Configurable Software Verification based on Slicing Abstractions

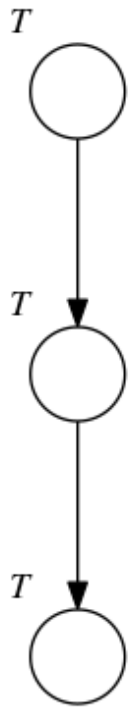
Martin Spießl

LMU Munich

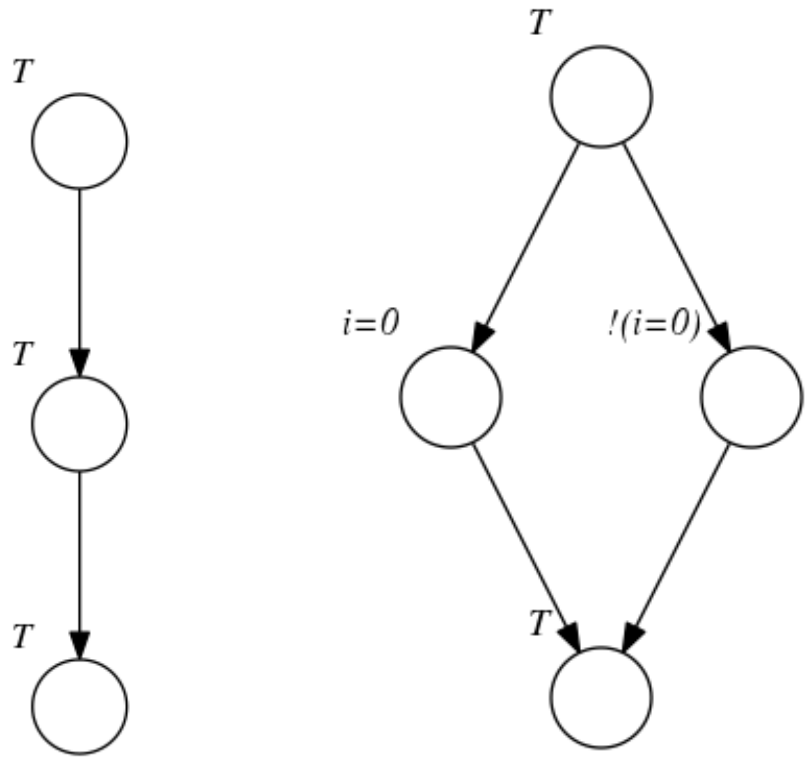
Outline

- Basic Idea
- Example Program
  - Splitting via Interpolants (Kojak)
  - Slicing Abstractions (SLAB)
- Fitting Kojak into the CPAchecker Framework
  - CPA Algorithm
  - CEGAR Algorithm
  - Adjustable-Block Encoding
- Fitting SLAB into the CPAchecker Framework
  - Flexible-Block Encoding
- Evaluation
- Summary

Slicing Abstractions Idea

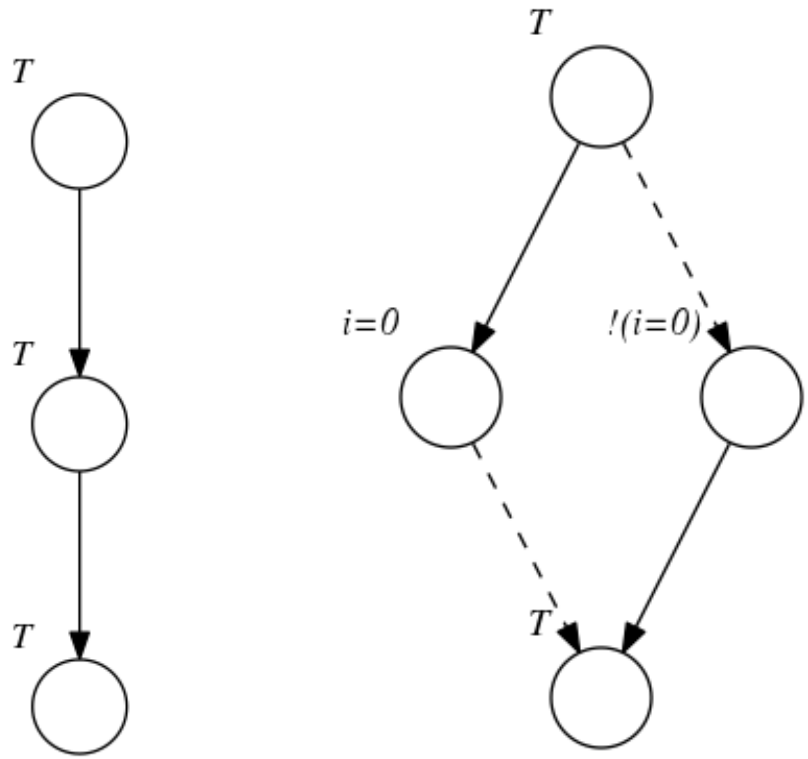


Slicing Abstractions Idea



- Split abstraction state into two states
- Disjunction of the splitted states represent the same concrete states as in the original state $\Rightarrow$ soundness
- Incoming & outgoing edges have to be copied

Slicing Abstractions Idea



- Split abstraction state into two states
- Disjunction of the splitted states represent the same concrete states as in the original state $\Rightarrow$ soundness
- Incoming & outgoing edges have to be copied
- Slice by removing infeasible edges
- disconnected subgraphs can be removed

Slicing Abstractions in Software Model Checking

(2007) Slicing Abstractions

- program counter (if present) tracked symbolically
- motivated by predicate abstraction
- Implementation: SLAB

(2012) Splitting via Interpolants

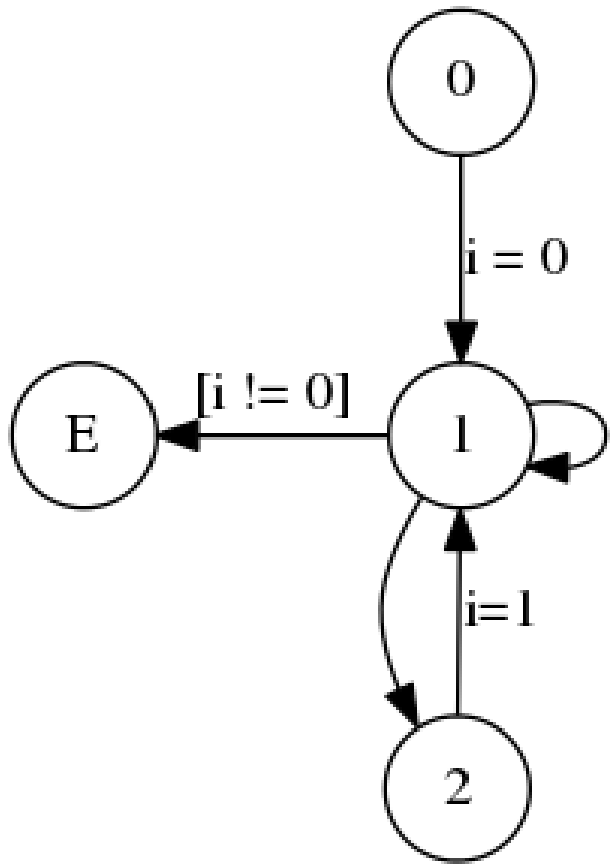
- makes use of control-flow graph
  $\Rightarrow$ symbolic program counter not needed
- motivated by SLAB
- uses Large-Block Encoding
- Implementation: Ultimate Kojak

Example Program

example.c

```
0  int i = 0;
1  do {
2     assert i == 0;
3     if (*) {
4        i = 1;
5     }
6  } while (true);
```

CFA



Splitting via Interpolants (Kojak)

Basic Steps:

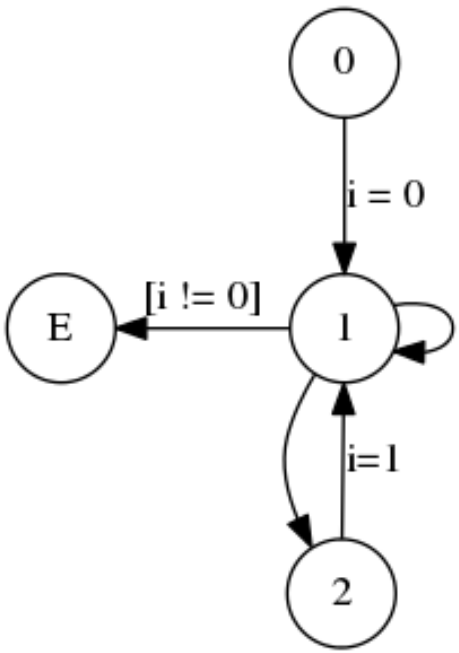
- **Split:** Split states along the error path using interpolants
- **Slice:** Remove infeasible edges

Termination

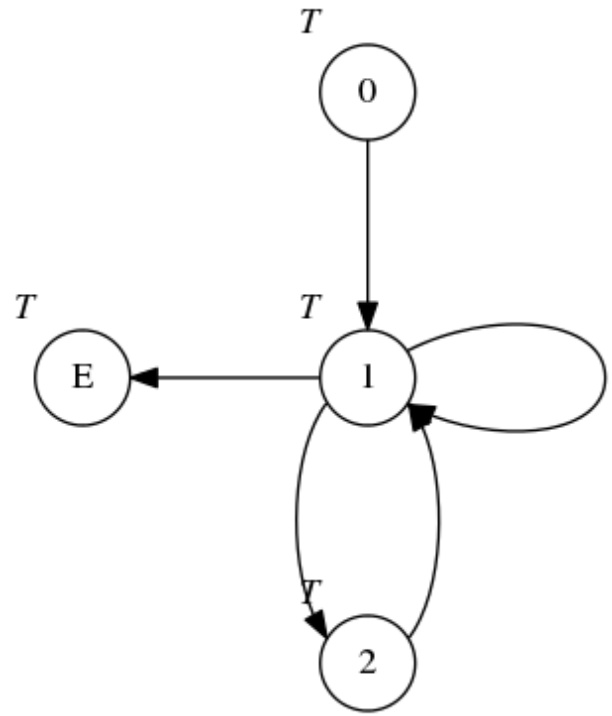
- when finding a real counterexample: return false
- when all error states are disconnected: return true

