Software Verification

An Overview of the State of the Art

Dirk Beyer

LMU Munich, Germany
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,
  → software model checking

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)
Competition 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]

(alphaetic order)
Automatic Software Verification

Program

Specification

Verifier

Program Specification

Result (True/False)

Witness
SV-COMP (Automatic Tools 2014, cumulative)
SV-COMP (Automatic Tools 2017, cumulative)
SV-COMP (Automatic Tools 2018, 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?

<table>
<thead>
<tr>
<th>Participant</th>
<th>CEGAR</th>
<th>Predicate Abstraction</th>
<th>Symbolic Execution</th>
<th>Bounded Model Checking</th>
<th>k-Induction</th>
<th>Property-Directed Reachability</th>
<th>Explicit-Value Analysis</th>
<th>Numeric Interval Analysis</th>
<th>Shape Analysis</th>
<th>Separation Logic</th>
<th>Bit-Precise Analysis</th>
<th>ARB-Based Analysis</th>
<th>Lazy Abstraction</th>
<th>Interpolation</th>
<th>Automata-Based Analysis</th>
<th>Concurrency Support</th>
<th>Ranking Functions</th>
<th>Evolutionary Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>2LS</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AProVE</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBMC</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBMC-Path</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPA-BAM-BnB</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPA-Lockator</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPA-Seq</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DePthK</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVINE-explicit</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVINE-SMT</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESBMC-kind</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAYHORN</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JBMc</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPF</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lazy-CSeg</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map2Check</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PeSCo</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pinaka</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PredatorHP</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skink</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmaCK</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPF</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Symbiotic</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UAutomizer</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ukoja</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UTsub</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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], → Poster)
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**
  \((\text{BLAST, CPAchecker, Slam, ...})\)

- **Impact**
  \((\text{CPAchecker, Impact, Wolverine, ...})\)

- **Bounded Model Checking**
  \((\text{Cbmc, CPAchecker, Esbmc, ...})\)

- **k-Induction**
  \((\text{CPAchecker, Esbmc, 2ls, ...})\)

- **Property-Directed Reachability (PDR, also known as IC3)**
  \((\text{CPAchecker, SeaHorn, VVT, ...})\)

- **Trace Abstraction**
  \((\text{Ultimate Automizer, CPAchecker in progress, ...})\)
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

→ What are their core concepts and key differences?
Motivation

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

→ 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

→ 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
Configurable Program Analysis (CPA):

- 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

![Diagram showing the flow of Source Code through Parser & CFA Builder to CPA Algorithm and finally to Results.]
Using the CPA Framework

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

Source Code → Parser & CFA Builder → CPA Algorithm → Predicate CPA → Results
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

\[ D_P = (C, E_P, \cdot_P) \]

\[ \Pi_P \]

\[ \overline{P} \]

merge\(_P\)

stop\(_P\)

prec\(_P\)
Predicate CPA

\[ D_P = (C, E_P, [\cdot]_P) \]

**Symbols:**
- \( D_P \): Domain of the predicate CPA
- \( E_P \): Environment
- \([\cdot]_P\): Implication operator
- \( \Pi_P \): Annotation
- \( merge_P \): Merge operation
- \( stop_P \): Stop condition
- \( prec_P \): Precise operation
- \( fcover_P \): F-cover operation
- \( refine_P \): Refinement operation

**Abbreviations:**
- ABVFP
- BDD
- QF_UFLIRA
- SMT Theory
- SBE
- blk
- l
- lf
- never
- fcover
- refine
- refine

**Topics:**
- Abstract Facts
- Interpolants
- Path Invariants
- Weakest Preconditions
- Heuristic Predicates
- Refinement Strategies
- Predicate Impact
- Cartesian Boolean
- SMT-based Predicate Abstraction
- Strongest Postcondition
- Impact Refinement

