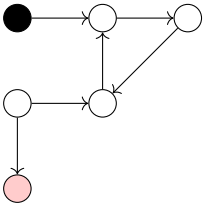
Giving Some Pointers for Abstraction-Based Model Checking

Levente Bajczi,
Dániel Szekeres, Csanád Telbisz, Vince Molnár

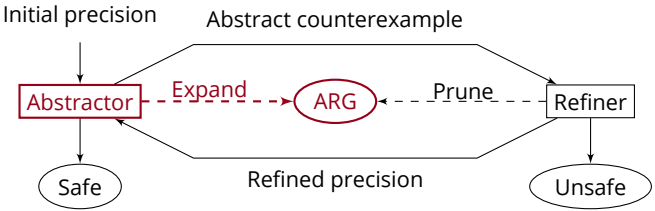
February 3, 2025

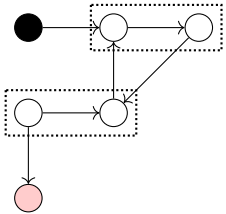
MŰEGYETEM 1782
BME FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS
ftsrg Critical Systems Research Group

Abstraction in Verification



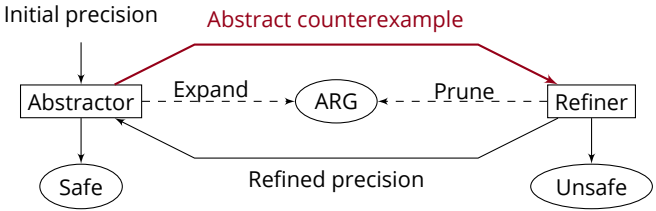
Abstraction in Verification

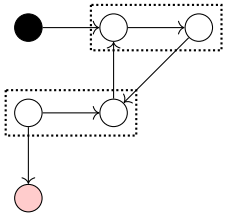




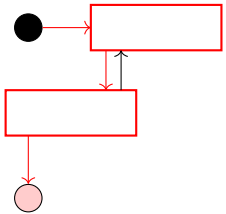
Abstract state space

Abstraction in Verification



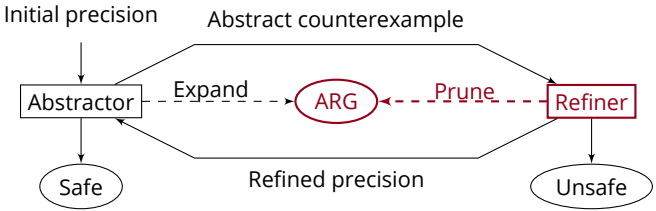


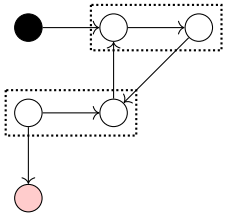
Abstract state space



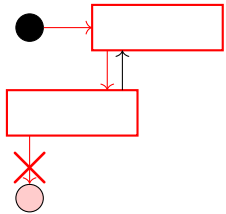
Abstract counterexample

Abstraction in Verification



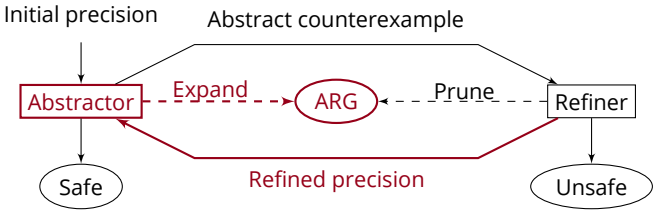


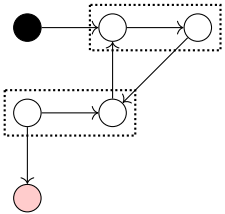
Abstract state space



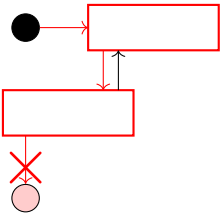
Abstract counterexample

Abstraction in Verification

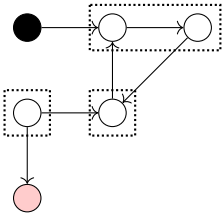




Abstract state space

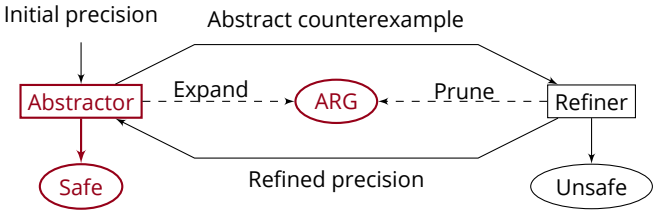


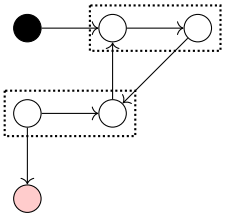
Abstract counterexample



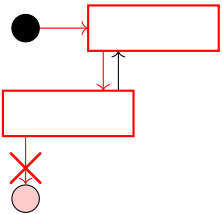
Refined abstract state space

Abstraction in Verification

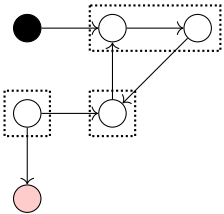




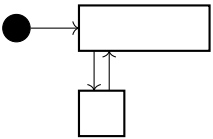
Abstract state space