Splitting via Interpolants (Kojak)

CFA

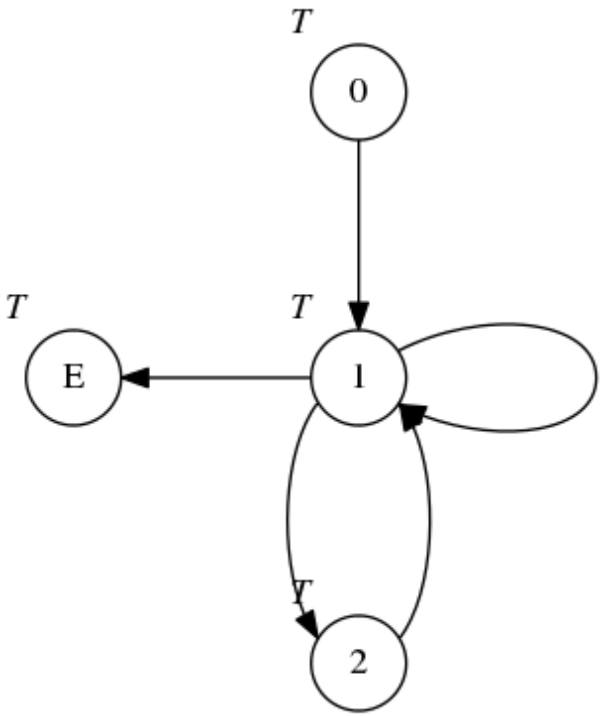


Initial Abstract Model

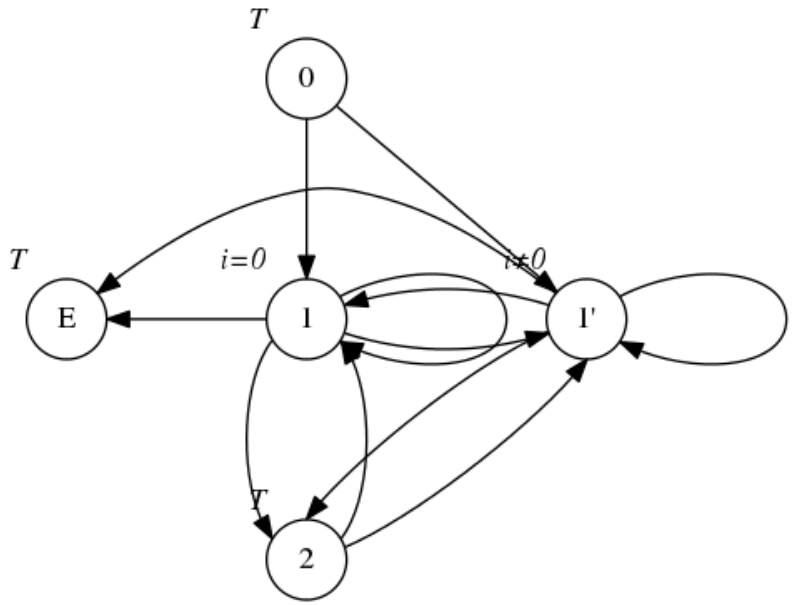


Splitting via Interpolants (Kojak)

Initial Abstract Model

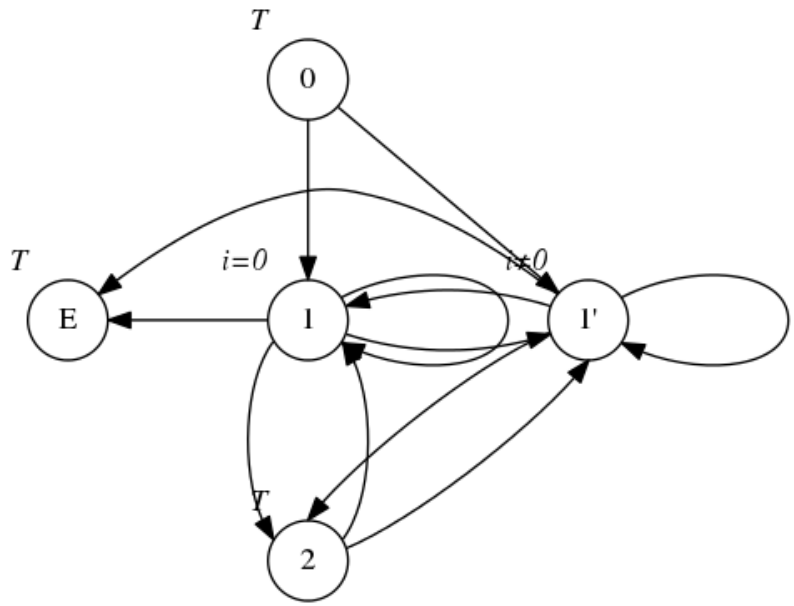


Split

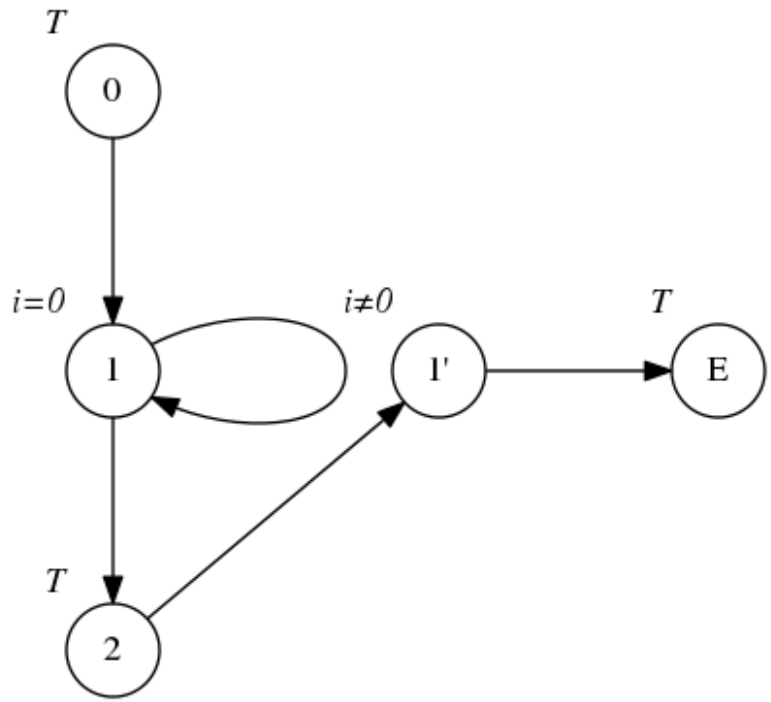


Splitting via Interpolants (Kojak)

Split



Slice



Slicing Abstractions (SLAB)

Basic Steps:

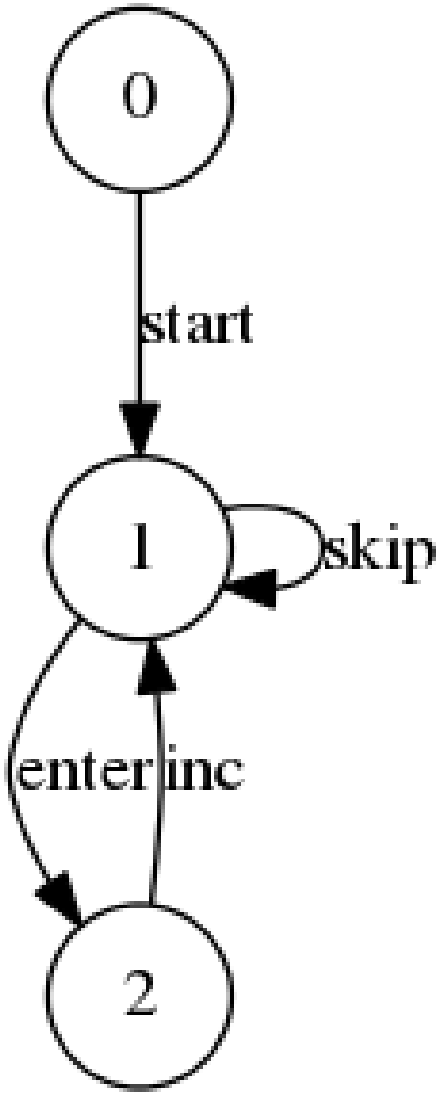
- Similar to Kojak, but treat program counter symbolically
- Special initial abstract model with predicates $init$ and $error$

Termination

- when finding a feasible counterexample: return false
- when no states with predicate $error$ are left: return true