Predicate CPA: Abstract Domain

- Abstract state: $(\psi, \varphi)$
  - tuple of abstraction formula $\psi$ and path formula $\varphi$ (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:** \((\psi, \varphi)\)
  - tuple of abstraction formula \(\psi\) and path formula \(\varphi\) (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

\[ D_P = (C, \mathcal{E}_P, \cdot)_P \]

\[ \Pi_P, \sim_P, \text{merge}_P, \text{stop}_P, \text{prec}_P, \text{fcover}_P, \text{refine}_P \]

Abstraction-Formula Representation

BDD

SMT-based
Predicate CPA: CPA Operators

- Transfer relation:
  - computes strongest post
  - changes only path formula, new abstract state is \((\psi, \varphi')\)
  - 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 $(\psi, \varphi')$
  - 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 $(\psi, \varphi')$
  - 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 \((\psi, \varphi')\)
  - 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 \(\text{blk}\))
  - computes abstraction of current abstract state
  - new abstract state is \((\psi', \text{true})\)
Predicate CPA

$D_P = (C, \mathcal{E}_P, \vdash_P)$

$\Pi_P \leadsto P \implies \Pi_P \Rightarrow P$

$\Pi_P \Rightarrow P$

Strongest Postcondition

Predicate CPA $P$

Predicate Abstraction

$blk$

SMT Theory

$SBE$

$blk$

Boolean

Cartesian

SMT-based

BDD

QF_UFLIRA

ABVFP

...
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**

- $D_p = (C, e_p, [\cdot]_p)$
- $\Pi_p$
- $\sim_p$
- $\text{merge}_p$
- $\text{stop}_p$
- $\text{prec}_p$
- $\text{fcover}_p$
- $\text{refine}_p$

- **Abstraction-Formula Representation**
  - BDD
  - SMT Theory

- **Strongest Postcondition**
  - blk
  - $blk^{SBE}$
  - $blk^l$
  - $blk^{nf}$
  - QF_UFLIRA

- **Predicate Abstraction**
  - Cartesian
  - $fcover^{id}$
  - $fcover^{IMPACT}$

- **Abstract Facts**
  - $\Pi_p$
  - $\sim_p$
  - $\text{merge}_p$
  - $\text{stop}_p$

- **Refinement Strategy**
  - Interpolants
  - Predicate
  - Path Invariants
  - Unsat Cores
  - Weakest Preconditions
  - Heuristic Predicates

**SMT Theory**

**Impact**

**Predicates**

**Path Invariants**

**Unsat Cores**

**Weakest Preconditions**

**Heuristic Predicates**
Predicate Abstraction

[18, 20, 23, 19, 24]

Abstract-interpretation technique

Abstract domain constructed from a set of predicates $\pi$

Use CEGAR to add predicates to $\pi$ (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. \textbf{while} true \textbf{do}
2. \quad run CPA Algorithm
3. \quad \textbf{if} target state found \textbf{then}
4. \quad \quad call refine
5. \quad \quad \textbf{if} target state reachable \textbf{then}
6. \quad \quad \quad return false
7. \quad \quad else
8. \quad \quad \quad return true
Predicate CPA

\[ D_P = (C, E_P, [], P) \]

\[ \Pi_P \parallel \sim \parallel P \]

\[ \text{merge}_P \]

\[ \text{stop}_P \]

\[ \text{prec}_P \]

\[ \text{fcover}_P \]

\[ \text{refine}_P \]

\[ \text{Abstraction-Formula Representation} \]

\[ \text{Strongest Postcondition} \]

\[ \text{Predicate Abstraction} \]

\[ \text{Abstract Facts} \]

\[ \text{Refinement Strategy} \]

\[ \text{SMT Theory} \]

\[ \text{BDD} \]

\[ \text{ABVFP} \]

\[ \text{QF_UFLIRA} \]

\[ \text{SMT-based} \]

\[ \text{blk}^{SBE} \]

\[ \text{blk}' \]

\[ \text{blk}_{lf} \]

\[ \text{blk}_{never} \]

\[ \text{Cartesian} \]

\[ \text{Boolean} \]

\[ \text{fcover}_{id} \]

\[ \text{fcover}_{IMPACT} \]

\[ \text{Interpolants} \]

\[ \text{Unsat Cores} \]

\[ \text{Path Invariants} \]

\[ \text{Weakest Preconditions} \]

\[ \text{Heuristic Predicates} \]

