A Unifying View on SMT-Based Software Verification

Dirk Beyer
Joint work with Matthias Dangl and Philipp Wendler
LMU Munich, Germany
Based on [1]:

Dirk Beyer, Matthias Dangl, Philipp Wendler:
A Unifying View on SMT-Based Software Verification

https://doi.org/10.1007/s10817-017-9432-6
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**
  \( \text{(CPAchecker, Esbmc, 2ls, ...)} \)

- **New: Interpolation-based model checking**
  \( \text{(CPAchecker)} \)
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):

- Beyer, Henzinger, Théoduloz: [2, CAV ’07]
- One single unifying algorithm for all algorithms based on state-space exploration
- Configurable components: abstract domain, abstract-successor computation, path sensitivity, ...
Using the CPA Framework

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

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

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

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

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

Π_P \quad \sim_P \quad \text{merge}_P \quad \text{stop}_P \quad \text{prec}_P
Predicate CPA: Abstract Domain

- Abstract state: \((\psi, \varphi)\)
  - tuple of abstraction formula \(\psi\) and path formula \(\varphi\) (for ABE)
  - conjunction 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)
  - conjunction 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

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

Abstraction-Formula Representation

BDD

SMT-based

\[ \Pi_P \quad \sim_P \quad \text{merge}_P \quad \text{stop}_P \quad \text{prec}_P \quad \text{fcover}_P \quad \text{refine}_P \]
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 \(blk\))
  - computes abstraction of current abstract state
  - new abstract state is \((\psi', true)\)
Predicate CPA

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

- Abstraction-Formula Representation
- BDD
- SMT-based
- SMT Theory

Strongest Postcondition
- $\Pi_P$
- $\sim_P$
- merge$_P$
- stop$_P$
- prec$_P$

Predicate CPA $P$

Predicate CPA $P$

Predication Abstraction
- blk
- blk$^{SBE}$
- blk$^l$
- blk$^{lf}$
- blk$^{never}$

Predicate CPA $P$

SMT Theory
- ABVFP
- QF_UFLIRA
- Cartesian
- Boolean
- blk$^l$
- blk$^{lf}$
- blk$^{never}$

Dirk Beyer
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 CPA**
  - $D_P = (C, \mathcal{E}_P, \mathcal{F}_P)$
  - $\Pi_P$
  - $\sim_P$
  - $\text{merge}_P$
  - $\text{stop}_P$
  - $\text{prec}_P$
  - $\text{fcover}_P$
  - $\text{refine}_P$

**Abstraction-Formula Representation**
- BDD
  - SMT-based
  - ABVFP
  - QF_UFLIRA

**Strongest Postcondition**
- blk
  - blk$^{SBE}$
  - blk$^l$
  - blk$^{lf}$
  - blk$^{never}$

**Predicate Abstraction**
- Cartesian
  - Boolean
  - blk$^{id}$
  - blk$^{IMPACT}$

**Abstract Facts**
- fcover$^{id}$
- fcover$^{IMPACT}$

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

**SMT Theory**
- SMT-based

**Heuristic Predicates**

**Predicates**

**ABVFP**

**Cartesian**

**Boolean**

**blk**

**fcover$^{id}$**

**fcover$^{IMPACT}$**

**Unsat Cores**

**Weakest Preconditions**

**Heuristic Predicates**

**Predicates**
Predicate Abstraction

- Predicate Abstraction
  - [5, CAV ’97], [7, POPL ’02], [6, POPL ’04]
  - Abstract-interpretation technique
  - Abstract domain constructed from a set of predicates $\pi$
  - Use CEGAR to add predicates to $\pi$ (refinement)
  - [4, J. ACM ’03]
  - 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 \textbf{return} false
7. \quad \quad \textbf{else}
8. \quad \quad \textbf{return} true
Predicate CPA

$D_{\mathcal{P}} = (C, \mathcal{E}_\mathcal{P}, \llbracket \cdot \rrbracket_{\mathcal{P}})$

- **Abstraction-Formula Representation**
  - BDD
  - SMT-based
  - ABVFP
  - QF_UFLIRA

- **Strongest Postcondition**
  - $\Pi_{\mathcal{P}}$
  - $\sim_{\mathcal{P}}$
  - merge$_{\mathcal{P}}$
  - stop$_{\mathcal{P}}$
  - prec$_{\mathcal{P}}$

- **Predicate CPA $\mathcal{P}$**

- **Predicate Abstraction**
  - $\text{blk}^{\text{SBE}}$
  - $\text{blk}^{I}$
  - $\text{blk}^{lf}$
  - $\text{blk}^{never}$

