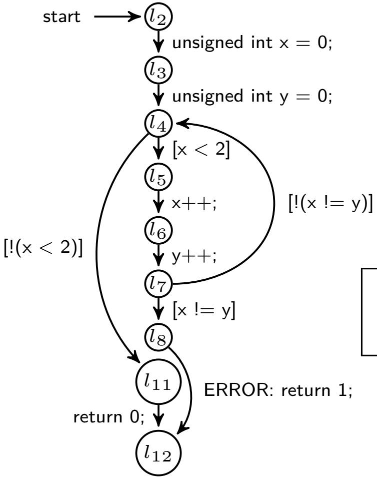
with blk^l

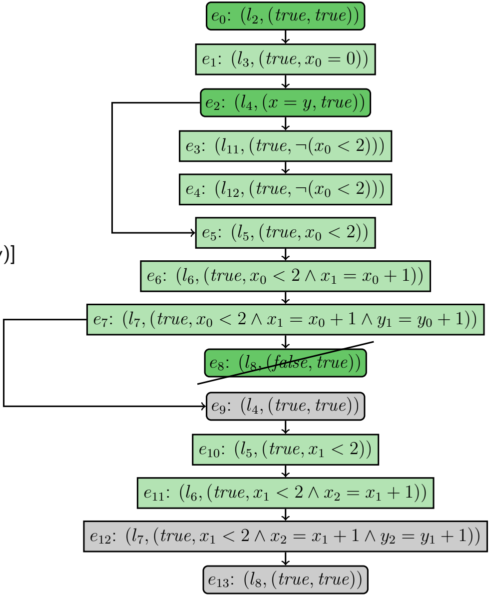




IMPACT: Example

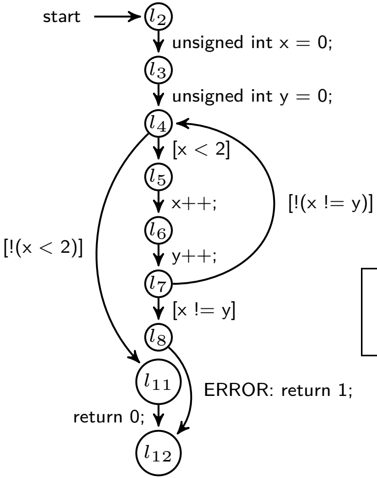
with blk^l

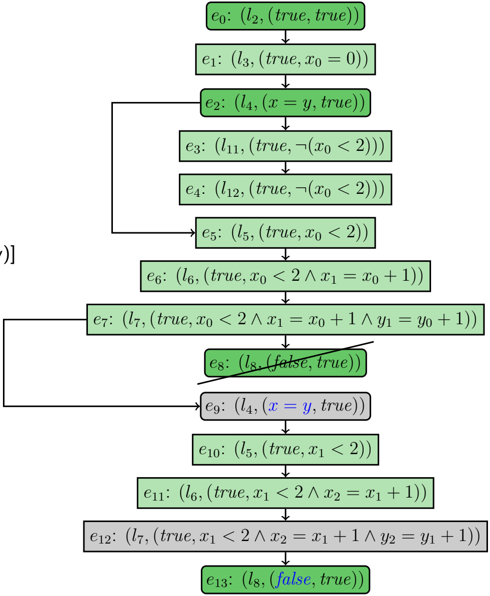




IMPACT: Example

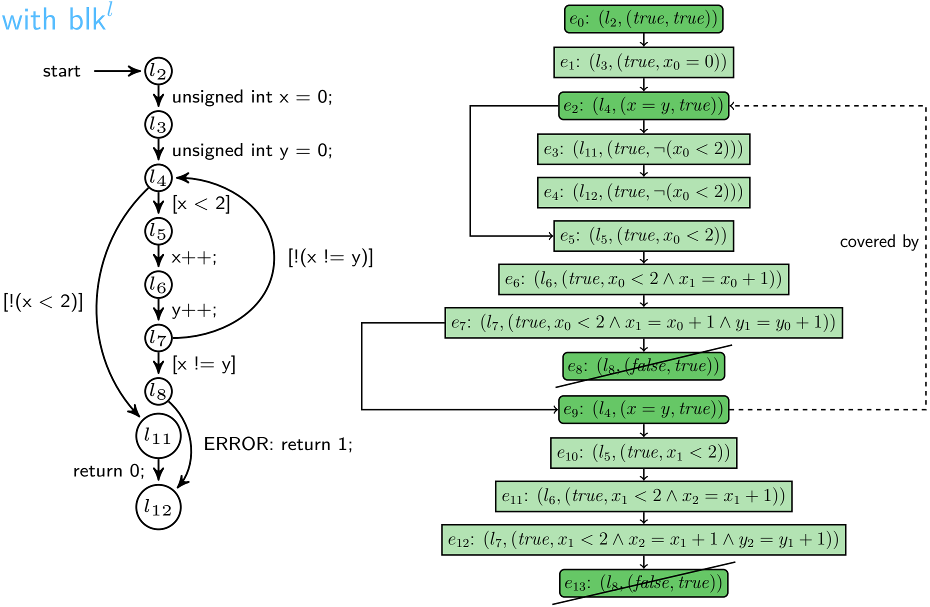
with blk^l





IMPACT: Example

with blk^l



Bounded Model Checking

- Bounded Model Checking:
  - Biere, Cimatti, Clarke, Zhu: Proc. TACAS 1999 [14]
  - 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

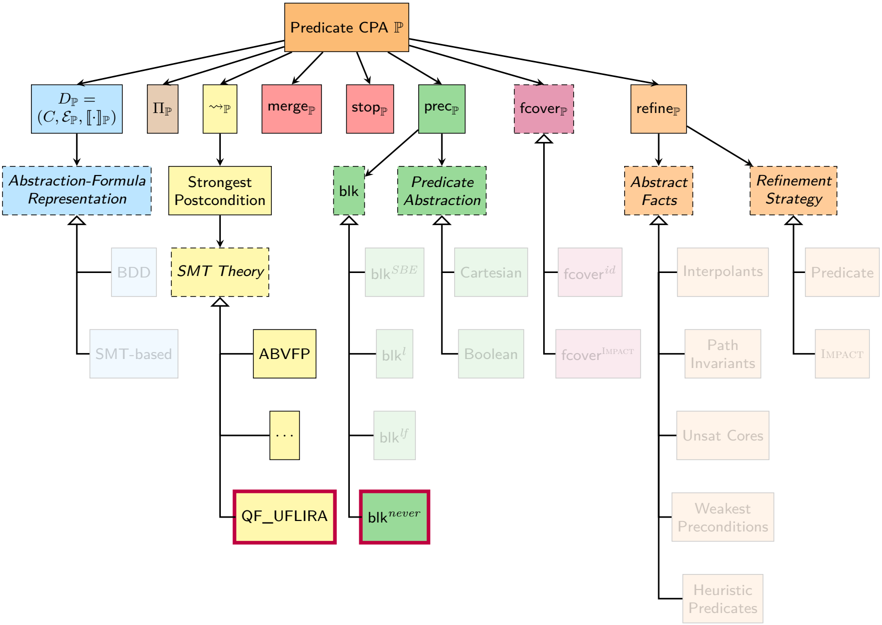