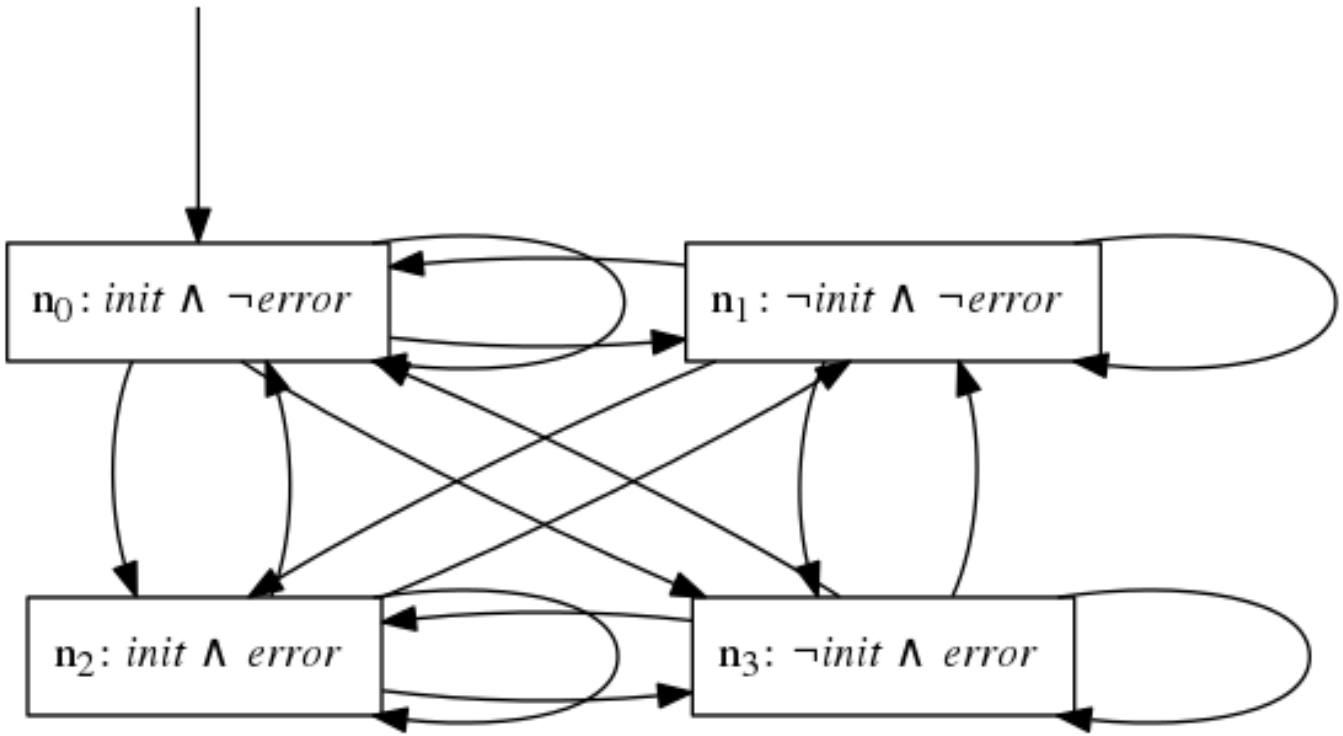
Slicing Abstractions (SLAB)

CFA

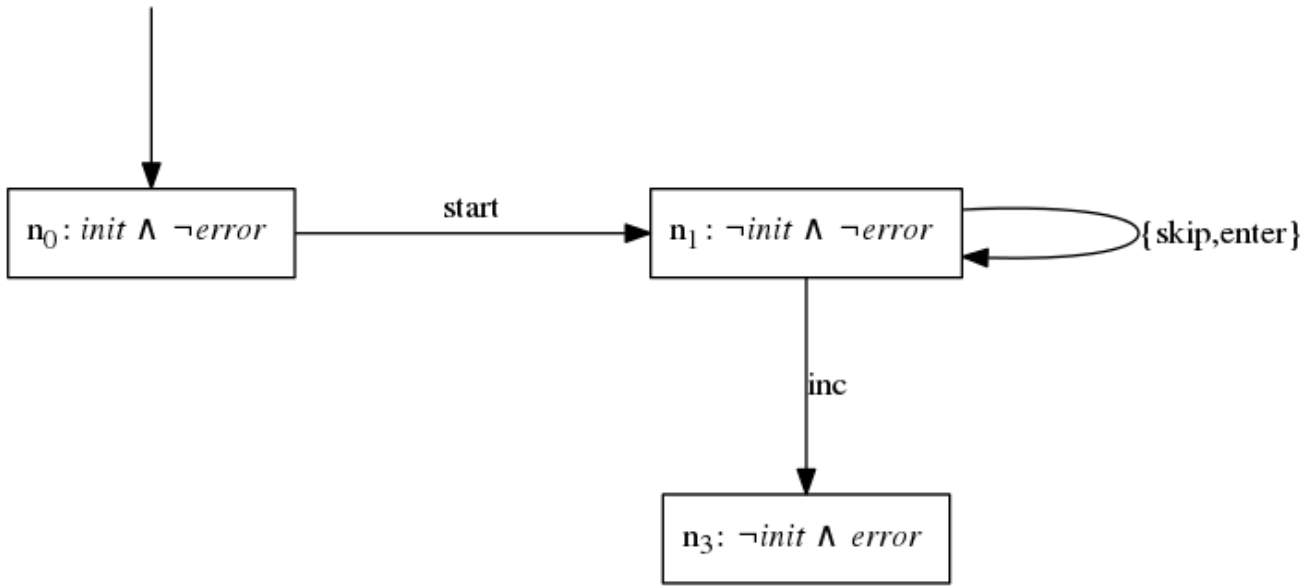


init	$pc = 0$
error	$pc = 1 \wedge i \neq 0$
start	$pc = 0 \wedge pc' = 1 \wedge i' = 0$
skip	$pc = 1 \wedge pc' = 1$
enter	$pc = 1 \wedge pc' = 2$
inc	$pc = 2 \wedge pc' = 1 \wedge i' = 1$

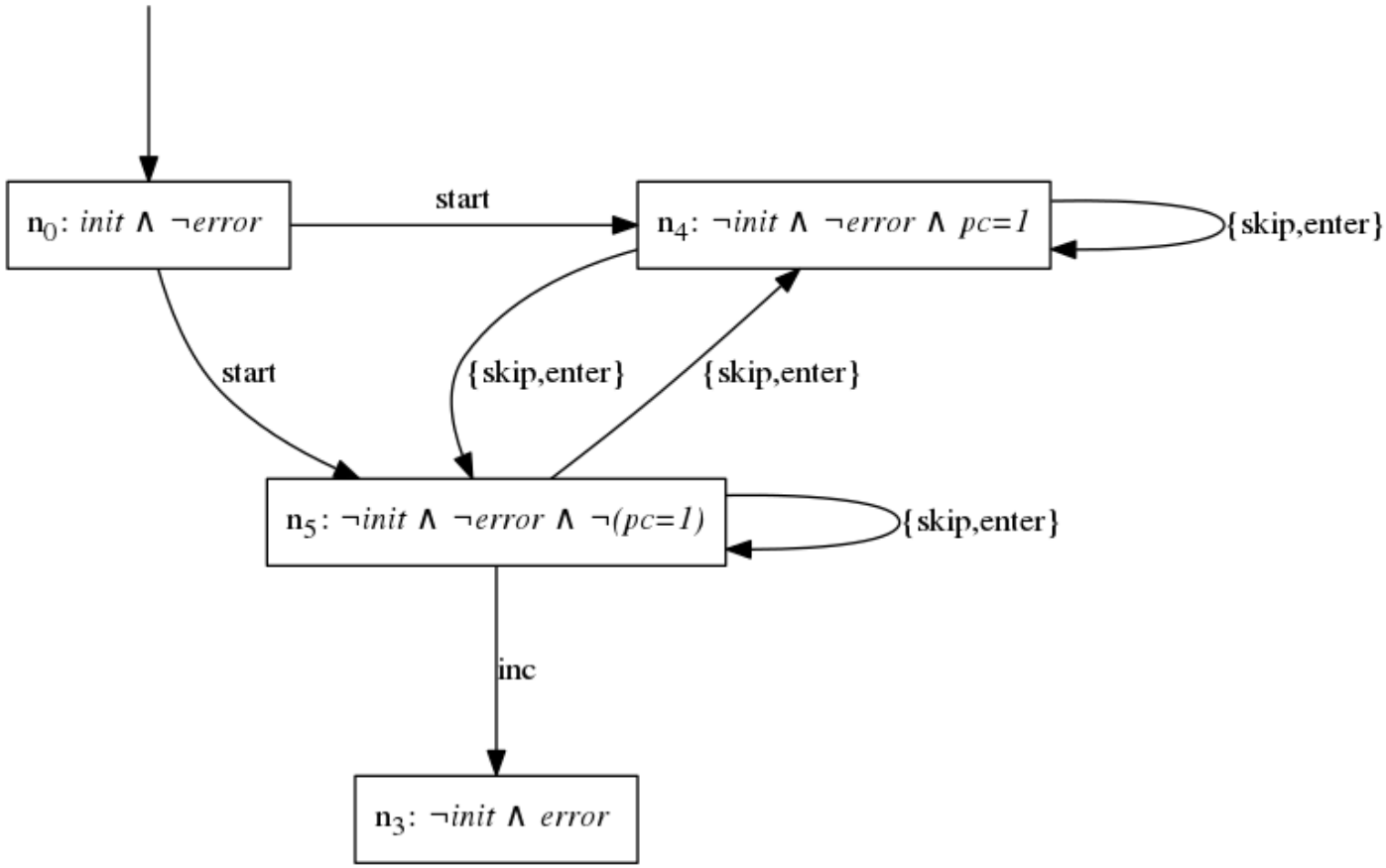
Slicing Abstractions (SLAB)



Slicing Abstractions (SLAB)



Slicing Abstractions (SLAB)



Implementation of Kojak in CPAchecker

- Entirely new CPA is NOT needed, instead we reuse the CPA of predicate abstraction with adjustable-block encoding (CPA_{ABE})
- Path formulas need to be constructed separately
- Special choice of CPA- and CEGAR algorithm inputs
- Refinement procedure that implements splitting and slicing
- Refinement operates on abstract reachability graph (ARG)
  $\Rightarrow$ lots of graph manipulation, esp. for ABE

CPA* Algorithm Setup for Kojak

```
1:  while waitlist ≠ ∅ do
2:      choose e from waitlist
3:      waitlist := waitlist \ {e}
4:      for all e' with e⇝e' do
5:          for all e'' ∈ reached do
6:              // combine with existing abstract state
7:              e_new := merge(e, e'')
8:              if e_new ≠ e'' then
9:                  waitlist := (waitlist ∪ {e_new}) \ {e''}
10:                 reached := (reached ∪ {e_new}) \ {e''}
11:         if ¬stop(e', reached) then
12:             if abort(e') then
13:                 return (reached, waitlist)
14:             waitlist := waitlist ∪ {e'}
15:             reached := reached ∪ {e'}
16: return (reached, waitlist)
```

* original algorithm also contains precision, removed here for brevity

CPA* Algorithm Setup for Kojak

```
1:  while waitlist ≠ ∅ do
2:      choose e from waitlist
3:      waitlist := waitlist \ {e}
4:      for all e' with e⇝e' do
5:          for all e'' ∈ reached do
6:              // combine with existing abstract state
7:              e_new := merge(e, e'')
8:              if e_new ≠ e'' then
9:                  waitlist := (waitlist ∪ {e_new}) \ {e''}
10:                 reached := (reached ∪ {e_new}) \ {e''}
11:         if ¬stop(e', reached) then
12:             if abort(e') then
13:                 return (reached, waitlist)
14:             waitlist := waitlist ∪ {e'}
15:             reached := reached ∪ {e'}
16: return (reached, waitlist)
```