- **fcover$_{\mathcal{P}}$**
  - fcover$_{id}$
  - fcover$_{\text{IMPACT}}$

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

- **Abstract Facts**
```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

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

- Abstraction-Formula Representation
- Strongest Postcondition
- Predicate CPA \( P \)
- Predicate CPA \( P \)
- \( \Pi_P \)
- \( \sim_P \)
- \( \text{merge}_P \)
- \( \text{stop}_P \)
- \( \text{prec}_P \)
- \( \text{fcover}_P \)
- \( \text{refine}_P \)

- SMT Theory
- BDD
- ABVFP
- QF_UFLIRA
- \( \text{blk}^{SBE} \)
- \( \text{blk}^l \)
- \( \text{blk}^f \)
- \( \text{blk}^{never} \)

- Predicate Abstraction
- \( \Pi_P \)

- Abstract Facts
- Refinement Strategy
- Interpolants
- Path Invariants
- Unsat Cores
- Weakest Preconditions
- Heuristic Predicates

- Strongest Postcondition
- SMT-based:
  - ABVFP
  - QF_UFLIRA
  - ...
Predicate Abstraction: Example

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

with blk₁, π(l₄) = \(\{x = y\}\) and π(l₈) = \(\{false\}\)

```
unsigned int x = 0;
unsigned int y = 0;
[x < 2]
[!(x < 2)]
[l_2]

l_2 -> l_3
uns合一ed int x = 0;
uns合一ed int y = 0;
l_3 -> l_4
[x < 2]
l_4 -> l_5
[x < 2]
[l_5]
[!(x = y)]
[x == y];
[l_6]
[y++];
[l_7]
[x != y]
[l_7]
[l_8]
[l_8]
[l_11]
ERROR: return 1;
return 0;
[l_12]
```

```
e_0: (l_2, (true, true))
e_1: (l_3, (true, x_0 = 0))
e_2: (l_4, (true, x_0 = 0 ∧ y_0 = 0))
```
Predicate Abstraction: Example

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

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

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

[x < 2] [!(x != y)]
[x < 2]
[x < 2 ∧ x_1 = x_0 + 1 ∧ y_1 = y_0 + 1 ∧ ¬(x_1 = y_1)]
```

```
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, (x = y, ¬(x_0 < 2)))
e_4: (l_12, (x = y, ¬(x_0 < 2)))
e_5: (l_5, (x = y, x_0 < 2))
e_6: (l_6, (x = y, x_0 < 2 ∧ x_1 = x_0 + 1))
e_7: (l_7, (x = y, x_0 < 2 ∧ x_1 = x_0 + 1 ∧ y_1 = y_0 + 1))
e_8: (l_4, (x = y, x_0 < 2 ∧ x_1 = x_0 + 1 ∧ y_1 = y_0 + 1 ∧ ¬(x_1 = y_1)))
```
Predicat Abstraction: Example

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

```
unsigned int x = 0;
unsigned int y = 0;
[x < 2]
[x != y]
ERROR: return 1;
return 0;
```
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 $\pi(l_8) = \{false\}$
Predicate Abstraction: Example

with \( \pi(l_4) = \{x = y\} \) and \( \pi(l_8) = \{false\} \)

\[
\begin{align*}
n &\quad \text{unsigned int } x = 0; \\
&\quad \text{unsigned int } y = 0; \\
\{x < 2\} &\quad \text{[!} (x \neq y) \text{]} \\
\{! (x < 2)\} &\quad \text{ERROR: return 1;} \\
\end{align*}
\]

Diagram:

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

with \( \text{blk}^l, \pi(l_4) = \{ x = y \} \) and \( \pi(l_8) = \{ \text{false} \} \)
"Lazy Abstraction with Interpolants" [10, CAV ’06]
Abstraction is derived dynamically/lazily
Solution to avoiding expensive abstraction computations
Compute fixed point over three operations
   - Expand
   - Refine
   - Cover
Abstraction formula as SMT formula
Optimization: forced covering
Expressing **IMPACT**

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

Furthermore:

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

\[ D_p = (C, E_p, [\cdot]_p) \]

- **Strongest Postcondition**
  - \( \Pi_p \)
  - \( \sim^p \)
  - \( \text{merge}_p \)
  - \( \text{stop}_p \)
  - \( \text{prec}_p \)
  - \( \text{fcover}_p \)

- **Predicate Abstraction**
  - \( \text{blk}_p \)

- **Abstract Facts**
  - \( \text{fcover}^{id} \)
  - \( \text{fcover}^{IMPACT} \)

- **Refinement Strategy**
  - \( \text{Interpolants} \)
  - \( \text{Path Invariants} \)
  - \( \text{Unsat Cores} \)
  - \( \text{Weakest Preconditions} \)
  - \( \text{Heuristic Predicates} \)