- Block Size (blk): blk^{never}

Furthermore:

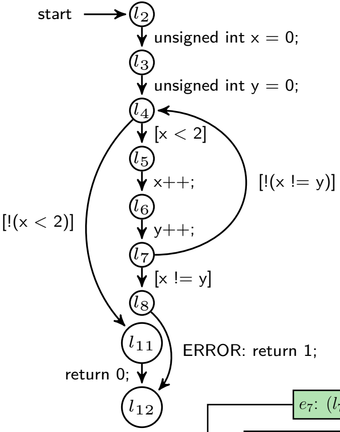
- Add CPA for bounding state space (e.g., loop bounds)
- Choices for abstraction formulas and refinement irrelevant because block end never encountered
- Use Algorithm for iterative BMC:

  $k = 1$
  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: Proc. PDMV 2011 [25]

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, 61 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

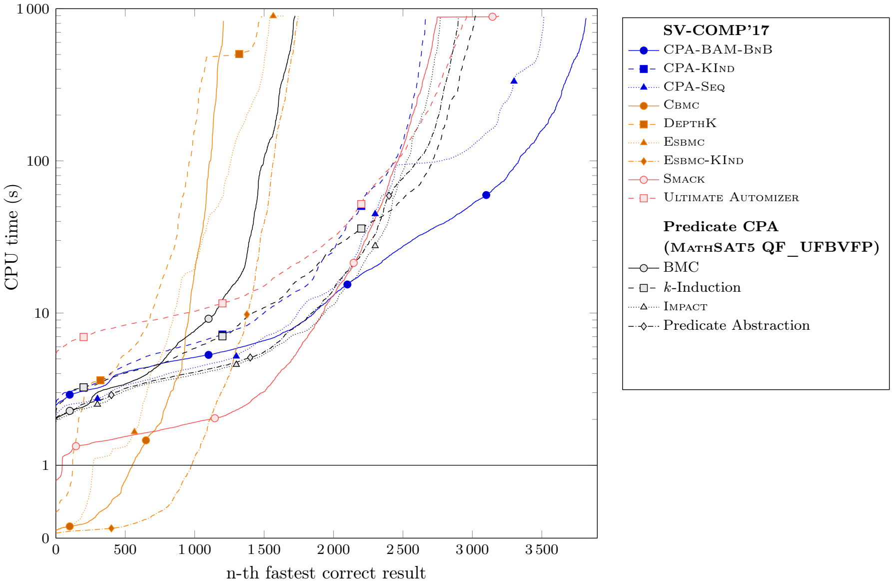
Number of correctly solved tasks:

- 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



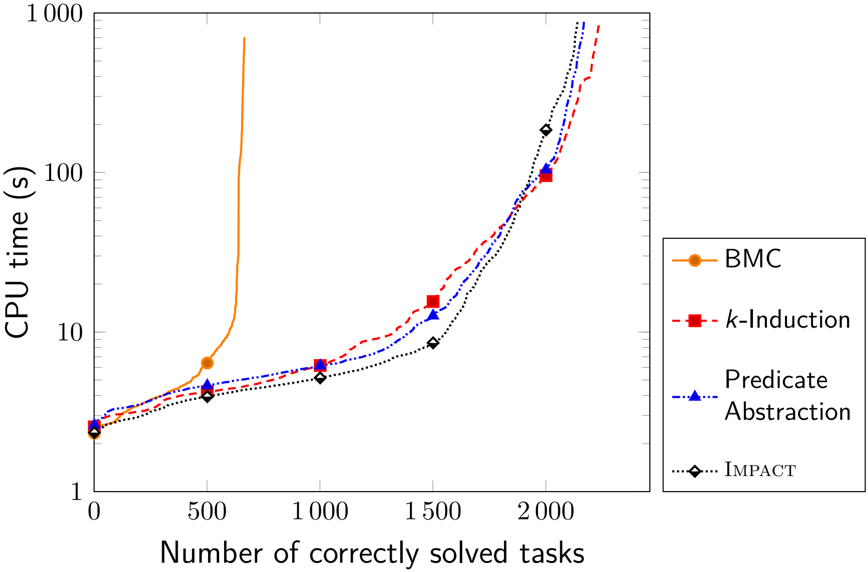
Evaluation: Enabling Experimental Studies

- Comparison of algorithms
  across different program categories
  Proc. VSTTE 2016, JAR [5, 9]
- SMT solvers for various theories and algorithms