Dirk Beyer

LMU Munich, Germany
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 Abstraction: Example

with $\text{blk}^l$, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{\text{false}\}$
Predicate Abstraction: Example

with $\text{blk}^l$, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{\text{false}\}$
Predicate Abstraction: Example

with \( \text{blk}^l \), \( \pi(l_4) = \{x = y\} \) and \( \pi(l_8) = \{\text{false}\} \)
Predicate Abstraction: Example

with \( blk_l \), \( \pi(l_4) = \{x = y\} \) and \( \pi(l_8) = \{\text{false}\} \)
Predicate Abstraction: Example

with $\text{blk}^l$, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{\text{false}\}$
Predicate Abstraction: Example

with $\text{blk}^l$, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{\text{false}\}$
Predicate Abstraction: Example

with $\text{blk}^l$, $\pi(l_4) = \{x = y\}$ and $\pi(l_8) = \{\text{false}\}$
Predicate Abstraction: Example

with blk\(^l\), \(\pi(l_4) = \{x = y\}\) and \(\pi(l_8) = \{false\}\)
Predicate Abstraction: Example

with \( \text{blk}^l, \pi(l_4) = \{x = y\} \) and \( \pi(l_8) = \{\text{false}\} \)
Predicate Abstraction: Example

with \( \text{blk}^l, \pi(l_4) = \{x = y\} \) and \( \pi(l_8) = \{false\} \)
Predicate Abstraction: Example

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

- "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

\[ D_P = (C, \mathcal{E}_P, \cdot)_P \]

Abstraction-Formula Representation

Strongest Postcondition

Predicate CPA \( \Pi_P \)

\( \sim_P \)

merge\(_P\)

stop\(_P\)

prec\(_P\)

fcover\(_P\)

refine\(_P\)

Abstract Facts

Refinement Strategy

SMT Theory

Predicat Abstraction

\( \text{blk} \)

\( \text{blk}^{\text{SBE}} \)

\( \text{blk}^L \)

\( \text{blk}^L \)

\( \text{blk}^{\text{never}} \)

ABVFP

Cartesian

Boolean

Interpolants

Impact

Path Invariants

Unsat Cores

Weakest Preconditions

Heuristic Predicates

QF_UFLIRA

SMT-based

BDD

SMT-based

Dirk Beyer

LMU Munich, Germany

49 / 99
**Predicate CPA**

\[
D_P = (C, \mathcal{E}_P, E_P)
\]

**Abstraction-Formula Representation**

- **Strongest Postcondition**
- **Predicate CPA**
- **Refine**
- **Predicate CPA**
- **Predicate CPA**

**SMT Theory**

- **BDD**
- **ABVFP**
- **QF_UFLIRA**

**Predicate CPA**

- **blk**
- **blk^{SBE}**
- **blk^l**
- **blk^{lf}**
- **blk^{never}**

**Refinement Strategy**

- **Interpolants**
- **Impact**
- **Unsat Cores**
- **Weak Preconditions**
- **Heuristic Predicates**
- **Path Invariants**

**Predicates**

- **fcover^{id}**
- **fcover^{IMPACT}**

**Abstract Facts**

- **Cartesian**
- **Boolean**

**Dirk Beyer**

LMU Munich, Germany
Impact: Example with blk$^l$

start → $l_2$
  → unsigned int $x = 0$;
  → $l_3$
    → unsigned int $y = 0$;
    → $l_4$
      → $[x < 2]$
      → $l_5$
        → $x++$;
        → $[! (x != y)]$
        → $l_6$
          → $y++$;
          → $l_7$
            → $[x != y]$ (error: return 1;)
            → $l_8$
              → $l_11$

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

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

$e_2$: ($l_4$, (true, $x_0 = 0 \land y_0 = 0$))
**Impact:** Example with $\text{blk}^l$

```c
unsigned int x = 0;
unsigned int y = 0;

[x < 2]  
[!x < 2]]

[x < 2]  
[!(x != y)]

[x != y]

[!(x != y)]

[l_11] ERROR: return 1;

return 0;
```