- **Abstract-Formula Representation**
  - \( \text{BDD} \)
  - \( \text{SMT Theory} \)
  - \( \text{ABVFP} \)
  - \( \cdots \)

- **SMT-based**
  - \( \text{QF_UFLIRA} \)
  - \( \text{blk}^{SBE} \)
  - \( \text{blk}^l \)
  - \( \text{blk}^{lf} \)
  - \( \text{blk}^{never} \)

- **Heuristic Predicates**
  - \( \text{Predicate} \)
  - \( \text{IMPACT} \)
Predicate CPA

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

Predicates CPA

- \( \Pi_P \)
- \( \neg \Pi_P \)
- \( \text{merge}_P \)
- \( \text{stop}_P \)
- \( \text{prec}_P \)
- \( \text{fcov}_P \)
- \( \text{refine}_P \)

- **Abstraction-Formula Representation**
  - BDD
  - SMT-based
    - ABVFP
      - QF_UFLIRA
        - \( \text{blk}^{\\text{SBE}} \)
        - \( \text{blk}^{\\text{lf}} \)
        - \( \text{blk}^{\\text{never}} \)
  - SMT Theory

- **Strongest Postcondition**
  - Cartesian
  - Boolean

- **Predicate Abstraction**
  - \( \text{fcover}^{id} \)
  - \( \text{fcover}^{\text{IMPACT}} \)

- **Refinement Strategy**
  - Interpolants
  - Predicate
  - IMPACT

- **Abstract Facts**
  - Path Invariants
  - Unsat Cores
  - Weakest Preconditions
  - Heuristic Predicates

- **Heuristic Predicates**
Impact: Example

with blk\(^l\)

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

\[\text{if } x < 2\]
\[\text{if } !(x < 2)\]
x++;
y++;

\[\text{if } x \neq y\]
\[\text{if } !(x \neq y)\]

ERROR: return 1;
return 0;

\[e_0: (l_2, (true, true))\]
\[e_1: (l_3, (true, x_0 = 0))\]
\[e_2: (l_4, (true, x_0 = 0 \land y_0 = 0))\]
**Impact:** Example with blk

```c
unsigned int x = 0;
unsigned int y = 0;
[x < 2]
[!(x < 2)]
en0: (l2, (true, true))

[x != y]
[!(x != y)]
en1: (l3, (true, x0 = 0))

[x != y]
en2: (l4, (true, true))

[x < 2]
[!(x != y)]
en5: (l5, (true, true))

[x < 2]
en6: (l6, (true, true))

[x != y]
en7: (l7, (true, true))

[x < 2]
en8: (l8, (true, true))

[x == y]
en11: (l11, (true, true))

return 0;
en12: (l12, (true, true))

ERROR: return 1;
```

Diagram with nodes and transitions.
**Impact:** Example

with blk¹

```c
unsigned int x = 0;
unsigned int y = 0;
[x < 2]  [!(x < 2)]
[x != y]  [!(x != y)]
ERROR: return 1;
```

```
e0: (l2, (true, true))
  ↓
e1: (l3, (true, x0 = 0))
  ↓
e2: (l4, (true, true))
  ↓
e3: (l11, (true, ¬(x0 < 2)))
  ↓
e4: (l12, (true, ¬(x0 < 2)))
```

Covered by

Dirk Beyer
**Impact:** Example with blk$^l$

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

[x < 2]

[x < 2]

[x != y]

[x != y]

[!(x < 2)]

[!(x < 2)]

return 0;

return 0;
```

```
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))
```
**Impact:** Example with \( \text{blk}^l \)

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

![x < 2]
```

```
[! !(x != y)]
```

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

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

```
[l_4, (true, true)]
```

```
[l_5, (true, true)]
```

```
[l_6, (true, x_0 < 2 ∧ x_1 = x_0 + 1)]
```

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

```
[l_8, (true, x_0 < 2 ∧ x_1 = x_0 + 1 ∧ y_1 = y_0 + 1 ∧ ¬(x_1 = y_1))]`
```
**IMPACT: Example**

with blk$^l$

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

ERROR: return 1;
```
**IMPACT: Example**

with blk$^l$

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

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

[l12]
```

```
return 0;
```

```
ERROR: return 1;
```

```
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, ¬(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, (false, true))
```
**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++; [!(x != y)]
[l6] y++; [!(x < 2)]
[l7] [x != y]
[l8] [!(x < 2)]
[l11] ERROR: return 1;
[l12] return 0;
```

```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 $\text{blk}^l$

 Impact with $\text{blk}^l$

unsigned int $x = 0$;
unsigned int $y = 0$;