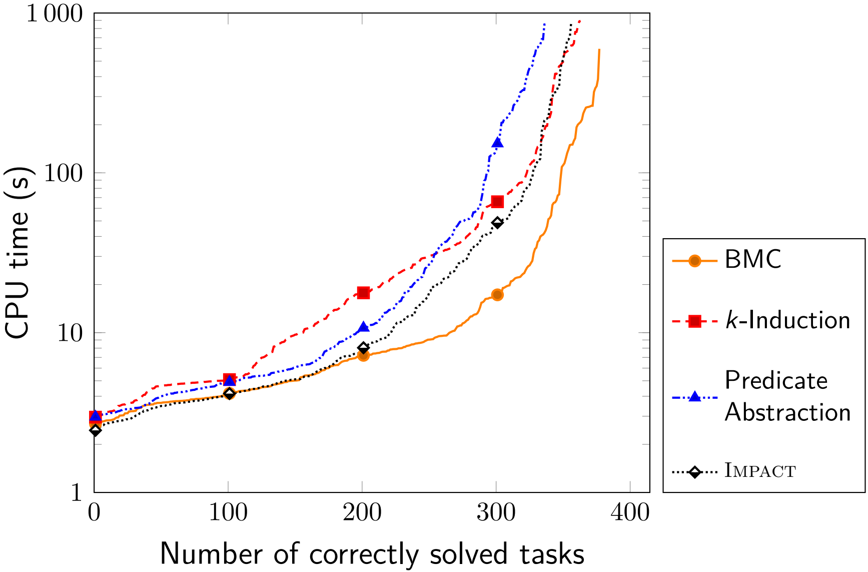
Experimental Comparison of Algorithms

- 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



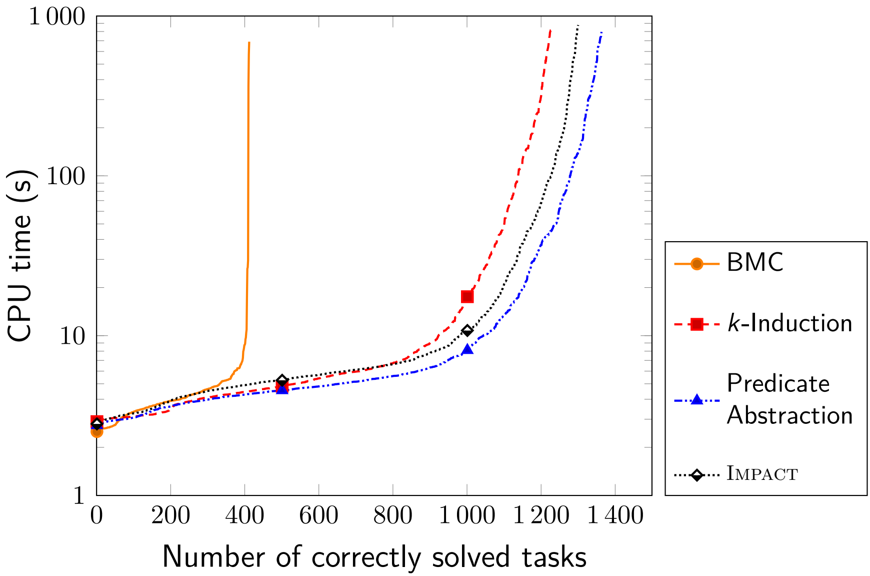
All 1 374 tasks with known bugs



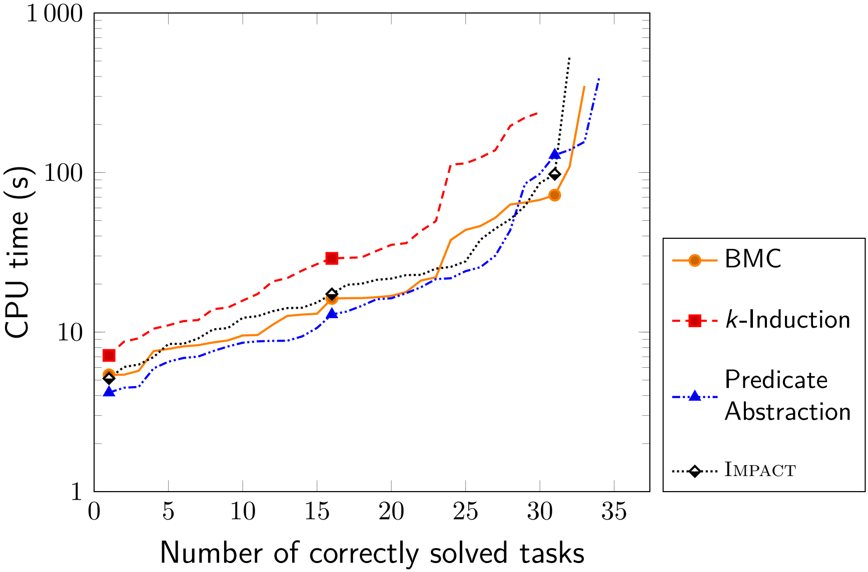
Category *Device Drivers*

- 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)*

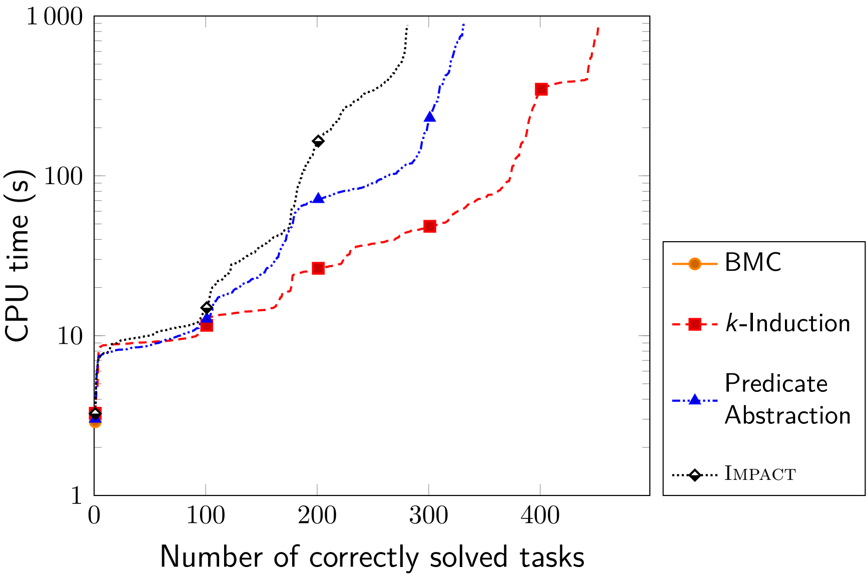
- Several thousand LOC per task
- Auto-generated
- Only integer variables
- Linear and non-linear arithmetics
- Complex and dense control structure