![Diagram of the program flow and conditions](image)
**Impact**: Example

with blk

```
unsigned int x = 0;
unsigned int y = 0;

[x < 2] x++;
[y + +;
[x != y]

return 0;
```

```
[e_0: (l_2, (true, true))
[e_1: (l_3, (true, x_0 = 0))
[e_2: (l_4, (true, true))
[e_3: (l_11, (true, ¬(x_0 < 2)))
[e_4: (l_12, (true, ¬(x_0 < 2)))
```
**Impact: Example with blk\(^l\)**

```c
unsigned int x = 0;
unsigned int y = 0;
[x < 2]  
[l5]  
[l6]  
[l7]  
[l8]  
[l11] ERROR: return 1;

[l2]  
[l3]  
[l4]  
[l12]
```

**States and Transitions:**

- **Start State:** `l2`
- **States:** `l2, l3, l4, l5, l6, l7, l11, l12`
- **Transitions:**
  - `l2` to `l3` on `unsigned int x = 0;`
  - `l3` to `l4` on `unsigned int y = 0;`
  - `l4` on `[x < 2]` to `l5`
  - `l5` on `x++;` to `l6`
  - `l6` on `y++;` to `l7`
  - `l7` on `[x != y]` to `l8`
  - `l8` on `return 0;` to `l11`
  - `l11` on `return 1;` to `l12`

**Error Conditions:**

- `e0: (l2, (true, true))`
- `e1: (l3, (true, x0 = 0))`
- `e2: (l4, (true, true))`
- `e3: (l11, (true, !x < 2))`
- `e4: (l12, (true, !x < 2))`
- `e5: (l5, (true, x0 < 2))`
- `e6: (l6, (true, x0 < 2 ∧ x1 = x0 + 1))`
- `e7: (l7, (true, x0 < 2 ∧ x1 = x0 + 1 ∧ y1 = y0 + 1))`

**Covered by:** Dirk Beyer LMU Munich, Germany
**IMPACT:** Example with blk$^l$

```
unsigned int x = 0;
unsigned int y = 0;

if (x < 2)
    y++;

if (!(x != y))
    return 1;

return 0;
```
**IMPACT: Example**

with blk\(^l\)

```
unsigned int x = 0;
unsigned int y = 0;
[x < 2]  
[l2]  
  unsigned int x = 0;
[l3]  
  unsigned int y = 0;
[l4]  
  [x < 2]
[l5]  
  x++;
[l6]  
  y++;
[l7]  
  [x != y]
[l8]  
return 0;
[l11]  
  ERROR: return 1;
[l12]
```

e\(_0\): (l\(_2\), (true, true))
e\(_1\): (l\(_3\), (true, x\(_0\) = 0))
e\(_2\): (l\(_4\), (true, true))
e\(_3\): (l\(_11\), (true, ¬(x\(_0\) < 2)))
e\(_4\): (l\(_12\), (true, ¬(x\(_0\) < 2)))
e\(_5\): (l\(_5\), (true, x\(_0\) < 2))
e\(_6\): (l\(_6\), (true, x\(_0\) < 2 ∧ x\(_1\) = x\(_0\) + 1))
e\(_7\): (l\(_7\), (true, x\(_0\) < 2 ∧ x\(_1\) = x\(_0\) + 1 ∧ y\(_1\) = y\(_0\) + 1))
e\(_8\): (l\(_8\), (true, true))
Impact: Example with $\text{blk}^l$

```
unsigned int x = 0;
unsigned int y = 0;
if (x < 2) {
    x++;
    y++;
}
if (x != y) {
    ERROR: return 1;
}
return 0;
```
**Impact:** Example with blk<sup>l</sup>

```
unsigned int x = 0;
unsigned int y = 0;
[l < 2]
x++;
y++;
[l != y]
```

```
e0: (l2, (true, true))
e1: (l3, (true, x0 = 0))
e2: (l4, (x = y, true))
e3: (l11, (true, ¬(x0 < 2)))
e4: (l12, (true, ¬(x0 < 2)))
e5: (l5, (true, x0 < 2))
e6: (l6, (true, x0 < 2 ∧ x1 = x0 + 1))
e7: (l7, (true, x0 < 2 ∧ x1 = x0 + 1 ∧ y1 = y0 + 1))
e8: (l8, (false, true))
e9: (l4, (true, x0 < 2 ∧ x1 = x0 + 1 ∧ y1 = y0 + 1 ∧ ¬(x1 = y1)))
```
Impact: Example