- merge operator: Merge states at same location in the ARG until parent sets are equal

* original algorithm also contains precision, removed here for brevity

CPA* Algorithm Setup for Kojak

```
1:  while waitlist ≠ ∅ do
2:      choose e from waitlist
3:      waitlist := waitlist \ {e}
4:      for all e' with e⇝e' do
5:          for all e'' ∈ reached do
6:              // combine with existing abstract state
7:              e_new := merge(e, e'')
8:              if e_new ≠ e'' then
9:                  waitlist := (waitlist ∪ {e_new}) \ {e''}
10:                 reached := (reached ∪ {e_new}) \ {e''}
11:         if ¬stop(e', reached) then
12:             if abort(e') then
13:                 return (reached, waitlist)
14:             waitlist := waitlist ∪ {e'}
15:             reached := reached ∪ {e'}
16: return (reached, waitlist)
```

- merge operator: Merge states at same location in the ARG until parent sets are equal
- $stop_{sep}$ is sufficient, since all states have abstraction formula $\top$

* original algorithm also contains precision, removed here for brevity

CPA* Algorithm Setup for Kojak

```
1:  while waitlist ≠ ∅ do
2:      choose e from waitlist
3:      waitlist := waitlist \ {e}
4:      for all e' with e⇝e' do
5:          for all e'' ∈ reached do
6:              // combine with existing abstract state
7:              e_new := merge(e, e'')
8:              if e_new ≠ e'' then
9:                  waitlist := (waitlist ∪ {e_new}) \ {e''}
10:                 reached := (reached ∪ {e_new}) \ {e''}
11:         if ¬stop(e', reached) then
12:             if abort(e') then
13:                 return (reached, waitlist)
14:             waitlist := waitlist ∪ {e'}
15:             reached := reached ∪ {e'}
16: return (reached, waitlist)
```

- merge operator: Merge states at same location in the ARG until parent sets are equal
- $stop_{sep}$ is sufficient, since all states have abstraction formula $\top$
- Global Refinement: abort (e') always returns $\bot$

* original algorithm also contains precision, removed here for brevity

Slicing Abstractions as CEGAR Refinement Strategy

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$,
an initial abstract state $e_0 \in E$,
a function refine : $\mathcal{P}(E) \times \mathcal{P}(E) \to \mathcal{P}(E) \times \mathcal{P}(E)$,
a function abort : $E \to \mathbb{B}$
a function isTargetState : $E \to \mathbb{B}$

1: **loop**
2: (reached, waitlist) := CPA*($\mathbb{D}$, reached, waitlist, abort)
3: **if** $\exists e \in$ reached : isTargetState(e) **then**
4: (reached, waitlist) := refine(reached, waitlist)
5: **if** $\exists e \in$ reached : isTargetState(e) **then**
6: **return** $\mathbf{false}$
7: **else**
8: **return** $\mathbf{true}$

Slicing Abstractions as CEGAR Refinement Strategy

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$,
an initial abstract state $e_0 \in E$,
a function refine : $\mathcal{P}(E) \times \mathcal{P}(E) \to \mathcal{P}(E) \times \mathcal{P}(E)$,
a function abort : $E \to \mathbb{B}$
a function isTargetState : $E \to \mathbb{B}$

1: **loop**
2: (reached, waitlist) := CPA*($\mathbb{D}$, reached, waitlist, abort)
3: **if** $\exists e \in$ reached : isTargetState(e) **then**
4: (reached, waitlist) := refine(reached, waitlist)
5: **if** $\exists e \in$ reached : isTargetState(e) **then**
6: **return** $\mathbf{false}$
7: **else**
8: **return** $\mathbf{true}$

- Formulate splitting and slicing as CEGAR refinement strategy (called in refine)

Slicing Abstractions as CEGAR Refinement Strategy

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$,
an initial abstract state $e_0 \in E$,
a function refine : $\mathcal{P}(E) \times \mathcal{P}(E) \to \mathcal{P}(E) \times \mathcal{P}(E)$,
a function abort : $E \to \mathbb{B}$
a function isTargetState : $E \to \mathbb{B}$

1: **loop**
2: (reached, waitlist) := CPA*($\mathbb{D}$, reached, waitlist, abort)
3: **if** $\exists e \in$ reached : isTargetState(e) **then**
4: (reached, waitlist) := refine(reached, waitlist)
5: **if** $\exists e \in$ reached : isTargetState(e) **then**
6: **return** $\mathbf{false}$
7: **else**
8: **return** $\mathbf{true}$

- Formulate splitting and slicing as CEGAR refinement strategy (called in refine)
- Strategy repeats splitting and slicing until feasible counterexample is found or all error states have been removed

Slicing Abstractions as CEGAR Refinement Strategy

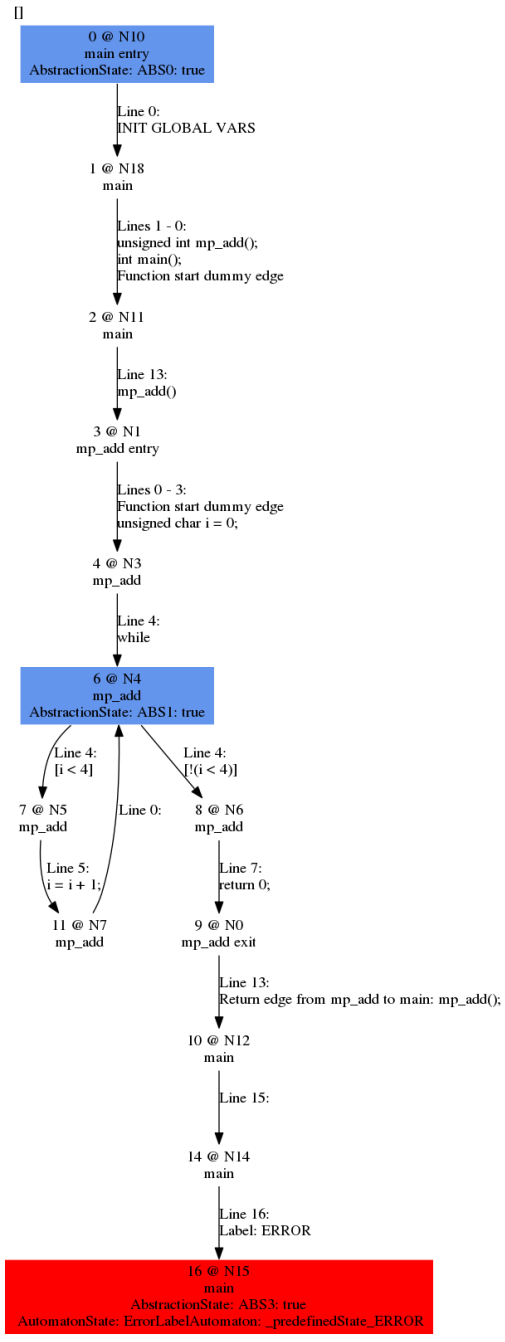
Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$,
an initial abstract state $e_0 \in E$,
a function refine : $\mathcal{P}(E) \times \mathcal{P}(E) \to \mathcal{P}(E) \times \mathcal{P}(E)$,
a function abort : $E \to \mathbb{B}$
a function isTargetState : $E \to \mathbb{B}$

1: **loop**
2: (reached, waitlist) := CPA*($\mathbb{D}$, reached, waitlist, abort)
3: **if** $\exists e \in$ reached : isTargetState(e) **then**
4: (reached, waitlist) := refine(reached, waitlist)
5: **if** $\exists e \in$ reached : isTargetState(e) **then**
6: **return** $\mathbf{false}$
7: **else**
8: **return** $\mathbf{true}$

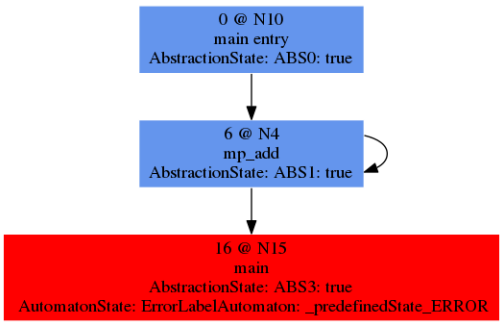
- Formulate splitting and slicing as CEGAR refinement strategy (called in refine)
- Strategy repeats splitting and slicing until feasible counterexample is found or all error states have been removed
- $\Rightarrow$ CPA* and refine will only be called once in this setup

Adjustable-Block Encoding

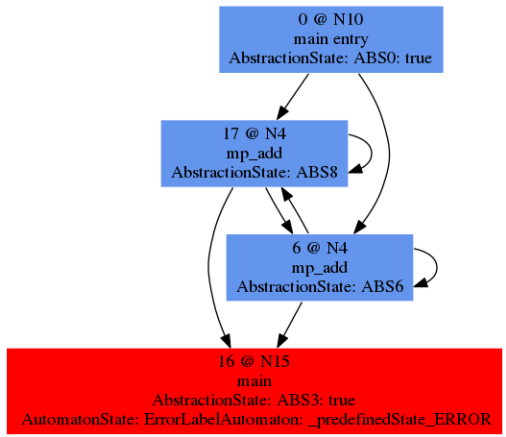
- ABE as flexible replacement for LBE (used by Kojak)
- PredicateCPA is already used to store abstraction formulas
  $\Rightarrow$ ABE from PredicateCPA can be reused
- However: block formulas from PredicateCPA cannot be used because the abstraction states in the ARG do not form a tree (previous abstraction state will be ambigious)
  $\Rightarrow$ dynamically recalculate them
- Loss of tree shape in ARG causes other problems
- Non-abstraction states have to be copied when splitting states (example on next slide)