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



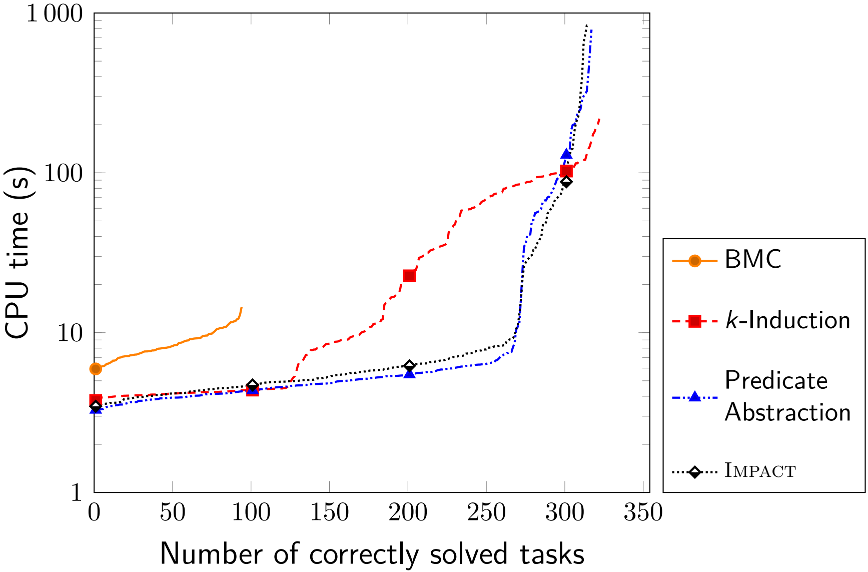
Category *ECA*: 411 tasks with known bugs

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

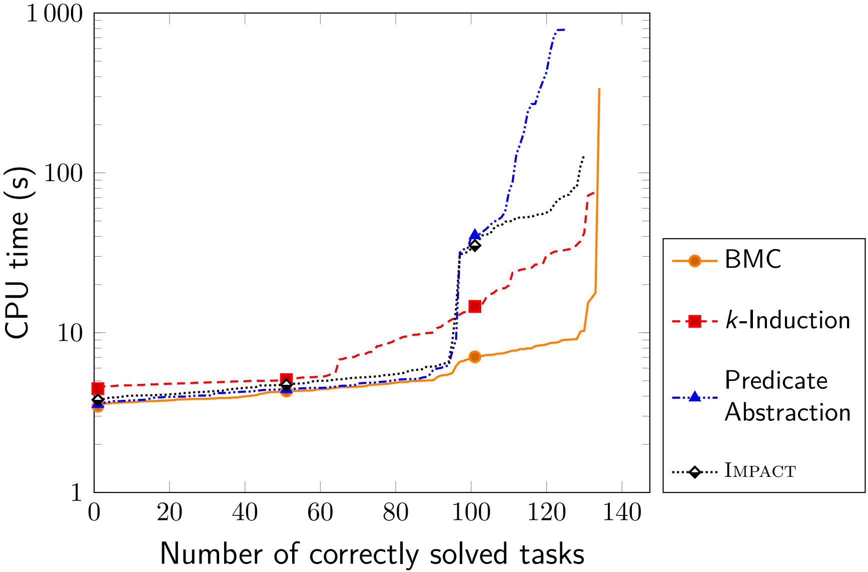
Category *Product Lines*

- 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

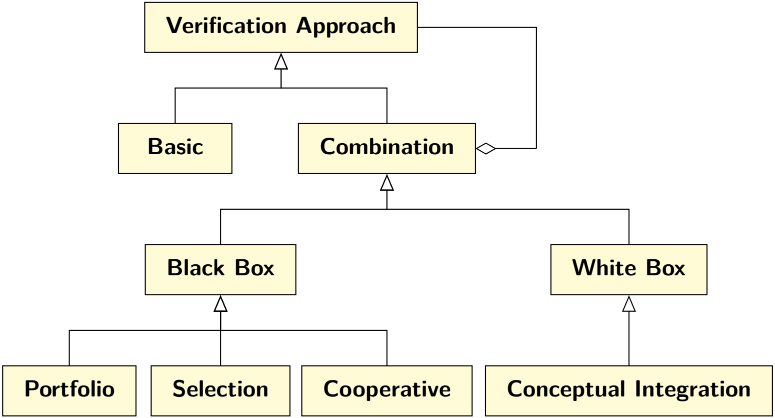
Depending on configuration, either (A) or (B) can be true!