Abstract counterexample

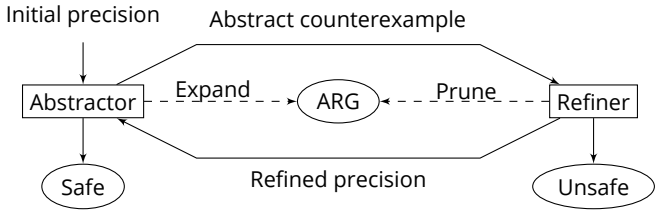


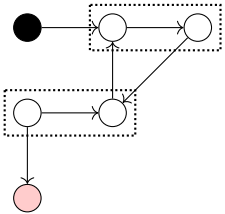
Refined abstract state space



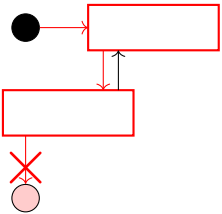
Safe ARG

Abstraction in Verification

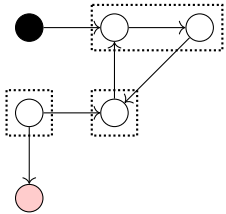




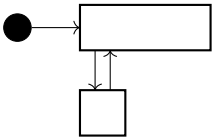
Abstract state space



Abstract counterexample



Refined abstract state space



Safe ARG

Pointers in Verification

Pointers in Verification

Ignoring Pointers

- Easy to implement
- Great for safety proofs (if lucky)
- Unsuitable for bug finding

Pointers in Verification

Ignoring Pointers

- Easy to implement
- Great for safety proofs (if lucky)
- Unsuitable for bug finding

Modeling Pointers Explicitly

- Moderately complex to implement
- Unsuitable for safety proofs
- Great for finding bugs

Pointers in Verification

Ignoring Pointers

- Easy to implement
- Great for safety proofs (if lucky)
- Unsuitable for bug finding

Modeling Pointers Explicitly

- Moderately complex to implement
- Unsuitable for safety proofs
- Great for finding bugs

Shape Analyses

- Complex to implement
- Great for safety proofs
- Great for finding bugs

Pointers in Verification

Ignoring Pointers

- Easy to implement
- Great for safety proofs (if lucky)
- Unsuitable for bug finding

Modeling Pointers Explicitly

- Moderately complex to implement
- Unsuitable for safety proofs
- Great for finding bugs