$x < 2$

$y++$;

$[x != y]$

$[!(x < 2)]$

ERROR: return 1;

return 0;

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

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

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

$e_3: (l_11, (true, \neg(x_0 < 2)))$

$e_4: (l_12, (true, \neg(x_0 < 2)))$

$e_5: (l_5, (true, x_0 < 2))$

$e_6: (l_6, (true, x_0 < 2 \land x_1 = x_0 + 1))$

$e_7: (l_7, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1))$

$e_8: (l_8, (false, true))$

$e_9: (l_4, (true, true))$
**Impact: Example**

with blk$^l$

```
unsigned int x = 0;
unsigned int y = 0;
if (x < 2)
    if (!(x != y))
        x++;
        y++;
    else if (x == y)
        return 0;
else
    if (x < 2)
        return 1;
```

```
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, !(x < 2)))
e_4: (l_12, (true, !(x < 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, (false, true))
e_9: (l_4, (true, true))
e_{10}: (l_5, (true, x_1 < 2))
e_{11}: (l_6, (true, x_1 < 2 ∧ x_2 = x_1 + 1))
e_{12}: (l_7, (true, x_1 < 2 ∧ x_2 = x_1 + 1 ∧ y_2 = y_1 + 1))
e_{13}: (l_8, (true, x_1 < 2 ∧ x_2 = x_1 + 1 ∧ y_2 = y_1 + 1 ∧ !(x_2 = y_2)))
```
**Impact:** Example

with \( \text{blk}^l \)

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

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

```c
return 0;
```

```c
ERROR: return 1;
```

---

**Diagram:**

![Diagram of the code example]
**Impact: Example**

with \( \text{blk}^l \)

```
unsigned int x = 0;
unsigned int y = 0;
[l_2]
  [x < 2]
  [l_3]
  [x != y]
  [l_4]
  [x < 2]
  [l_5]
  [x++;
  [l_6]
  [y++;
  [l_7]
  [x != y]
  [l_8]
  [l_11]
  ERROR: return 1;
  [ l_12]
```

```
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, -(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, (false, true))
e_9: (l_4, (x = y, true))
e_10: (l_5, (true, x_1 < 2))
e_11: (l_6, (true, x_1 < 2 ∧ x_2 = x_1 + 1))
e_12: (l_7, (true, x_1 < 2 ∧ x_2 = x_1 + 1 ∧ y_2 = y_1 + 1))
e_13: (l_8, (false, true))
```
**Impact:** Example

with blk

```
unsigned int x = 0;
unsigned int y = 0;
if (x < 2)
  x++;
  y++;
if (x != y)
  return 1;
return 0;
```
Bounded Model Checking

- Bounded Model Checking:
  - Biere, Cimatti, Clarke, Zhu: [3, TACAS '99]
  - No abstraction
  - Unroll loops up to a loop bound $k$
  - Check that $P$ holds in the first $k$ iterations:

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

▸ Block Size (blk): $\text{blk}^{\text{never}}$

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. $k++$
Predicates CPA

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

- \( \Pi_P \)
- \( \sim P \)
- \( \text{merge}_P \)
- \( \text{stop}_P \)
- \( \text{prec}_P \)
- \( \text{fcover}_P \)
- \( \text{refine}_P \)

- \( \text{Abstraction-Formula Representation} \)
- \( \text{Strongest Postcondition} \)
- \( \text{Predicate Abstraction} \)
- \( \text{Abstract Facts} \)
- \( \text{Refinement Strategy} \)

- \( \text{BDD} \)
- \( \text{SMT Theory} \)
- \( \text{ABVFP} \)
- \( \text{QF_UFLIRA} \)
- \( \text{blk} \)
- \( \text{blk}^{SBE} \)
- \( \text{blk}^{lf} \)
- \( \text{blk}^{never} \)
- \( \text{Cartesian} \)
- \( \text{Boolean} \)
- \( \text{fcover}^{id} \)
- \( \text{fcover}^{IMPACT} \)
- \( \text{Interpolants} \)
- \( \text{Path Invariants} \)
- \( \text{Predicate} \)
- \( \text{Unsat Cores} \)
- \( \text{Weakest Preconditions} \)
- \( \text{Heuristic Predicates} \)

Predicate CPA
Bounded Model Checking: Example with $k = 1$

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

[l2] start

[l3] unsigned int x = 0;

[l4] unsigned int y = 0;

[e0]: (l2, (true, true), \{l4 \mapsto -1\})

[e1]: (l3, (true, x_0 = 0), \{l4 \mapsto -1\})

[e2]: (l4, (true, x_0 = 0 \land y_0 = 0), \{l4 \mapsto 0\})