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

Part 3

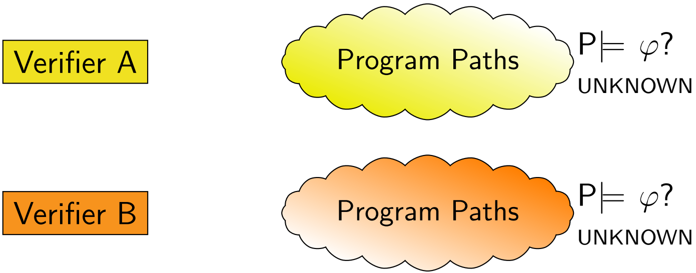
Cooperative Verification

Approaches for Combinations



Facing Hard Verification Tasks

Given: Program $P \models \varphi$?



Facing Hard Verification Tasks

Given: Program P$\models \varphi$?

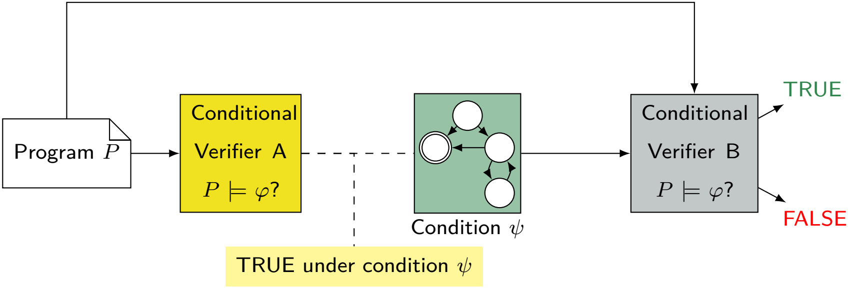
Verifier A — Program Paths — P$\models \varphi$? UNKNOWN

Verifier B — Program Paths — P$\models \varphi$? UNKNOWN

Verifier A + Verifier B — Program Paths — P$\models \varphi$ ✓

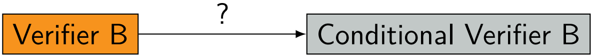
e.g., conditional model checking

Conditional Model Checking



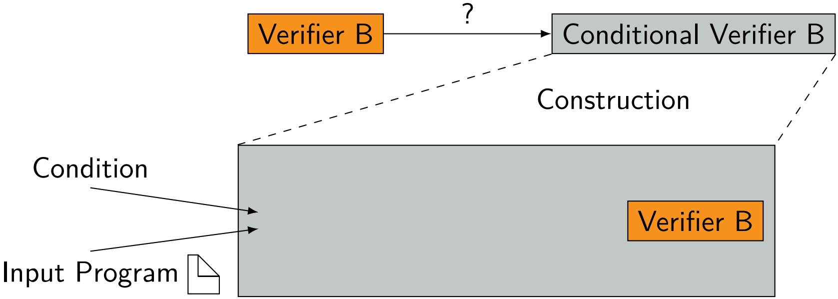
Proc. FSE 2012 [10]

Reducer-Based Construction



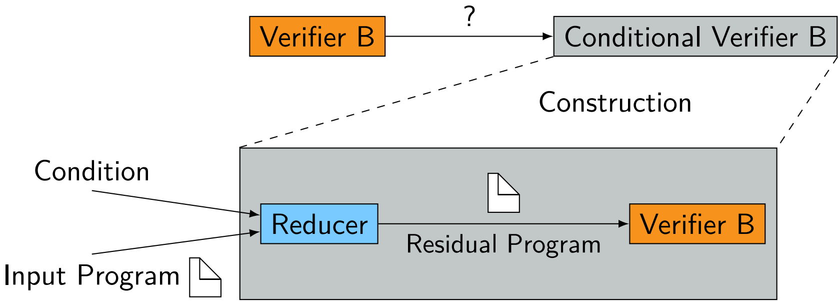
Proc. ICSE 2018 [13]

Reducer-Based Construction



Proc. ICSE 2018 [13]

Reducer-Based Construction

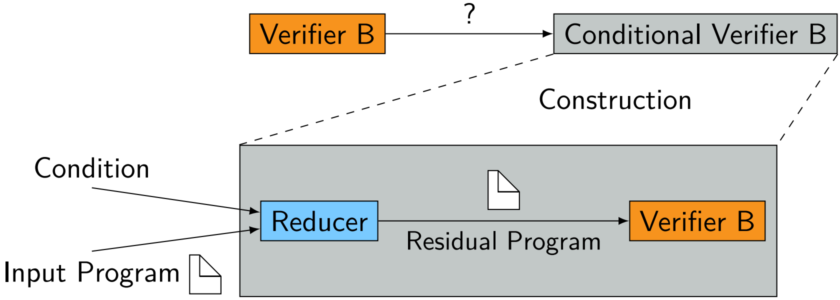


Reducer (preprocessor)

- Builds standard input (C program)
- Representing a subset of paths
- Contains at least all non-verified paths