[]
0 @ N10
main entry
AbstractionState: ABS0: true
Line 0:
INIT GLOBAL VARS
1 @ N18
main
Lines 1 - 0:
unsigned int mp_add();
int main();
Function start dummy edge
2 @ N11
main
Line 13:
mp_add()
3 @ N1
mp_add entry
Lines 0 - 3:
Function start dummy edge
unsigned char i = 0;
4 @ N3
mp_add
Line 4:
while
6 @ N4
mp_add
AbstractionState: ABS1: true
Line 4:
[i < 4]
Line 4:
[!(i < 4)]
7 @ N5
mp_add
8 @ N6
mp_add
Line 0:
Line 5:
i = i + 1;
Line 7:
return 0;
11 @ N7
mp_add
9 @ N0
mp_add exit
Line 13:
Return edge from mp_add to main: mp_add();
10 @ N12
main
Line 15:
14 @ N14
main
Line 16:
Label: ERROR
16 @ N15
main
AbstractionState: ABS3: true
AutomatonState: ErrorLabelAutomaton: _predefinedState_ERROR
In Refinement after Splitting
0 @ N10
main entry
AbstractionState: ABS0: true
Line 0:
INIT GLOBAL VARS
Line 0:
INIT GLOBAL VARS
28 @ N18
main
1 @ N18
main
Lines 1 - 0:
unsigned int mp_add();
int main();
Function start dummy edge
Lines 1 - 0:
unsigned int mp_add();
int main();
Function start dummy edge
29 @ N11
main
2 @ N11
main
Line 13:
mp_add()
Line 13:
mp_add()
30 @ N1
mp_add entry
3 @ N1
mp_add entry
Lines 0 - 3:
Function start dummy edge
unsigned char i = 0;
Lines 0 - 3:
Function start dummy edge
unsigned char i = 0;
31 @ N3
mp_add
4 @ N3
mp_add
Line 4:
while
17 @ N4
mp_add
AbstractionState: ABS8
Line 4:
[i < 4]
Line 4:
[i < 4]
Line 4:
[!(i < 4)]
Line 4:
while
22 @ N5
mp_add
Line 0:
24 @ N5
mp_add
18 @ N6
mp_add
Line 5:
i = i + 1;
Line 5:
i = i + 1;
Line 7:
return 0;
23 @ N7
mp_add
25 @ N7
mp_add
19 @ N0
mp_add exit
Line 0:
Line 0:
Line 13:
Return edge from mp_add to main: mp_add();
6 @ N4
mp_add
AbstractionState: ABS6
20 @ N12
main
Line 4:
[i < 4]
Line 4:
[!(i < 4)]
Line 4:
[i < 4]
Line 15:
7 @ N5
mp_add
Line 0:
8 @ N6
mp_add
26 @ N5
mp_add
21 @ N14
main
Line 5:
i = i + 1;
Line 7:
return 0;
Line 5:
i = i + 1;
11 @ N7
mp_add
9 @ N0
mp_add exit
27 @ N7
mp_add
Line 13:
Return edge from mp_add to main: mp_add();
Line 16:
Label: ERROR
10 @ N12
main
Line 15:
14 @ N14
main
Line 16:
Label: ERROR
16 @ N15
main
AbstractionState: ABS3: true
AutomatonState: ErrorLabelAutomaton: _predefinedState_ERROR
In Refinement after slicing
0 @ N10
main entry
AbstractionState: ABS0: true
Line 0:
INIT GLOBAL VARS
1 @ N18
main
Lines 1 - 0:
unsigned int mp_add();
int main();
Function start dummy edge
2 @ N11
main
Line 13:
mp_add()
3 @ N1
mp_add entry
Lines 0 - 3:
Function start dummy edge
unsigned char i = 0;
4 @ N3
mp_add
Line 4:
while
6 @ N4
mp_add
AbstractionState: ABS6
Line 4:
[i < 4]
Line 4:
[i < 4]
7 @ N5
mp_add
Line 0:
26 @ N5
mp_add
Line 5:
i = i + 1;
Line 5:
i = i + 1;
11 @ N7
mp_add
27 @ N7
mp_add
Line 0:
17 @ N4
mp_add
AbstractionState: ABS8
Line 4:
[!(i < 4)]
Line 4:
[i < 4]
18 @ N6
mp_add
Line 0:
24 @ N5
mp_add
Line 7:
return 0;
Line 5:
i = i + 1;
19 @ N0
mp_add exit
25 @ N7
mp_add
Line 13:
Return edge from mp_add to main: mp_add();
20 @ N12
main
Line 15:
21 @ N14
main
Line 16:
Label: ERROR
16 @ N15
main
AbstractionState: ABS3: true
AutomatonState: ErrorLabelAutomaton: _predefinedState_ERROR

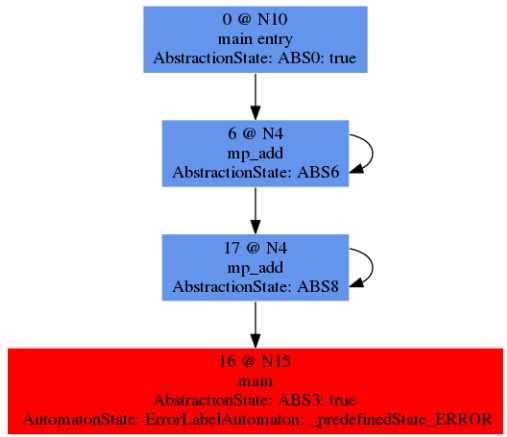
[]



In Refinement after Splitting



In Refinement after slicing



Adjustments for SLAB

- We need a custom transfer relation in the `PredicateCPA`
  $\Rightarrow$Generates successors: $init \wedge error, \neg init \wedge error, \ldots$
- Adapt path-formula creation to generate formulas for program counter
- We need a way to store multiple CFA edges for the transition between two ARG states
  $\Rightarrow$ refinement refine_{SliAbs} needs to be aware of this
- Presence of `LocationCPA` is sometimes assumed in CPAchecker
  $\Rightarrow$ add right handling for when this is not the case
- Apart from that, we can use the same procedure as for Kojak

Flexible-Block Encoding

Problem:

- How can we add ABE to SLAB?
- We cannot form blocks in the initial abstract model

Solution: Flexible-Block Encoding (FBE)

- Use a blk operator similar to that of ABE
- Convert abstraction states after each refinement step
- Fixed-point iteration: apply blk, slice, repeat ...
- FBE can also be used for Kojak

Interprocedural Analysis

For Kojak

- Add `CallstackCPA` that tracks the call stack
- effectively results in function inlining in the initial abstract model
  $\Rightarrow$ sound and precise, but potentially inefficient
- obviously fails for recursive procedures
  (Ultimate Kojak suppports recursion via nested word automata)

Interprocedural Analysis

For SLAB

- Simply adding a `CallstackCPA` would lead to an infinite initial abstract model
- Fix: `CallstackCPA` would need to be aware of whether a function can be called from the current context
- This is then similar to a recursive CFG as used by Ultimate Kojak
  $\Rightarrow$ potentially extendable for recursion

Benchmark

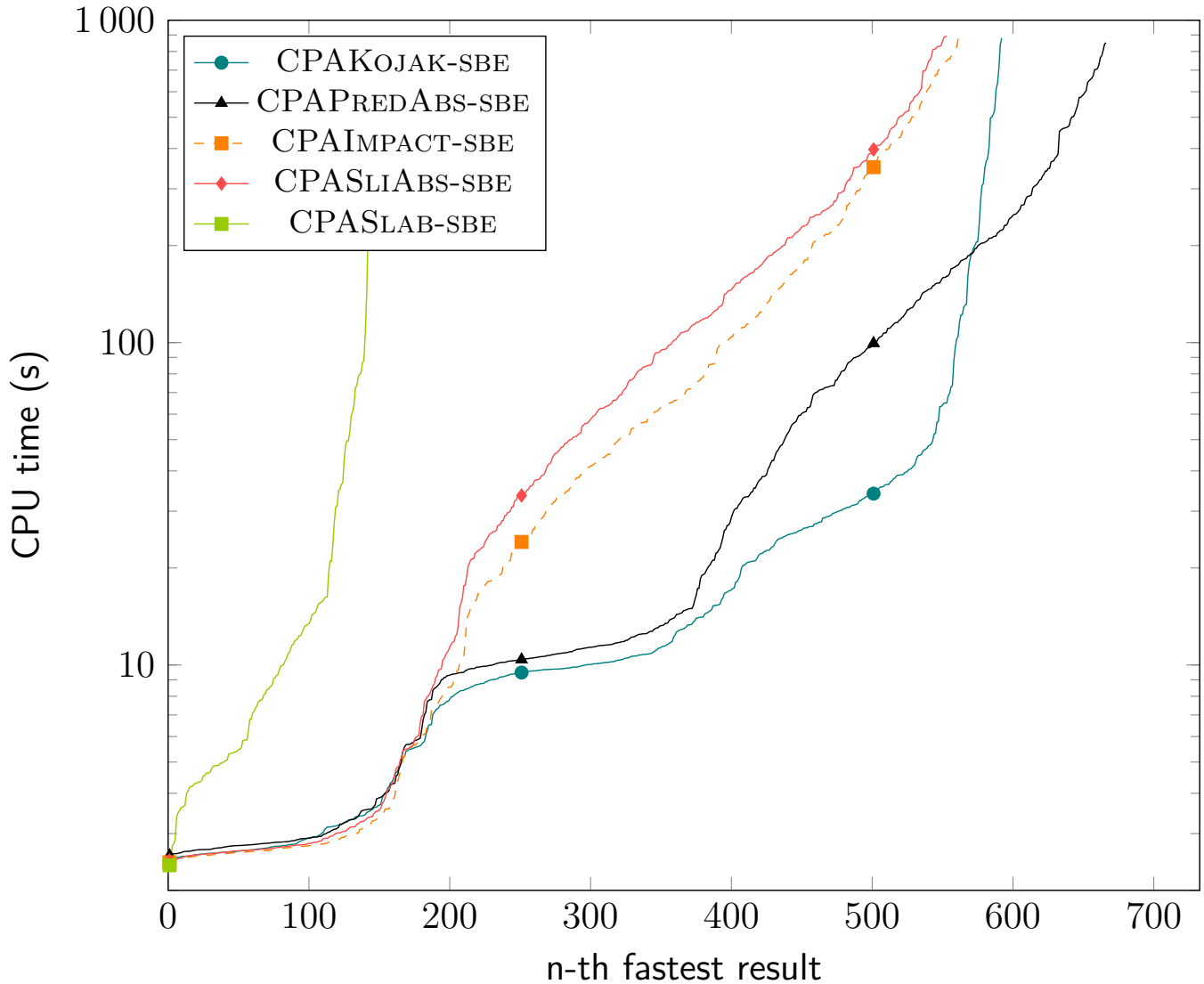
Tasks

- 2942 tasks from the reach-safety category of the 2018 Competition on Software Verification (SVCOMP18)
- 10 subcategories