Symbolic Pointers

- Moderately complex to implement
- OK for safety proofs
- OK for finding bugs

Shape Analyses

- Complex to implement
- Great for safety proofs
- Great for finding bugs

Pointers in Verification

Ignoring Pointers

- Easy to impl
- Great proofs (if lucky)
- Unsuitable for bug finding

Rarely useful

Modeling Pointers Explicitly

- Moderately
- safety proofs
- for finding bugs

Widespread tool support

Symbolic Pointers

- Moderately complex to implement
- OK for safety proofs
- OK for finding bugs

Shape Analyses

- Complex to
- proofs
- Great for finding bugs

Plenty of research

Pointers in Verification

Ignoring Pointers

- Easy to impl
- Great proofs (if lucky)
- Unsuitable for bug finding

Rarely useful

Modeling Pointers Explicitly

- Moderately
- safety proofs
- for finding bugs

Widespread tool support

Symbolic Pointers

- Moderately complex to implement
- OK for safety proofs
- OK for finding bugs

Shape Analyses

- Complex to
- Gr proofs
- Great for finding bugs

Plenty of research

Summary of Contributions

Theoretical

- Precise, symbolic pointer analysis leveraging SMT solvers
- Extensible pointer analysis *plug-in* for CEGAR
- Examination of the effect of abstract domain choice

Summary of Contributions

Theoretical

- Precise, symbolic pointer analysis leveraging SMT solvers
- Extensible pointer analysis *plug-in* for CEGAR
- Examination of the effect of abstract domain choice

Practical

- Plug-In implementation in Θtheta
- Evaluation on software benchmarks

Pointers in Verification

```
1 int a;
2 int b;
3 a = 0; b = 0;
4 int cond = nondet();
5 int *c =  cond?&a:&b;
6 int *d = !cond?&a:&b;
7 *c = 1;
8 *d = 2;
9 assert(a==1||b==1);
```

Pointers in Verification

```
1 int a;
2 int b;
3 a = 0; b = 0;
4 int cond = nondet();
5 int *c = cond?&a:&b;
6 int *d = !cond?&a:&b;
7 *c = 1;
8 *d = 2;
9 assert(a==1||b==1);
```

Pointers in Verification

```
1 int a;
2 int b;
3 a = 0; b = 0;
4 int cond = nondet();
5 int *c = cond?&a:&b;
6 int *d = !cond?&a:&b;
7 *c = 1;
8 *d = 2;
9 assert(a==1||b==1);
```

Some nondeterministic input

Aliasing either a or b

Pointers in Verification

```
1 int a;
2 int b;
3 a = 0; b = 0;
4 int cond = nondet();
5 int *c = cond?&a:&b;
6 int *d = !cond?&a:&b;
7 *c = 1;
8 *d = 2;
9 assert(a==1||b==1);
```

Some nondeterministic input

Aliasing either a or b

Never the same as c

Pointers in Verification

```
1 int a;
2 int b;
3 a = 0; b = 0;
4 int cond = nondet();
5 int *c = cond?&a:&b;
6 int *d = !cond?&a:&b;
7 *c = 1;
8 *d = 2;
9 assert(a==1||b==1);
```

Some nondeterministic input

Aliasing either a or b

Never the same as c

At least a or b was aliased by c

Pointers in Verification

Reference

```
1 int a;
2 int b;
3 a = 0; b = 0;
4 int cond = nondet();
5 int *c = cond?&a:&b;
6 int *d = !cond?&a:&b;
7 *c = 1;
8 *d = 2;
9 assert(a==1||b==1);
```

At least a or b was aliased by c

Pointers in Verification

```
1 int a;
2 int b;
3 a = 0; b = 0;
4 int cond = nondet();
5 int *c = cond?&a:&b;
6 int *d = !cond?&a:&b;
7 *c = 1;
8 *d = 2;
9 assert(a==1||b==1);
```

Reference

Dereference

At least a or b was aliased by c

Pointers in Verification

```
1 int a;
2 int b;
3 a = 0; b = 0;
4 int cond = nondet();
5 int *c = cond?&a:&b;
6 int *d = !cond?&a:&b;
7 *c = 1;
8 *d = 2;
9 assert(a==1||b==1);
```

Simplicity!