[e3]: (l_{11}, (true, x_0 = 0 \land y_0 = 0 \land \neg(x_0 < 2)), \{l4 \mapsto 0\})

[e4]: (l_{12}, (true, x_0 = 0 \land y_0 = 0 \land \neg(x_0 < 2)), \{l4 \mapsto 0\})

[e5]: (l_5, (true, x_0 = 0 \land y_0 = 0 \land x_0 < 2), \{l4 \mapsto 0\})

[e6]: (l_6, (true, x_0 = 0 \land y_0 = 0 \land x_0 < 2 \land x_1 = x_0 + 1), \{l4 \mapsto 0\})

[e7]: (l_7, (true, x_0 = 0 \land y_0 = 0 \land x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1), \{l4 \mapsto 0\})

[e8]: (l_8, (true, x_0 = 0 \land y_0 = 0 \land x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)), \{l4 \mapsto 0\})

[e9]: (l_{12}, (true, x_0 = 0 \land y_0 = 0 \land x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)), \{l4 \mapsto 0\})

[e10]: (l_4, (true, x_0 = 0 \land y_0 = 0 \land x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(\neg(x_1 = y_1))), \{l4 \mapsto 1\})

[l5] [x < 2]

[l6] x++;

[l7] y++;

[e1]: (l_3, (true, x_0 = 0), \{l4 \mapsto -1\})

[l8] [x != y]

[l9] return 0;

[l10] return 0;

[l11] return 1;

[l12] return 0;
1-Induction

- **1-Induction:**
  - **Base case:** Check that the safety property holds in the first loop iteration:
    \[ P(1) \]
    
  \[ \rightarrow \text{ Equivalent to BMC with loop bound 1} \]
  - **Step case:** Check that the safety property is 1-inductive:
    \[ \forall n : (P(n) \Rightarrow P(n + 1)) \]
\textit{k-Induction}

\begin{itemize}
  \item \textit{k-Induction generalizes the induction principle:}
    \begin{itemize}
      \item No abstraction
      \item Base case: Check that $P$ holds in the first $k$ iterations:
        \[ \rightarrow \text{ Equivalent to BMC with loop bound } k \]
      \item Step case: Check that the safety property is $k$-inductive:
        \[ \forall n : \left( \left( \bigwedge_{i=1}^{k} P(n+i-1) \right) \rightarrow P(n+k) \right) \]
    \end{itemize}
  \item Stronger hypothesis is more likely to succeed
  \item Add auxiliary invariants
  \item Kahsai, Tinelli: \cite[\textit{PDMC ’11}]{8}
\end{itemize}
**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 = ∅
3. while !finished do
4. invariants = GenInv(prec)
5. prec = RefinePrec(prec)
**k-Induction: Example** with $k = 1$ (and loop bound $k + 1 = 2$)

- $e_0: (l_4, (true, true), \{l_4 \mapsto 0\})$
- $e_1: (l_{11}, (true, \neg(x_0 < 2)), \{l_4 \mapsto 0\})$
- $e_2: (l_{12}, (true, \neg(x_0 < 2)), \{l_4 \mapsto 0\})$
- $e_3: (l_5, (true, x_0 < 2), \{l_4 \mapsto 0\})$
- $e_4: (l_6, (true, x_0 < 2 \land x_1 = x_0 + 1), \{l_4 \mapsto 0\})$
- $e_5: (l_7, (true, \land x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1), \{l_4 \mapsto 0\})$
- $e_6: (l_8, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)), \{l_4 \mapsto 0\})$
- $e_7: (l_{12}, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)), \{l_4 \mapsto 0\})$
- $e_8: (l_4, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)), \{l_4 \mapsto 1\})$
- $e_9: (l_{11}, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land \neg(x_1 < 2), \{l_4 \mapsto 1\})$
- $e_{10}: (l_{12}, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land \neg(x_1 < 2), \{l_4 \mapsto 1\})$
- $e_{11}: (l_5, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land x_1 < 2 \land x_2 = x_1 + 1, \{l_4 \mapsto 1\})$
- $e_{12}: (l_6, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land x_1 < 2 \land x_2 = x_1 + 1, \{l_4 \mapsto 1\})$
- $e_{13}: (l_7, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land x_1 < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1, \{l_4 \mapsto 1\})$
- $e_{14}: (l_8, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land x < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1 \land \neg(x_2 = y_2), \{l_4 \mapsto 1\})$
- $e_{15}: (l_{12}, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land x < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1 \land \neg(x_2 = y_2), \{l_4 \mapsto 1\})$
- $e_{16}: (l_4, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg(x_1 = y_1)) \land x < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1 \land \neg(x_2 = y_2), \{l_4 \mapsto 2\})$
Interpolation and SAT-Based Model Checking