Environment (identical to SVCOMP18)

- Hosts: Intel Xeon E3-1230 v5 CPU, 8 processing units, 33 GB of memory, Ubuntu 16.04 / Linux kernel 4.4.0-128
- Limits: 900s, 15 GB, 8 processing units
- BenchExec is used for run isolation, resource measurement and limitation

Comparision using Single-Block Encoding

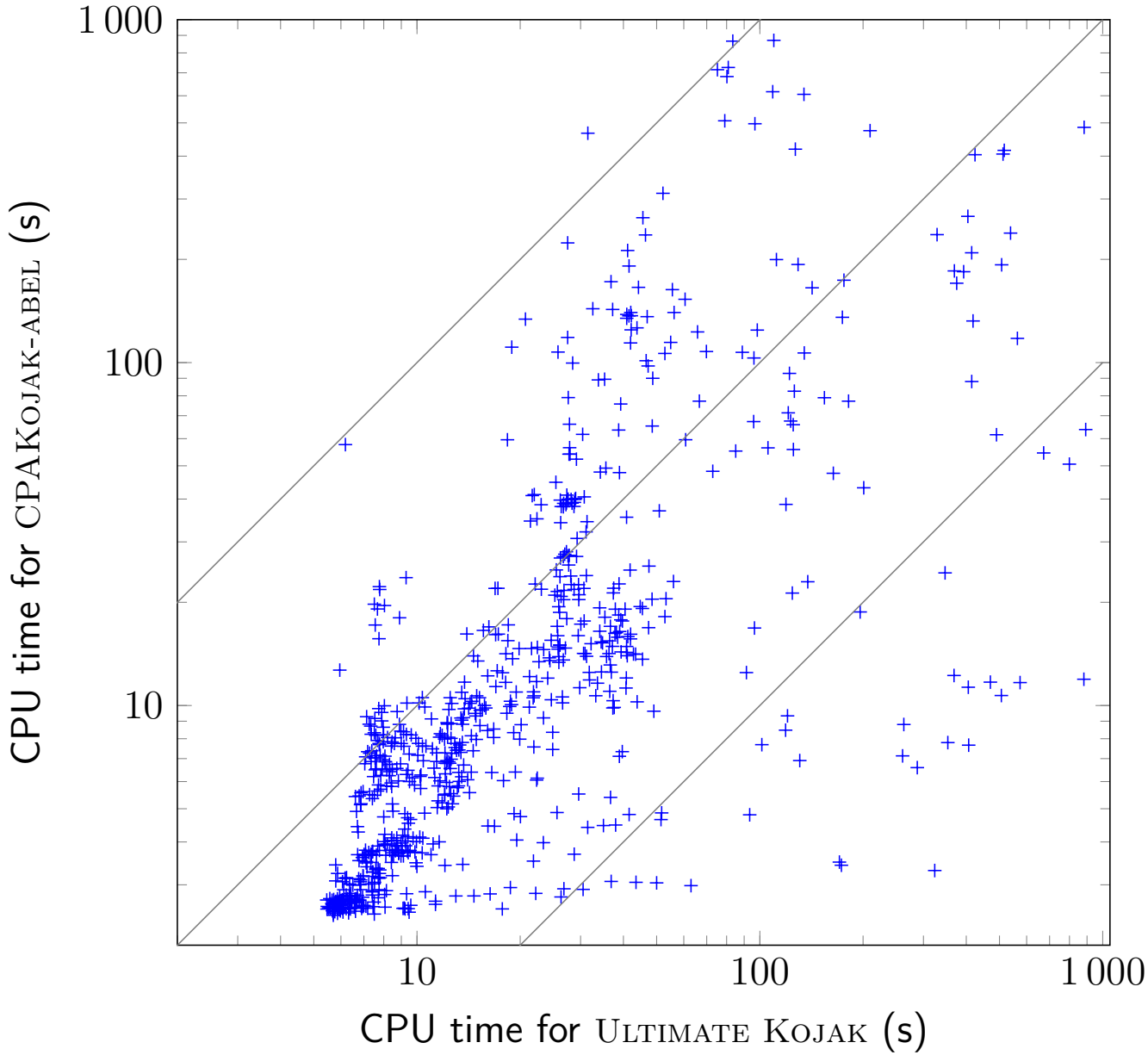


Comparision using Single-Block Encoding

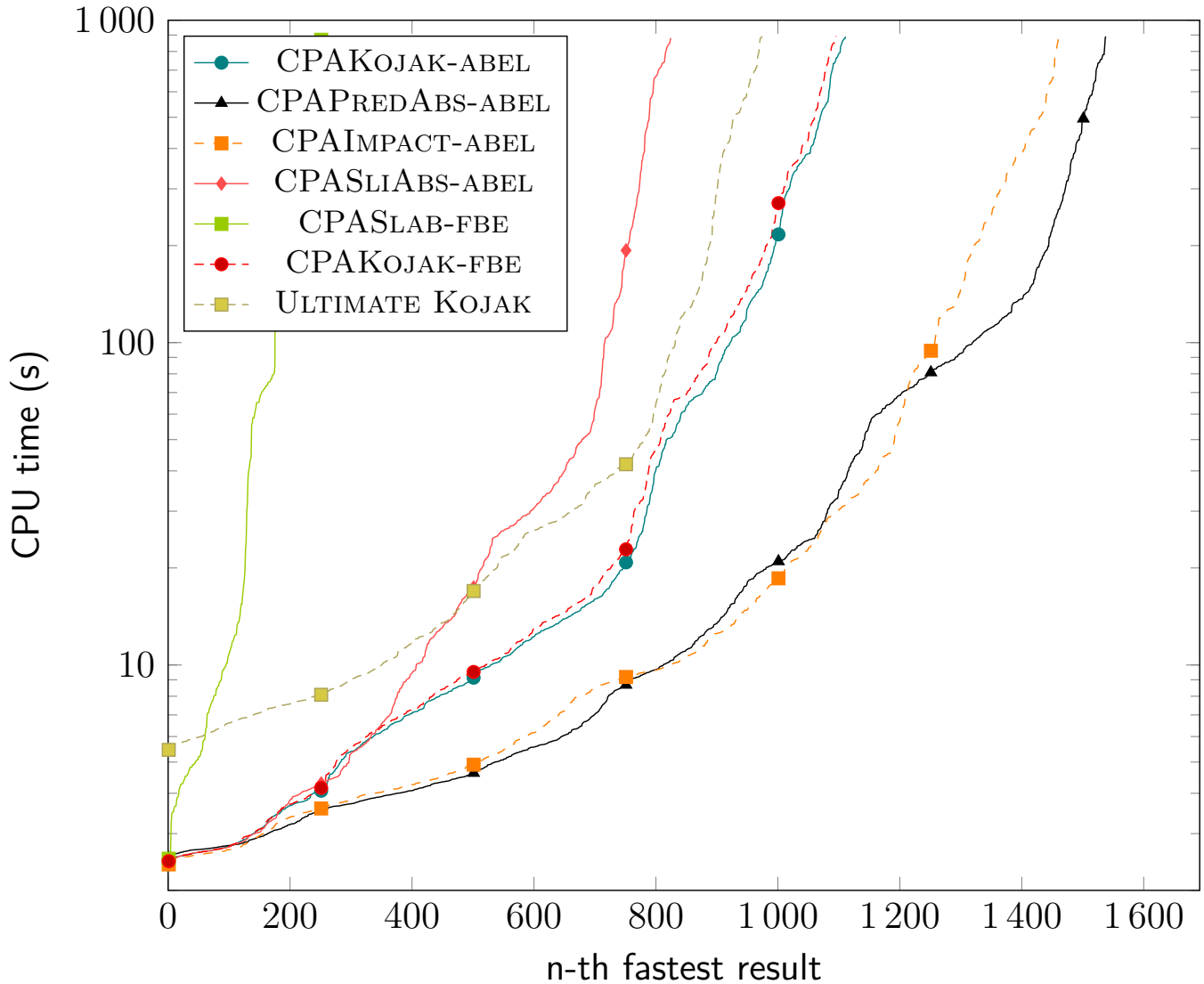
Quantile-Plot Insights

- Replacing the refiner of IMPACT by our $\texttt{refine}_{SliAbs}$ does not change the results significantly (apart from minor overhead of splitting/slicing)
- CPAKOJAK-ABEL is faster than CPAPREDABS-SBE, but solves less tasks (differences are mostly due to product-lines category)
- CPASLAB-SBE takes too many solver calls to discover the control flow
- ULTIMATE KOJAK is not shown since there is no option to disable LBE