Reference

Dereference

At least a or b was aliased by c

Pointers in Verification

Simplicity!

~~Reference~~

Dereference

```
1 int a;
2 int b;
3 a = 0; b = 0;
4 int cond = nondet();
5 int *c = cond?&a:&b;
6 int *d = !cond?&a:&b;
7 *c = 1;
8 *d = 2;
9 assert(a==1||b==1);
```

At least a or b was aliased by c

Pointers in Verification

~~Reference~~

Dereference

```
1 int a;
2 int b;
3 a = 0; b = 0;
4 int cond = nondet();
5 int *c = cond?&a:&b;
6 int *d = !cond?&a:&b;
7 *c = 1;
8 *d = 2;
9 assert(a==1||b==1);
```

```
1 int *a = malloc();
2 int *b = malloc();
3 *a = 0; *b = 0;
4 int cond = nondet();
5 int *c = cond?a:b;
6 int *d = !cond?a:b;
7 *c = 1;
8 *d = 2;
9 assert(*a==1||*b==1);
```

Pointers in Verification

Reference

Dereference

```
1 int a;
2 int b;
3 a = 0; b = 0;
4 int cond = nondet();
5 int *c = cond?&a:&b;
6 int *d = !cond?&a:&b;
7 *c = 1;
8 *d = 2;
9 assert(a==1||b==1);
```

```
1 int *a = malloc();
2 int *b = malloc();
3 *a = 0; *b = 0;
4 int cond = nondet();
5 int *c = cond?a:b;
6 int *d = !cond?a:b;
7 *c = 1;
8 *d = 2;
9 assert(*a==1||*b==1);
```

Contribution 1: Reference Elimination

Pointers in Verification

```
1 (declare-fun deref (Int Int Int) Int)
  (declare-fun a () Int) (declare-fun b () Int) (declare-fun c () Int)
  (declare-fun d () Int) (declare-fun cond () Bool)

2 (assert (= a 0))
  (assert (= b 1))

3 (assert (= (deref a 0 1) 0)) (assert (= (deref b 0 1) 0))

4 (assert (= c (ite cond a b))) (assert (= d (ite (not cond) a b)))

5 (assert
      (let ((idx1 (+ 1 (ite (= c 1) 1 (ite (= c 0) 1 0)))))
      (let ((idx2 (+ 1 (ite (= d c) idx1 (ite (= d 1) 1 (ite (= d 0) 1 0))))))
      (and (= (deref c 0 idx1) 1)
           (= (deref d 0 idx2) 2)
6          (let ((idx3 (ite (= 0 d) idx2 (ite (= 0 c) idx1 1))))
           (let ((idx4 (ite (= 1 d) idx2 (ite (= 1 c) idx1 1))))
           (or (= (deref a 0 idx3) 1)
               (= (deref b 0 idx4) 1)))))))
```

```
1 int *a = malloc();
2 int *b = malloc();
3 *a = 0; *b = 0;
4 int cond = nondet();
5 int *c =  cond?a:b;
6 int *d = !cond?a:b;
7 *c = 1;
8 *d = 2;
9 assert(*a==1||*b==1);
```