with blk\textsuperscript{l}

\begin{itemize}
  \item unsigned int x = 0;
  \item unsigned int y = 0;
  \item [x < 2]
  \item x++;
  \item y++;
  \item ![x != y]
  \item ![x < 2]
  \item ![x != y]
  \item ERROR: return 1;
\end{itemize}
**Impact:** Example with \( \text{blk}^l \)

\[
\begin{align*}
\text{unsigned int } x &= 0; \\
\text{unsigned int } y &= 0;
\end{align*}
\]

- \( e_0: (l_2, (true, true)) \)
- \( e_1: (l_3, (true, x_0 = 0)) \)
- \( e_2: (l_4, (x = y, true)) \)
- \( e_3: (l_{11}, (true, \neg(x_0 < 2))) \)
- \( e_4: (l_{12}, (true, \neg(x_0 < 2))) \)
- \( e_5: (l_5, (true, x_0 < 2)) \)
- \( e_6: (l_6, (true, x_0 < 2 \land x_1 = x_0 + 1)) \)
- \( e_7: (l_7, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1)) \)
- \( e_8: (l_8, (false, true)) \)
- \( e_9: (l_4, (true, true)) \)
- \( e_{10}: (l_5, (true, x_1 < 2)) \)
- \( e_{11}: (l_6, (true, x_1 < 2 \land x_2 = x_1 + 1)) \)
- \( e_{12}: (l_7, (true, x_1 < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1)) \)
- \( e_{13}: (l_8, (true, x_1 < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1 \land \neg(x_2 = y_2))) \)

**Diagram:**

- Start at \( l_2 \)
- \( l_3 \): unsigned int \( y = 0; \)
- \( l_4 \): \([x < 2]\)
  - \( l_5 \): \([!(x ! = y)]\)
  - \( l_6 \): \([x ! = y]\)
  - \( l_7 \): \([x ! = y]\)
  - \( l_8 \): \([!(x < 2)]\)
- \( l_{11} \): ERROR: return 1;
- \( l_{12} \): return 0;
**IMPACT:** Example

with blk\(^l\)

```
unsigned int x = 0;
unsigned int y = 0;
```

```
\[x < 2\]
```

```
\[!(x \neq y)\]
```

```
\[!(x < 2)\]
```

```
ERROR: return 1;
```

```
return 0;
```

```
e_0: (l_2, (true, true))
e_1: (l_3, (true, x_0 = 0))
e_2: (l_4, (x = y, true))
e_3: (l_{11}, (true, \neg(x_0 < 2)))
e_4: (l_{12}, (true, \neg(x_0 < 2)))
e_5: (l_5, (true, x_0 < 2))
e_6: (l_6, (true, x_0 < 2 \land x_1 = x_0 + 1))
e_7: (l_7, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1))
e_8: (l_8, (false, true))
e_9: (l_4, (true, true))
e_{10}: (l_5, (true, x_1 < 2))
e_{11}: (l_6, (true, x_1 < 2 \land x_2 = x_1 + 1))
e_{12}: (l_7, (true, x_1 < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1))
e_{13}: (l_8, (true, true))
```
Impact: Example