Ultimate Kojak vs. CPAKojak-abel



Comparison using Adjustable-Block Encoding

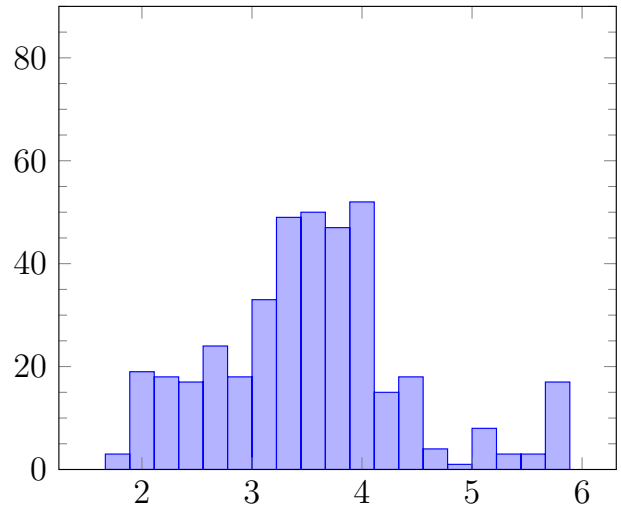


Comparision using Adjustable-Block Encoding

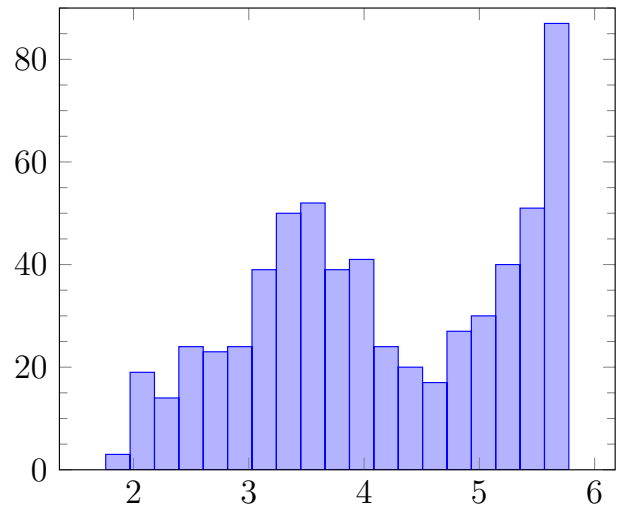
Quantile-Plot Insights

- CPAKojak-abel outperforms Ultimate Kojak (however: not in all subcategories)
- CPAKojak-abel outperformed by CPAPredAbs-abel and CPAImpact-abel
- CPASlab-fbe is better than CPASlab-sbe but has still same problem (too many solver calls for control flow)
- CPAKojak-fbe is no improvement compared to CPAKojak-abel

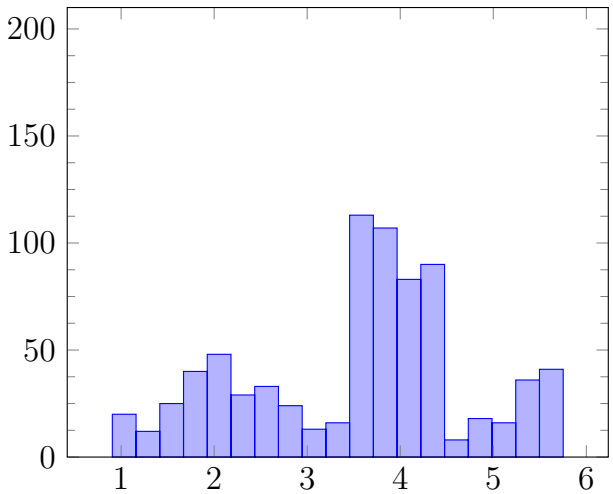
Solver Calls for Slicing



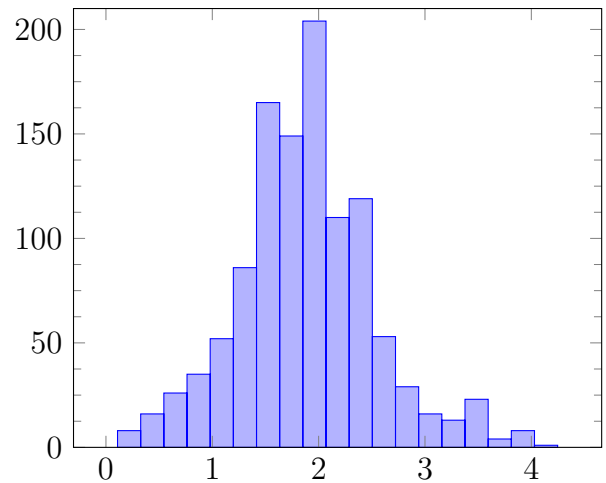
$\log_{10}$ of # of solver calls for slicing in CPASlab-sbe



$\log_{10}$ of # of solver calls for slicing in CPASlab-fbe



$\log_{10}$ of # of solver calls for slicing in CPAkojak-sbe



$\log_{10}$ of # of solver calls for slicing in CPAkojak-abel

Summary

- CPAchecker can now do analyses comparable to Kojak and SLAB
- Our implementation of Kojak is not worse than Ultimate Kojak
- Cost of slicing edges becomes dominant with increasing block size
- Direct usage of control-flow information is important when dealing with programs
- Still a lot of open questions for future work, e. g. recursion, function handling for SLAB, increasing slicing performance, concurrency, witness generation/validation, ...

BACKUP

	Ultimate Kojak	CPAKojak-abel-lin	CPAKojak-abel	CPAKojak-sbe	CPAKojak-fbe	CPAImpact-abel	CPAImpact-sbe	CPAPredAbs-abel	CPAPredAbs-sbe	CPASliAbs-abel	CPASliAbs-sbe	CPASlab-sbe	CPASlab-fbe
arrays (167 tasks)													
correct results	10	11	4	3	5	4	3	6	6	3	3	7	5
bitvectors (50 tasks)													
correct results	19	19	36	27	33	34	30	39	28	31	30	17	19
control flow (94 tasks)													
correct results	31	68	59	44	52	55	24	62	34	39	23	4	8
ECA (1149 tasks)													
correct results	328	235	200	6	201	450	3	477	4	241	3	1	3
floats (172 tasks)													
correct results	33	53	34	6	34	28	6	91	11	27	5	2	2
heap (181 tasks)													
correct results	100	117	108	94	103	110	94	114	94	100	94	54	77
loops (163 tasks)													
correct results	96	103	76	73	74	74	71	82	69	71	68	64	63
product lines (597 tasks)													
correct results	295	462	463	327	463	578	309	551	406	203	304	0	78
recursive (96 tasks)													
correct results	45	0	0	0	0	0	0	0	0	0	0	0	0
sequentialized (273 tasks)													
correct results	17	155	131	12	131	127	22	116	14	109	23	1	2
total (2942)													
correct results	974	1223	1111	592	1096	1460	562	1538	666	824	553	150	257
correct true	664	680	623	368	623	785	291	915	369	391	281	88	179
correct false	310	543	488	224	473	675	271	623	297	433	272	62	78
incorrect results	0	226	1	1	1	1	2	2	1	1	2	31	29

Lazy Abstraction with Interpolants (IMPACT)

Basic Steps:

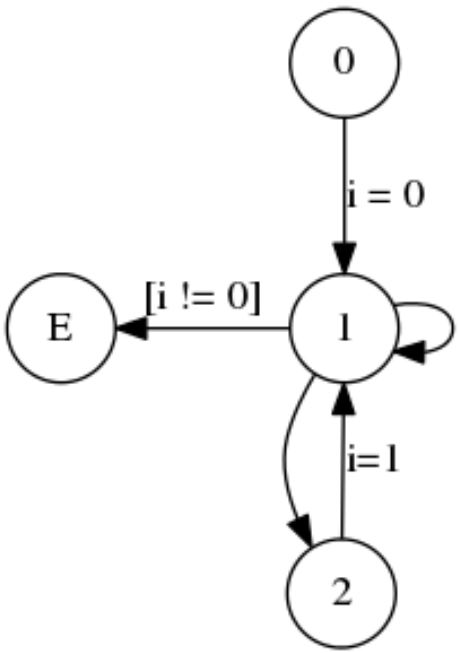
- **Expand:** Unwind the CFA until error state is discovered
- **Refine:** Use interpolation to remove infeasible states from abstraction
- **Cover:** Calculate coverage relation to reach fixed point

Termination

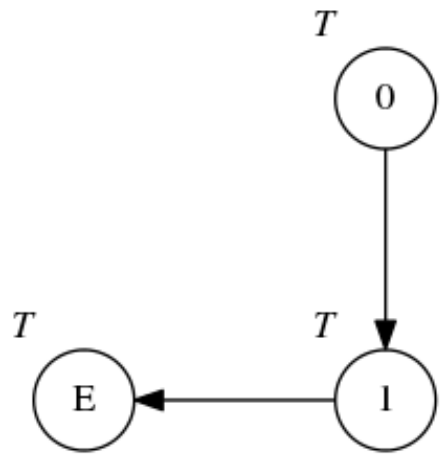
- when finding a feasible counterexample: return false
- when fixed point is reached / all states explored: return true

Lazy Abstraction with Interpolants (IMPACT)

CFA

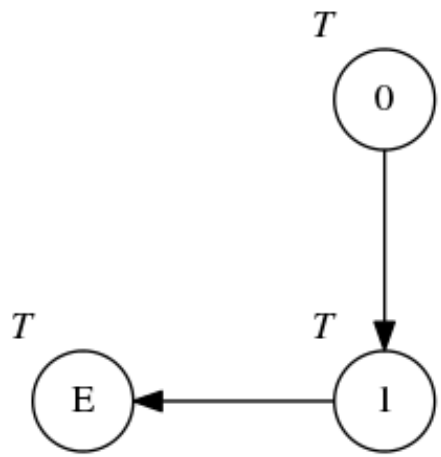


Expand

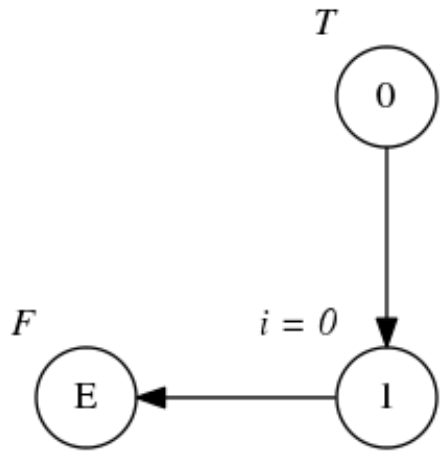


Lazy Abstraction with Interpolants (IMPACT)