- McMillan: [9, CAV ’03]
- Interpolation-based model checking (IMC)
  - Construct fixed points by interpolants derived from unsatisfiable BMC queries
  - Originally designed for finite-state systems (circuit); recently adopted for programs
Expressing IMC

- Block Size (blk): blk\(^l\)

Furthermore:
- Use *block formulas* to partition BMC queries
  - Already recorded in predicate abstract state: (ψ, φ, σ)
- IMC algorithm (on top of CPA Algorithm):
  1. \(k = 1\)
  2. **while** !finished **do**
  3. run CPA Algorithm
  4. check feasibility of each abstract error state
  5. partition unsatisfiable BMC queries
  6. construct fixed points by interpolants
  7. \(k++\)
**IMC: Example** (error path to $l_8$ with one loop unrolling)

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

```
x++;
```

```
y++;
```

```
[x < 2]
```

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

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

```
[!(x != y)]
```

```
ERROR: return 1;
```

```
return 0;
```

```
e_0: (l_2, (true, true, true), \{l_4 \mapsto \neg 1\})
e_1: (l_3, (true, x_0 = 0, true), \{l_4 \mapsto \neg 1\})
e_2: (l_4, (true, true, x_0 = 0 \land y_0 = 0), \{l_4 \mapsto 0\})
e_3: (l_5, (true, x_0 < 2, x_0 = 0 \land y_0 = 0), \{l_4 \mapsto 0\})
e_4: (l_6, (true, x_0 < 2 \land x_1 = x_0 + 1, x_0 = 0 \land y_0 = 0), \{l_4 \mapsto 0\})
e_5: (l_7, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1, x_0 = 0 \land y_0 = 0), \{l_4 \mapsto 0\})
e_6: (l_4, (true, true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg (\neg (x_1 = y_1))), \{l_4 \mapsto 1\})
e_7: (l_5, (true, x_1 < 2, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg (\neg (x_1 = y_1))), \{l_4 \mapsto 1\})
e_8: (l_6, (true, x_1 < 2 \land x_2 = x_1 + 1, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg (\neg (x_1 = y_1))), \{l_4 \mapsto 1\})
e_9: (l_7, (true, x_1 < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg (\neg (x_1 = y_1))), \{l_4 \mapsto 1\})
e_{10}: (l_8, (true, true, 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\})
```
IMC: Example (error path to $l_8$ with one loop unrolling)

\begin{align*}
e_0: & \ (l_2, (true, true, true), \{ l_4 \mapsto -1 \}) \\
e_1: & \ (l_3, (true, x_0 = 0, true), \{ l_4 \mapsto -1 \}) \\
e_2: & \ (l_4, (true, true, x_0 = 0 \land y_0 = 0), \{ l_4 \mapsto 0 \}) \\
e_3: & \ (l_5, (true, x_0 < 2, x_0 = 0 \land y_0 = 0), \{ l_4 \mapsto 0 \}) \\
e_4: & \ (l_6, (true, x_0 < 2 \land x_1 = x_0 + 1, x_0 = 0 \land y_0 = 0), \{ l_4 \mapsto 0 \}) \\
e_5: & \ (l_7, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1, x_0 = 0 \land y_0 = 0), \{ l_4 \mapsto 0 \}) \\
e_6: & \ (l_4, (true, true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg \neg (x_1 = y_1)), \{ l_4 \mapsto 1 \}) \\
e_7: & \ (l_5, (true, x_1 < 2, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg \neg (x_1 = y_1)), \{ l_4 \mapsto 1 \}) \\
e_8: & \ (l_6, (true, x_1 < 2 \land x_2 = x_1 + 1, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg \neg (x_1 = y_1)), \{ l_4 \mapsto 1 \}) \\
e_9: & \ (l_7, (true, x_1 < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg \neg (x_1 = y_1)), \{ l_4 \mapsto 1 \}) \\
e_{10}: & \ (l_8, (true, true, 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 \}) \
\end{align*}