Proc. ICSE 2018 [13]

Reducer-Based Construction



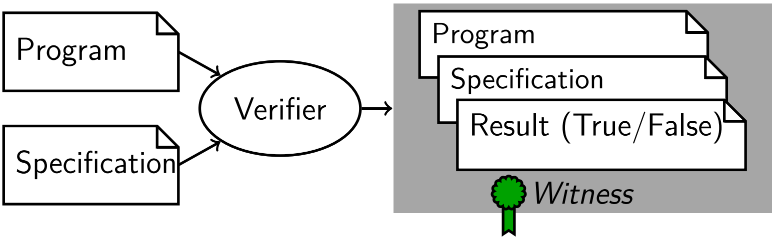
Reducer (preprocessor)

- Builds standard input (C program)
- Representing a subset of paths
- Contains at least all non-verified paths

+ Verifier-unspecific approach
+ Many conditional verifiers possible

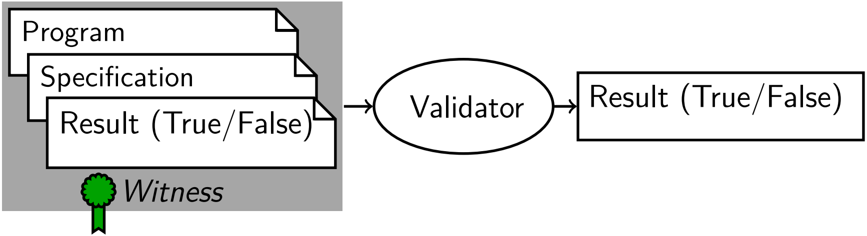
Proc. ICSE 2018 [13]

Software Verification with Witnesses



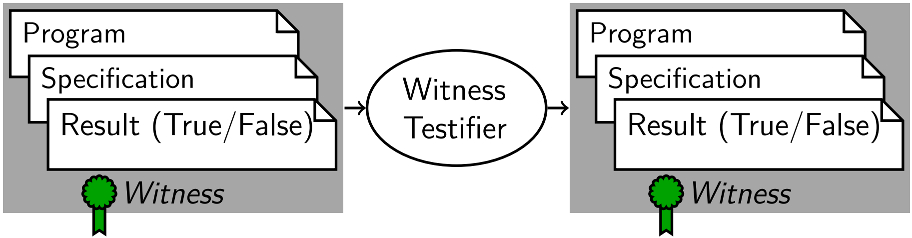
Proc. FSE 2015, 2016 [7, 6]

Witness-Based Result Validation

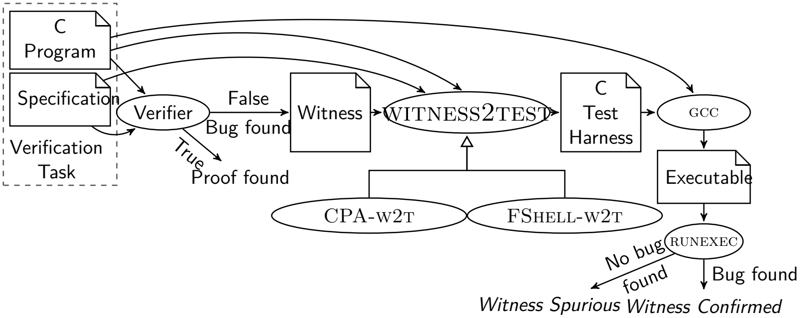


- Validate untrusted results
- Easier than full verification

Stepwise Refinement

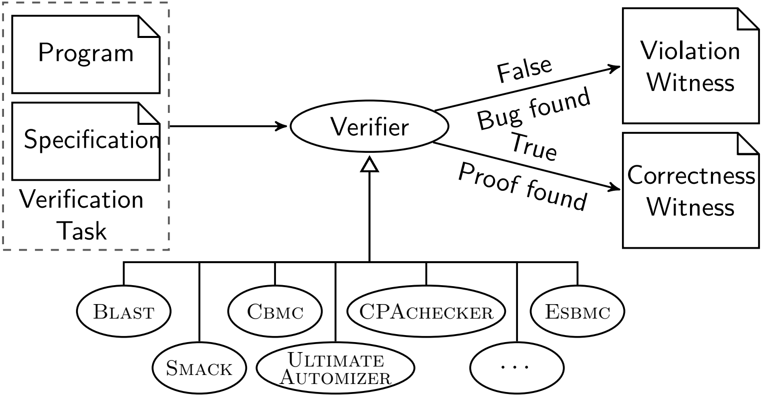


Execution-based Witness Validation



Proc. TAP 2018 [8]
Made "Generating Tests from Counterexamples" more practical (Proc. ICSE 2004, [4])