with $blk^l$

```
unsigned int x = 0;
unsigned int y = 0;

[x < 2]
```

```
!x != y]
```

```
[x < 2]
```

```
[x != y]
```

```
[l_2, (true, true)]
```

```
[l_3, (true, x_0 = 0))
```

```
[l_4, (x = y, true)]
```

```
[l_11, (true, -(x_0 < 2))]`
```

```
[l_12, (true, -(x_0 < 2))]
```

```
[l_5, (true, x_0 < 2)]
```

```
[l_6, (true, x_1 < 2 ∧ x_2 = x_1 + 1)]
```

```
[l_7, (true, x_0 < 2 ∧ x_1 = x_0 + 1 ∧ y_1 = y_0 + 1))]
```

```
[l_8, (false, true)]
```

```
[l_9, (false, true)]
```

```
[l_10, (true, x_1 < 2)]
```

```
[l_11, (true, x_1 < 2 ∧ x_2 = x_1 + 1)]
```

```
[l_12, (true, x_1 < 2 ∧ x_2 = x_1 + 1 ∧ y_2 = y_1 + 1))]
```

```
[l_13, (false, true)]
```
**IMPACT:** Example

with blk$^l$

```
unsigned int x = 0;
unsigned int y = 0;
[x < 2]
[x != y]
lib1: (l2, (true, true))
lib2: (l3, (true, x0 = 0))
lib3: (l4, (x = y, true))
lib4: (l5, (true, x0 < 2))
lib5: (l6, (true, x1 < 2))
lib6: (l7, (true, x0 < 2 ∧ x1 = x0 + 1))
lib7: (l8, (false, true))
```

```
lib8: (l9, (true, x1 < 2))
lib9: (l10, (true, x2 = x1 + 1))
lib10: (l11, (true, x1 < 2 ∧ x2 = x1 + 1))
lib11: (l12, (false, true))
```

```
ERROR: return 1;
lib12: (l13, (false, true))
lib13: (l14, (false, true))
```

```
[x != y]
lib1: (l2, (true, true))
lib2: (l3, (true, x0 = 0))
lib3: (l4, (x = y, true))
lib4: (l5, (true, x0 < 2))
lib5: (l6, (true, x1 < 2))
lib6: (l7, (true, x0 < 2 ∧ x1 = x0 + 1))
lib7: (l8, (false, true))
```

```
lib8: (l9, (true, x1 < 2))
lib9: (l10, (true, x2 = x1 + 1))
lib10: (l11, (true, x1 < 2 ∧ x2 = x1 + 1))
lib11: (l12, (false, true))
```

```
ERROR: return 1;
lib12: (l13, (false, true))
lib13: (l14, (false, true))
```

```
[x != y]
```
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): \( \text{blk}^{\text{never}} \)

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:
  1. \( k = 1 \)
  2. **while** !finished **do**
  3. run CPA Algorithm
  4. check feasibility of each abstract error state
  5. \( k++ \)
Predicate CPA

\[ D_P = (C, \xi_P, [\cdot]_P) \]

\[ \Pi_P \]

\[ \sim_P \]

\[ \text{merge}_P \]

\[ \text{stop}_P \]

\[ \text{prec}_P \]

\[ \text{fcover}_P \]

\[ \text{refine}_P \]

\[ \text{SMT Theory} \]

\[ \text{Predicates} \]

\[ \text{Refinement Strategy} \]

\[ \text{Abstract Facts} \]

\[ \text{Interpolants} \]

\[ \text{Predicate} \]

\[ \text{Path Invariants} \]

\[ \text{Unsat Cores} \]

\[ \text{Weakest Preconditions} \]

\[ \text{Heuristic Predicates} \]
Bounded Model Checking: Example with \( k = 1 \)

```c
unsigned int x = 0;
unsigned int y = 0;
[\( x < 2 \)]
\[ !(x < 2) \]
x++;
y++;  
[\( x \neq y \)]
\[ !(x \neq y) \]
```