\begin{align*}
& x_0 = 0 \land y_0 = 0 \land x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \neg \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) \\
& \text{interpolant: } x_1 = y_1 \\
\end{align*}
IMC: Example (error path to $l_8$ with one loop unrolling)

```
e_0: (l_2, (true, true, true), \{l_4 \mapsto -1\})
e_1: (l_3, (true, x_0 = 0, true), \{l_4 \mapsto -1\})
e_2: (l_4, (true, true, x_0 = 0 \land y_0 = 0), \{l_4 \mapsto 0\})
e_3: (l_5, (true, x_0 < 2, x_0 = 0 \land y_0 = 0), \{l_4 \mapsto 0\})
e_4: (l_6, (true, x_0 < 2 \land x_1 = x_0 + 1, x_0 = 0 \land y_0 = 0), \{l_4 \mapsto 0\})
e_5: (l_7, (true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1, x_0 = 0 \land y_0 = 0), \{l_4 \mapsto 0\})
e_6: (l_4, (true, true, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \lnot(\lnot(x_1 = y_1)), \{l_4 \mapsto 1\})
e_7: (l_5, (true, x_1 < 2, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \lnot(\lnot(x_1 = y_1)), \{l_4 \mapsto 1\})
e_8: (l_6, (true, x_1 < 2 \land x_2 = x_1 + 1, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \lnot(\lnot(x_1 = y_1)), \{l_4 \mapsto 1\})
e_9: (l_7, (true, x_1 < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1, x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \lnot(\lnot(x_1 = y_1)), \{l_4 \mapsto 1\})
e_{10}: (l_8, (true, true, x_1 < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1 \land \lnot(x_2 = y_2)), \{l_4 \mapsto 1\})
```

$x_0 = y_0 \land x_0 < 2 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \land \lnot(\lnot(x_1 = y_1)) \land$

\[x_1 < 2 \land x_2 = x_1 + 1 \land y_2 = y_1 + 1 \land \lnot(x_2 = y_2)\]

fixed point $x = y$ reached
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

- 5 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, 27 gold 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

Number of wrong results:

- Comparable with other tools
- No wrong proofs (sound)
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-B\textsc{NB}, CPA-\textsc{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

<table>
<thead>
<tr>
<th>CPU time (s)</th>
<th>SV-COMP’17</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPA-BAM-BnB</td>
<td></td>
</tr>
<tr>
<td>CPA-KInd</td>
<td></td>
</tr>
<tr>
<td>CPA-Seq</td>
<td></td>
</tr>
<tr>
<td>Cbmc</td>
<td></td>
</tr>
<tr>
<td>DepthK</td>
<td></td>
</tr>
<tr>
<td>Esbmc</td>
<td></td>
</tr>
<tr>
<td>Esbmc-KInd</td>
<td></td>
</tr>
<tr>
<td>Smack</td>
<td></td>
</tr>
<tr>
<td>Ultimate Automizer</td>
<td></td>
</tr>
<tr>
<td>Predicate CPA</td>
<td>(MathSAT5 QF_UFBVFP)</td>
</tr>
<tr>
<td>BMC</td>
<td></td>
</tr>
<tr>
<td>k-Induction</td>
<td></td>
</tr>
<tr>
<td>IMPACT</td>
<td></td>
</tr>
<tr>
<td>Predicate Abstraction</td>
<td></td>
</tr>
</tbody>
</table>

n-th fastest correct result

Dirk Beyer
Evaluation: Enabling Experimental Studies

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

- 5287 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

Number of correctly solved tasks vs. CPU time (s)

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

Dirk Beyer
All 1,374 tasks with known bugs

Number of correctly solved tasks vs. CPU time (s)

- **BMC**
- **k-Induction**
- **Predicate Abstraction**
- **IMPACT**
Category Device Drivers

- Several thousands LOC per task
- Complex structures
- Pointer arithmetics
Category *Device Drivers*: 2440 bug-free tasks
Category **Device Drivers**: 355 tasks with known bugs