Expand

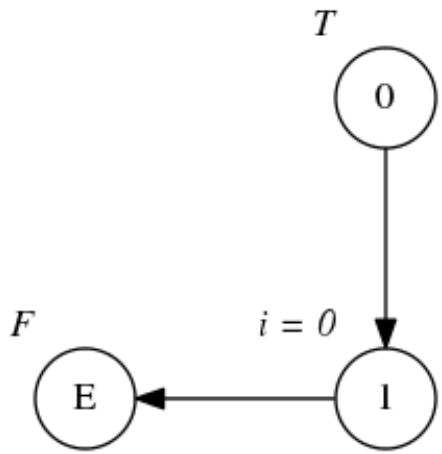


Refine

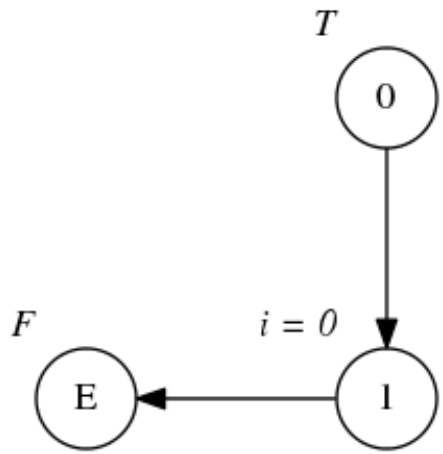


Lazy Abstraction with Interpolants (IMPACT)

Refine

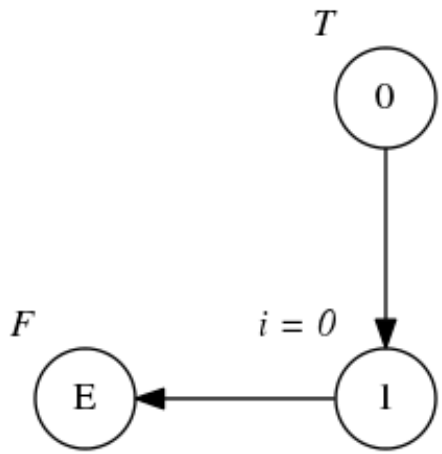


Cover

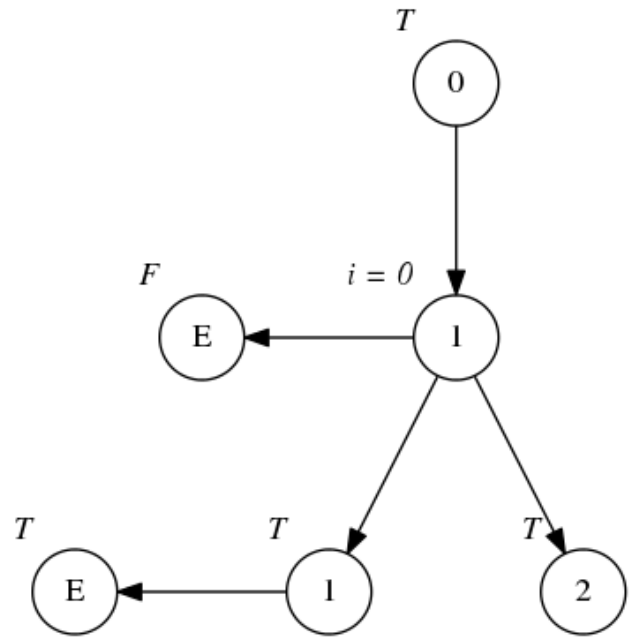


Lazy Abstraction with Interpolants (IMPACT)

Cover

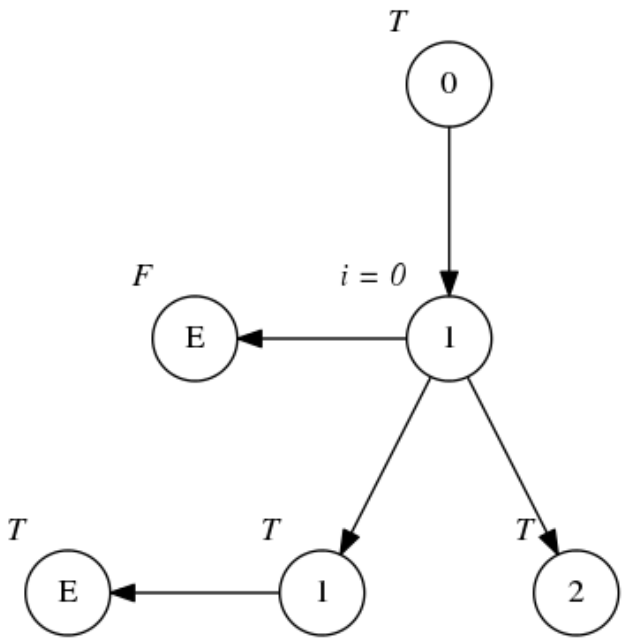


Expand

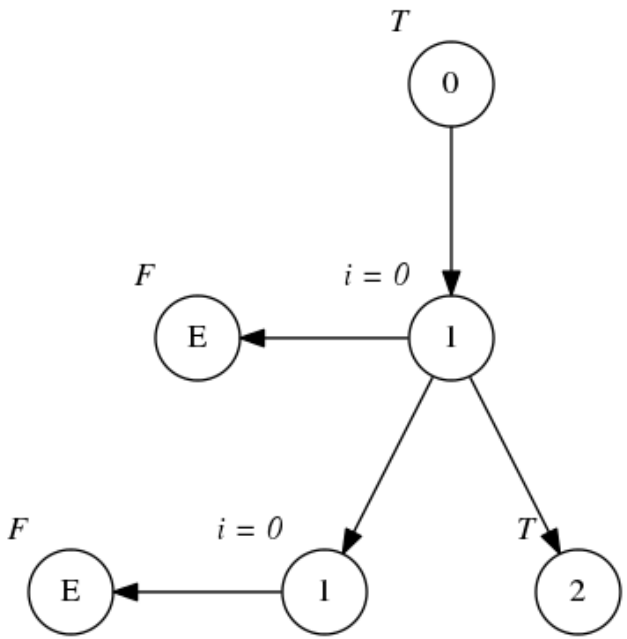


Lazy Abstraction with Interpolants (IMPACT)

Expand

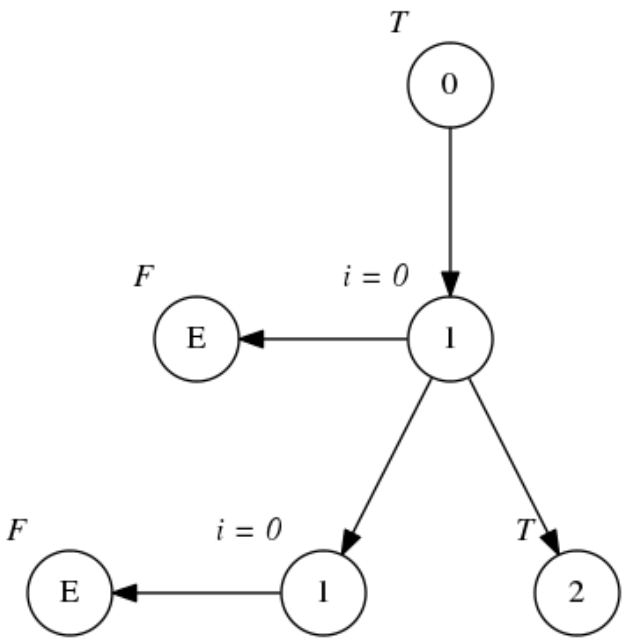


Refine

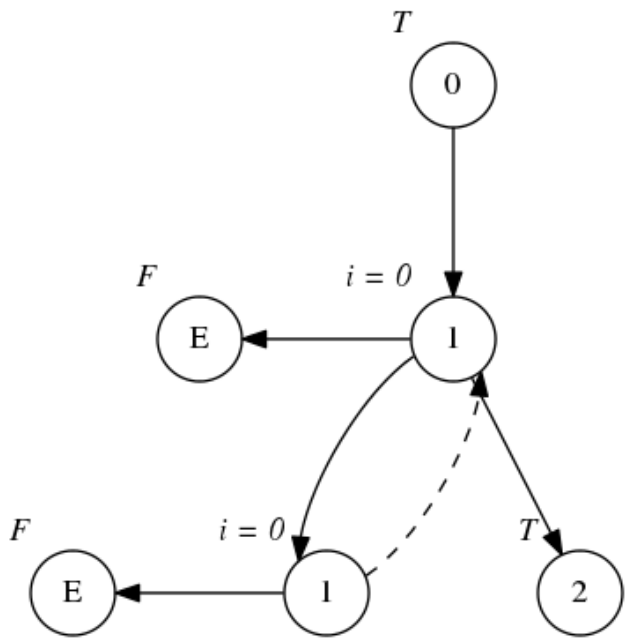


Lazy Abstraction with Interpolants (IMPACT)

Refine

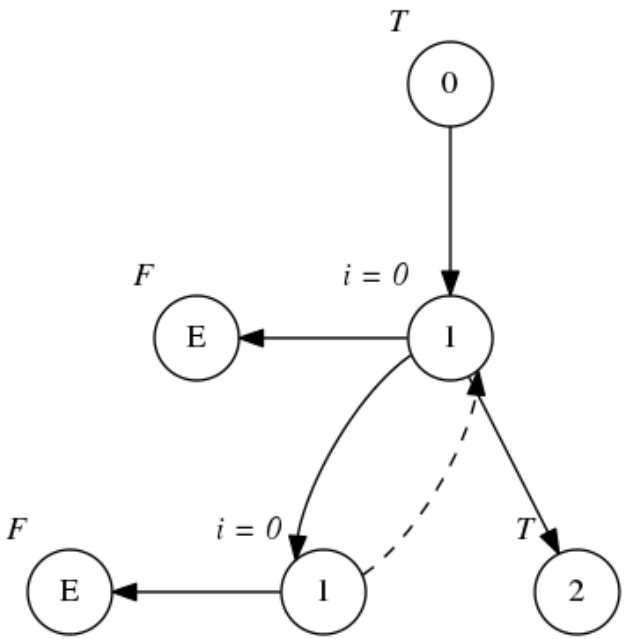


Cover



Lazy Abstraction with Interpolants (IMPACT)

Cover



Expand