Pointers in Verification

deref
UF

```
1 (declare-fun deref (Int Int Int) Int)
  (declare-fun a () Int) (declare-fun b () Int) (declare-fun c () Int)
  (declare-fun d () Int) (declare-fun cond () Bool)

2 (assert (= a 0))
  (assert (= b 1))

3 (assert (= (deref a 0 1) 0)) (assert (= (deref b 0 1) 0))

4 (assert (= c (ite cond a b))) (assert (= d (ite (not cond) a b)))

5 (assert
    (let ((idx1 (+ 1 (ite (= c 1) 1 (ite (= c 0) 1 0)))))
    (let ((idx2 (+ 1 (ite (= d c) idx1 (ite (= d 1) 1 (ite (= d 0) 1 0))))))
    (and (= (deref c 0 idx1) 1)
         (= (deref d 0 idx2) 2)
6        (let ((idx3 (ite (= 0 d) idx2 (ite (= 0 c) idx1 1))))
         (let ((idx4 (ite (= 1 d) idx2 (ite (= 1 c) idx1 1))))
         (or (= (deref a 0 idx3) 1)
             (= (deref b 0 idx4) 1)))))))))
```

```
1 int *a = malloc();
2 int *b = malloc();
3 *a = 0; *b = 0;
4 int cond = nondet();
5 int *c = cond?a:b;
6 int *d = !cond?a:b;
7 *c = 1;
8 *d = 2;
9 assert(*a==1||*b==1);
```

Dereference:
(ptr, offset, idx) -> value

Pointers in Verification

deref UF

Unique address

```
1 (declare-fun deref (Int Int Int) Int)
  (declare-fun a () Int) (declare-fun b () Int) (declare-fun c () Int)
  (declare-fun d () Int) (declare-fun cond () Bool)

2 (assert (= a 0))
  (assert (= b 1))

3 (assert (= (deref a 0 1) 0)) (assert (= (deref b 0 1) 0))

4 (assert (= c (ite cond a b))) (assert (= d (ite (not cond) a b)))

5 (assert
      (let ((idx1 (+ 1 (ite (= c 1) 1 (ite (= c 0) 1 0)))))
      (let ((idx2 (+ 1 (ite (= d c) idx1 (ite (= d 1) 1 (ite (= d 0) 1 0))))))
      (and (= (deref c 0 idx1) 1)
           (= (deref d 0 idx2) 2)
6          (let ((idx3 (ite (= 0 d) idx2 (ite (= 0 c) idx1 1))))
           (let ((idx4 (ite (= 1 d) idx2 (ite (= 1 c) idx1 1))))
           (or (= (deref a 0 idx3) 1)
               (= (deref b 0 idx4) 1)))))))))
```

```
1 int *a = malloc();
2 int *b = malloc();
3 *a = 0; *b = 0;
4 int cond = nondet();
5 int *c =  cond?a:b;
6 int *d = !cond?a:b;
7 *c = 1;
8 *d = 2;
9 assert(*a==1||*b==1);
```

Dereference:
(ptr, offset, idx) -> value

Pointers in Verification

deref UF

Unique address

Index calculation

```
1 (declare-fun deref (Int Int Int) Int)
  (declare-fun a () Int) (declare-fun b () Int) (declare-fun c () Int)
  (declare-fun d () Int) (declare-fun cond () Bool)

2 (assert (= a 0))
  (assert (= b 1))

3 (assert (= (deref a 0 1) 0)) (assert (= (deref b 0 1) 0))

4 (assert (= c (ite cond a b))) (assert (= d (ite (not cond) a b)))

5 (assert
    (let ((idx1 (+ 1 (ite (= c 1) 1 (ite (= c 0) 1 0)))))
    (let ((idx2 (+ 1 (ite (= d c) idx1 (ite (= d 1) 1 (ite (= d 0) 1 0))))))
    (and (= (deref c 0 idx1) 1)
         (= (deref d 0 idx2) 2)
6        (let ((idx3 (ite (= 0 d) idx2 (ite (= 0 c) idx1 1))))
         (let ((idx4 (ite (= 1 d) idx2 (ite (= 1 c) idx1 1))))
         (or (= (deref a 0 idx3) 1)
             (= (deref b 0 idx4) 1)))))))))
```

```
1 int *a = malloc();
2 int *b = malloc();
3 *a = 0; *b = 0;
4 int cond = nondet();
5 int *c =  cond?a:b;
6 int *d = !cond?a:b;
7 *c = 1;
8 *d = 2;
9 assert(*a==1||*b==1);
```