### Formal Description

- \( e_0 : (l_2, (true, true), \{l_4 \mapsto -1\}) \)
- \( e_1 : (l_3, (true, x = 0), \{l_4 \mapsto -1\}) \)
- \( e_2 : (l_4, (true, x = 0 \land y = 0), \{l_4 \mapsto 0\}) \)
- \( e_3 : (l_{11}, (true, x = 0 \land y = 0 \land \neg(x < 2)), \{l_4 \mapsto 0\}) \)
- \( e_4 : (l_{12}, (true, x = 0 \land y = 0 \land \neg(x < 2)), \{l_4 \mapsto 0\}) \)
- \( e_5 : (l_5, (true, x = 0 \land y = 0 \land x < 2), \{l_4 \mapsto 0\}) \)
- \( e_6 : (l_6, (true, x = 0 \land y = 0 \land x < 2 \land x_1 = x + 1), \{l_4 \mapsto 0\}) \)
- \( e_7 : (l_7, (true, x = 0 \land y = 0 \land x < 2 \land x_1 = x + 1 \land y_1 = y + 1), \{l_4 \mapsto 0\}) \)
- \( e_8 : (l_8, (true, x = 0 \land y = 0 \land x < 2 \land x_1 = x + 1 \land y_1 = y + 1 \land \neg(x_1 = y_1)), \{l_4 \mapsto 0\}) \)
- \( e_9 : (l_{12}, (true, x = 0 \land y = 0 \land x < 2 \land x_1 = x + 1 \land y_1 = y + 1 \land \neg(x_1 = y_1)), \{l_4 \mapsto 0\}) \)
- \( e_{10} : (l_4, (true, x = 0 \land y = 0 \land x < 2 \land x_1 = x + 1 \land y_1 = y + 1 \land \neg(x_1 = y_1)), \{l_4 \mapsto 1\}) \)

**ERROR:** return 1;
1-Induction

1-Induction:

- **Base case**: Check that the safety property holds in the first loop iteration:
  \[ P(1) \]

  → 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
$k$-Induction with Auxiliary Invariants

**Induction:**
1. $k = 1$
2. **while** !finished **do**
3. BMC($k$)
4. Induction($k$, invariants)
5. $k++$

**Invariant generation:**
1. prec = <weak>
2. invariants = $\emptyset$
3. **while** !finished **do**
4. invariants = GenInv(prec)
5. prec = RefinePrec(prec)
**k-Induction: Example**

1. \( e_0: (l_4, (true, true), \{l_4 \mapsto 0\}) \)
2. \( e_1: (l_{11}, (true, \neg(x_0 < 2)), \{l_4 \mapsto 0\}) \)
3. \( e_2: (l_{12}, (true, \neg(x_0 < 2)), \{l_4 \mapsto 0\}) \)
4. \( e_3: (l_5, (true, x_0 < 2), \{l_4 \mapsto 0\}) \)
5. \( e_4: (l_6, (true, x_0 < 2 \land x_1 = x_0 + 1), \{l_4 \mapsto 0\}) \)
6. \( e_5: (l_7, (true, \land x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1), \{l_4 \mapsto 0\}) \)
7. \( e_6: (l_8, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)), \{l_4 \mapsto 0\}) \)
8. \( e_7: (l_{12}, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)), \{l_4 \mapsto 0\}) \)
9. \( e_8: (l_4, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land \neg(x_1 < 2)), \{l_4 \mapsto 1\}) \)
10. \( e_9: (l_{11}, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land \neg(x_1 < 2)), \{l_4 \mapsto 1\}) \)
11. \( e_{10}: (l_{12}, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land \neg(x_1 < 2)), \{l_4 \mapsto 1\}) \)
12. \( e_{11}: (l_5, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land x_1 < 2 \land x_2 = x_1 + 1), \{l_4 \mapsto 1\}) \)
13. \( e_{12}: (l_6, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land x_1 < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1), \{l_4 \mapsto 1\}) \)
14. \( e_{13}: (l_7, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land x_1 < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1 \land \neg(x_2 = y_2)), \{l_4 \mapsto 1\}) \)
15. \( e_{14}: (l_8, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land x < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1 \land \neg(x_2 = y_2)), \{l_4 \mapsto 1\}) \)
16. \( e_{15}: (l_{12}, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land x < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1 \land \neg(x_2 = y_2)), \{l_4 \mapsto 1\}) \)
17. \( e_{16}: (l_4, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land x < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1 \land \neg(x_2 = y_2)), \{l_4 \mapsto 2\}) \)
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 → 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