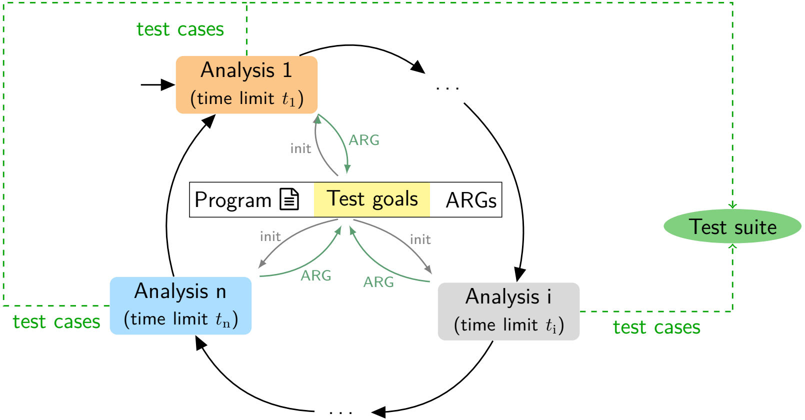
Witness Creation



CoVeriTest Approach: Thur 16:30 JUPITER

Proc. FASE 2019 [12]

Implemented in CPA✓



Information exchange realized with ARGs and init procedure

ARG: graph representation of explored, abstract state space

Conclusion

- Software verification: successful past, bright future
- Competitions solve several problems
- Science as knowledge compression
- Cooperating combinations are the future

Conclusion

- Software verification: successful past, bright future
- Competitions solve several problems
- Science as knowledge compression
- Cooperating combinations are the future

Thanks!

P.S.: Please ask the question about overfitting regarding TOOLympics and competitions in formal methods.

Overfitting or Requirements Specification?

Fitting to requirements and rules is natural for competitions.
When should this fitting and tuning be called **over**fitting?

The rules of the competition and the benchmark suite together are the requirements specification.
Consider examples from compiler construction and Formula 1.

Overfitting or Requirements Specification?

Fitting to requirements and rules is natural for competitions.
When should this fitting and tuning be called **over**fitting?

The rules of the competition and the benchmark suite together are the requirements specification.
Consider examples from compiler construction and Formula 1.

References I

- Beyer, D.: Competition on software verification (SV-COMP). In: Proc. TACAS. pp. 504–524. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_38
- Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proc. TACAS, Part 3. pp. 133–155. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_9
- Beyer, D.: Competition on software testing (Test-Comp). In: Proc. TACAS, Part 3. pp. 167–175. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_11
- Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE. pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
- Beyer, D., Dangl, M.: SMT-based software model checking: An experimental comparison of four algorithms. In: Proc. VSTTE. pp. 181–198. LNCS 9971, Springer (2016). https://doi.org/10.1007/978-3-319-48869-1_14
- Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351

References II

Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867

Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1

Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6

Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). https://doi.org/10.1145/2393596.2393664

Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV. pp. 504–518. LNCS 4590, Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51

Beyer, D., Jakobs, M.C.: Coveritest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23

References III

- Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
- Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
- Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008)
- Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: VeriFuzz: Program aware fuzzing (competition contribution). In: Proc. TACAS. LNCS 11429, Springer (2019)
- Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
- Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Proc. CAV. pp. 72–83. LNCS 1254, Springer (1997). https://doi.org/10.1007/3-540-63166-6_10

References IV

Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021

Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL. pp. 58–70. ACM (2002). https://doi.org/10.1145/503272.503279

Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D.: The RERS grey-box challenge 2012: Analysis of event-condition-action systems. In: Proc. ISoLA. pp. 608–614. LNCS 7609, Springer (2012). https://doi.org/10.1007/978-3-642-34026-0_45

Huisman, M., Klebanov, V., Monahan, R.: VerifyThis 2012 - A program verification competition. STTT **17**(6), 647–657 (2015). https://doi.org/10.1007/s10009-015-0396-8

Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438

Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Handbook of Model Checking, pp. 447–491. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_15

References V

- Kahsai, T., Tinelli, C.: PKind: A parallel k-induction based model checker. In: Proc. Int. Workshop on Parallel and Distributed Methods in Verification. pp. 55–62. EPTCS 72 (2011). https://doi.org/10.4204/EPTCS.72
- Kildall, G.A.: A unified approach to global program optimization. In: Proc. POPL. pp. 194–206. ACM (1973). https://doi.org/10.1145/512927.512945
- McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV. pp. 123–136. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_14
- Turing, A.: Checking a large routine. In: Report on a Conference on High Speed Automatic Calculating Machines. pp. 67–69. Cambridge Univ. Math. Lab. (1949)