Dereference:
(ptr, offset, idx) -> value

Pointers in Verification

ptr	*off*	*idx*	(*deref ptr off idx*)

```
1 int *a = malloc();
2 int *b = malloc();
3 *a = 0; *b = 0;
4 int cond = nondet();
5 int *c =  cond?a:b;
6 int *d = !cond?a:b;
7 *c = 1;
8 *d = 2;
9 assert(*a==1||*b==1);
```

Dereference:
(ptr, offset, idx) -> value

Pointers in Verification

ptr	*off*	*idx*	(*deref ptr off idx*)
0(a)	0	1	0
1(b)	0	1	0

```
1 int *a = malloc();
2 int *b = malloc();
3 *a = 0; *b = 0;
4 int cond = nondet();
5 int *c =  cond?a:b;
6 int *d = !cond?a:b;
7 *c = 1;
8 *d = 2;
9 assert(*a==1||*b==1);
```

Dereference:
(ptr, offset, idx) -> value

Pointers in Verification

ptr	*off*	*idx*	(*deref ptr off idx*)
0(a)	0	1	0
1(b)	0	1	0

$c == a$

$d == b$

ptr	*off*	*idx*	(*deref ptr off idx*)
0(a)	0	1	0
1(b)	0	1	0

$c == b$

$d == a$

```
1 int *a = malloc();
2 int *b = malloc();
3 *a = 0; *b = 0;
4 int cond = nondet();
5 int *c =  cond?a:b;
6 int *d = !cond?a:b;
7 *c = 1;
8 *d = 2;
9 assert(*a==1||*b==1);
```

Dereference:
(ptr, offset, idx) -> value

Pointers in Verification

ptr	*off*	*idx*	(*deref ptr off idx*)
0(a)	0	1	0
1(b)	0	1	0
0(a, c)	0	2	**1**

$c == a$

$d == b$

ptr	*off*	*idx*	(*deref ptr off idx*)
0(a)	0	1	0
1(b)	0	1	0
1(b, c)	0	2	**1**

$c == b$

$d == a$

```
1 int *a = malloc();
2 int *b = malloc();
3 *a = 0; *b = 0;
4 int cond = nondet();
5 int *c =  cond?a:b;
6 int *d = !cond?a:b;
7 *c = 1;
8 *d = 2;
9 assert(*a==1||*b==1);
```

Dereference:
(ptr, offset, idx) -> value

Pointers in Verification

ptr	*off*	*idx*	(*deref ptr off idx*)
0(a)	0	1	0
1(b)	0	1	0
0(a, c)	0	2	**1**
1(b, d)	0	2	**2**

$c == a$

$d == b$

ptr	*off*	*idx*	(*deref ptr off idx*)
0(a)	0	1	0
1(b)	0	1	0
1(b, c)	0	2	**1**
0(a, d)	0	2	**2**