SV-COMP’17
- CPA-BAM-BnB
- CPA-KIND
- CPA-Seq
- Cbmc
- DepthK
- Esbmc
- Esbmc-KIND
- Smack
- Ultimate Automizer

Predicate CPA
(MathSAT5 QF_UFBVFP)
- BMC
- k-Induction
- IMPACT
- Predicate Abstraction
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

![Graph showing CPU time vs. number of correctly solved tasks for different methods: BMC, k-Induction, Predicate Abstraction, and IMPACT.](image)
All 1,374 tasks with known bugs

![Graph showing the CPU time in seconds against the number of correctly solved tasks. The graph includes lines for BMC, k-Induction, Predicate Abstraction, and IMPACT.](image-url)
Category *Device Drivers*

- Several thousands LOC per task
- Complex structures
- Pointer arithmetics
Category Device Drivers: 2440 bug-free tasks

![Graph showing CPU time (s) vs. Number of correctly solved tasks for different methods: BMC, k-Induction, Predicate Abstraction, IMPACT.](image)
Category **Device Drivers**: 355 tasks with known bugs

![Graph showing the number of correctly solved tasks against CPU time for different methods: BMC, k-Induction, Predicate Abstraction, and IMPACT.](image)

- **BMC**
- **k-Induction**
- **Predicate Abstraction**
- **IMPACT**

**Number of correctly solved tasks**

**CPU time (s)**
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

```c
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

Number of correctly solved tasks vs. CPU time (s)
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, \textsc{Impact}) scalable
- \textsc{Impact} is lazy: explores the state space and finds bugs quicker
- Predicate Abstraction is eager: prunes irrelevant parts and finds proofs quicker
Now, which do you think is better, i.e., solves more tasks?

- **k-Induction** solves 29% more tasks with bitprecise arithmetic
- **Predicate Abstraction** solves 3% more tasks 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.
SMT Study: Motivation

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

(A) $k$-Induction solves 29% more tasks

(B) Predicate Abstraction solves 3% more tasks
Now, which do you think is better, i.e., solves more tasks?

(A) k-Induction solves 29% more tasks

Z3 with bitprecise arithmetic

(B) Predicate Abstraction solves 3% more tasks

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

Verification Approach

Basic

Combination

Black Box

Portfolio

Selection

Cooperative

White Box

Conceptual Integration
Facing Hard Verification Tasks

Given: Program $P \models \varphi$?

Verifier A

Verifier B

Program Paths

$P \models \varphi$?  UNKNOWN

$P \models \varphi$?  UNKNOWN
Facing Hard Verification Tasks

Given: Program $P \models \varphi$?

Verifier A

Verifier B

Verifier A + Verifier B

e.g., conditional model checking
Conditional Model Checking

Program $P$ → Conditional Verifier A
$P \models \varphi$?

Condition $\psi$

TRUE under condition $\psi$

Conditional Verifier B
$P \models \varphi$?

TRUE

FALSE

Proc. FSE 2012 [10]

Dirk Beyer
LMU Munich, Germany
Reducer-Based Construction

Verifier B → ? → Conditional Verifier B

Proc. ICSE 2018 [13]
Reducer-Based Construction

Verifier B \rightarrow ? \rightarrow Conditional Verifier B

Construction

Condition

Input Program

Verifier B

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

Witness

Program
Specification
Result (True/False)

Witness

Testifier

Program
Specification
Result (True/False)

Witness
Execution-based Witness Validation

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

Program

Specification

Verification Task

Verifier

Violation Witness

Correctness Witness

BLAST

CBMC

CPAchecker

ESBMC

SMACK

ULTIMATE AUTOMIZER

False

Bug found

True

Proof found

False

Bug found

True

Proof found
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 overfitting?

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 overfitting?

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

© 2010 by slitz from Portugal

© 2018 by Vauxford
https://commons.wikimedia.org/wiki/File:2018_BMW_X6_xDrive30d_M_Sport_Automatic_3.0_Front.jpg