![Graph showing CPU time (s) vs. Number of correctly solved tasks for different methods: BMC, k-Induction, Predicate Abstraction, IMPACT.](image)
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

```java
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 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

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

![Graph showing the number of correctly solved tasks vs CPU time for different methods: BMC, k-Induction, Predicate Abstraction, IMPACT.](image)
Category **Product Lines**: 265 tasks with known bugs

![Graph showing CPU time (s) vs. Number of correctly solved tasks for different techniques: BMC, k-Induction, Predicate Abstraction, and IMPACT.](image-url)
Recent Evaluation including IMC

- **CPAchecker** revision 40806
- Interpolants provided by **MathSAT5**
- Compared algorithms
  - IMC
  - PDR
  - BMC
  - $k$-Induction
  - Predicate abstraction
  - IMPACT
- Subset of *ReachSafety* from SV-COMP ’22
  - Safe: 4234 tasks
  - Unsafe: 1793 tasks
Quantile Plot: Safe Tasks

CPU time (s) vs. n-th fastest correct proofs for different methods:
- IMC
- PDR
- BMC
- k-Induction
- Predicate Abstraction
- Impact
Quantile Plot: Unsafe Tasks

![Quantile Plot](image)

- CPU time (s)
- n-th fastest correct alarms

Legend:
- IMC
- PDR
- BMC
- k-Induction
- Predicate Abstraction
- Impact

Dirk Beyer
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, $\text{IMPACT}$) scalable
- $\text{IMPACT}$ is lazy: explores the state space and finds bugs quicker
- Predicate Abstraction is eager: prunes irrelevant parts and finds proofs quicker
- IMC is competitive among polished SV approaches
SMT Solver Can Make a Difference

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.
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

\[ \text{Z3 with bitprecise arithmetic} \]

(B) Predicate Abstraction solves 3\% more tasks

\[ \text{MathSAT5 with linear arithmetic} \]

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

Technical details (e.g., choice of SMT theory) influence evaluation of algorithms
Comparison of SMT Solvers and Theories

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

- Which SMT solver should we use in a verifier?
- Which formula encoding?
- Which of these should we use for benchmarks in papers?
- Large study made possible by our framework
- Produced some interesting insights
- Resulted in change of default configuration of CPAchecker
Comparison of SMT Solvers and Theories

- Which SMT solver should we use in a verifier?
- Which formula encoding?
- Which of these should we use for benchmarks in papers?
- Large study made possible by our framework
- Produced some interesting insights
- Resulted in change of default configuration of CPAchecker
- Comparison using CPAchecker and Predicate CPA
- 5,594 verification tasks from SV-COMP’17
- 15 min time limit (CPU time), 15 GB memory limit
- Measured with BenchExec

Dirk Beyer
<table>
<thead>
<tr>
<th>BMC</th>
<th>$k$-Induction</th>
<th>IMPACT</th>
<th>Pred. Abs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MathSAT5</td>
<td>Princess</td>
<td>SMTInterpol</td>
<td>Z3</td>
</tr>
<tr>
<td></td>
<td>Bitprecise</td>
<td>Linear</td>
<td>Linear unsound</td>
</tr>
<tr>
<td></td>
<td>with Quantifiers</td>
<td>Quantifier-free</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arrays</td>
<td>UFs</td>
<td></td>
</tr>
</tbody>
</table>
Point of View: SMT Solvers

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

- Unsound linear encoding always the easiest (as expected)
Point of View: Theories and Encodings

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

$\Rightarrow$ MathSAT5 is really good with bitvectors.
$\Rightarrow$ Sound LIRA encoding rarely makes sense.
Point of View: Theories and Encodings

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

⇒ MathSAT5 is really good with bitvectors.
⇒ Sound LIRA encoding rarely makes sense.
Unsound linear encoding always the easiest (as expected)

Correctness as expected:
\[ BV > \text{sound LIRA} > \text{unsound LIRA} \]

Effectivity for \texttt{Z3} as expected:
\[ BV < \text{sound LIRA} < \text{unsound LIRA} \]

Effectivity for \texttt{MathSAT5}:
\[ \text{sound LIRA} < BV \approx \text{unsound LIRA} \]
(but BV needs more CPU time)

\[ \Rightarrow \text{MathSAT5 is really good with bitvectors.} \]
Unsound linear encoding always the easiest (as expected)

Correctness as expected:
BV > sound LIRA > unsound LIRA

Effectivity for Z3 as expected:
BV < sound LIRA < unsound LIRA

Effectivity for MathSAT5:
sound LIRA < BV ≈ unsound LIRA
(but BV needs more CPU time)

Effectivity for SMTInterpol:
sound LIRA ≪ unsound LIRA

⇒ MathSAT5 is really good with bitvectors.
Unsound linear encoding always the easiest (as expected)

Correctness as expected:
BV > sound LIRA > unsound LIRA

Effectivity for Z3 as expected:
BV < sound LIRA < unsound LIRA

Effectivity for MathSAT5:
sound LIRA < BV ≈ unsound LIRA
(but BV needs more CPU time)

Effectivity for SMTInterpol:
sound LIRA ≪ unsound LIRA

⇒ MathSAT5 is really good with bitvectors.
⇒ Sound LIRA encoding rarely makes sense.
Point of View: Algorithms

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

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

- Little difference with/without arrays/quantifiers
  - Arrays don’t hurt
    (though this might change once more complex array predicates are used)
Point of View: Arrays and Quantifiers

- Little difference with/without arrays/quantifiers
  ⇒ Arrays don’t hurt
  (though this might change
  once more complex array predicates are used)

- But quantifiers restrict solver choice
  (PRINCESS and Z3)
Choice of theories, solver, and encoding details affects comparisons of algorithms!

For now:
use MathSAT5 with bitvectors and arrays if possible

Possible problems for users: license, native binary
Next-best choice:
SMTInterpol with unsound linear arithmetic
No improvement of situation in sight