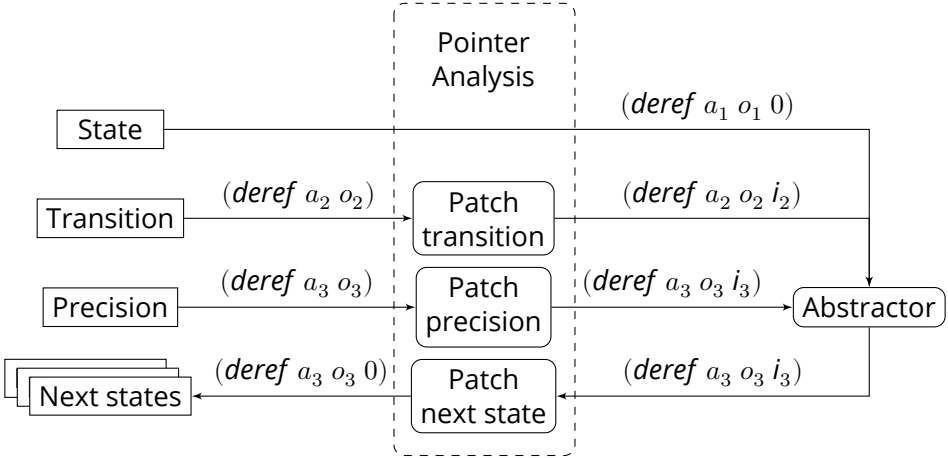
$c == b$

$d == a$

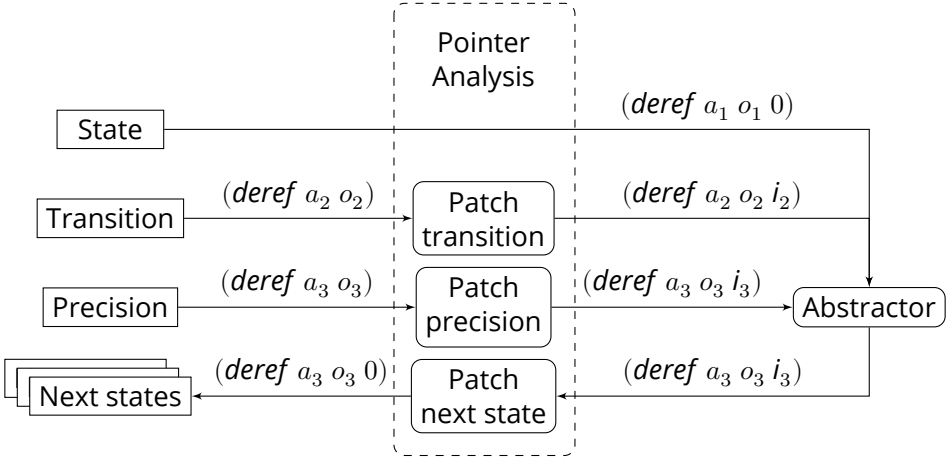
```
1 int *a = malloc();
2 int *b = malloc();
3 *a = 0; *b = 0;
4 int cond = nondet();
5 int *c =  cond?a:b;
6 int *d = !cond?a:b;
7 *c = 1;
8 *d = 2;
9 assert(*a==1||*b==1);
```

Dereference:
(ptr, offset, idx) -> value

CEGAR Plug-In



CEGAR Plug-In



Contribution 2: CEGAR Plug-In

Experiments

Implementation in Θ*theta*

- As an independent plug-in
- Supports any abstract domain and all refinement strategies

? Better than *havoc*?

? Which domain is best?

Experiments

Implementation in Θ*theta*

- As an independent plug-in
- Supports any abstract domain and all refinement strategies
- ? Better than *havoc*?
- ? Which domain is best?

Experiments on Benchmarks

- 633 *reachability* tasks from SV-Benchmarks
- BenchExec for accuracy

Experiments

Implementation in Θ*theta*

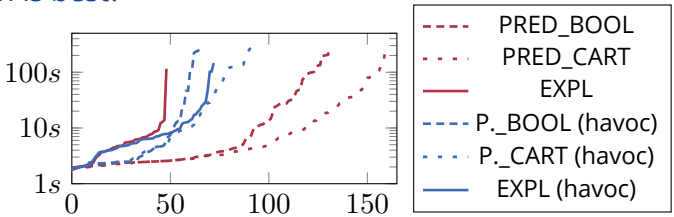
- As an independent plug-in
- Supports any abstract domain and all refinement strategies

? Better than *havoc*?

? Which domain is best?

Experiments on Benchmarks

- 633 *reachability* tasks from SV-Benchmarks
- BenchExec for accuracy



Experiments

Implementation in Θ*theta*

- As an independent plug-in
- Supports any abstract domain and all refinement strategies

? Better than *havoc*? **Yes!**

? Which domain is best? **Predicate**

Experiments on Benchmarks

- 633 *reachability* tasks from SV-Benchmarks
- BenchExec for accuracy