LMU

LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

Dirk Beyer • Thomas Lemberger

Symbolic Execution with CEGAR

Tackling the Path Explosion Problem of Symbolic Execution by Borrowing Counterexample-Guided Abstraction Refinement from Model Checking

Outline

1. Symbolic Execution and Path Explosion
2. Applying Counterexample-Guided Abstraction Refinement
3. Evaluation
4. Conclusion

Symbolic Execution and Path Explosion

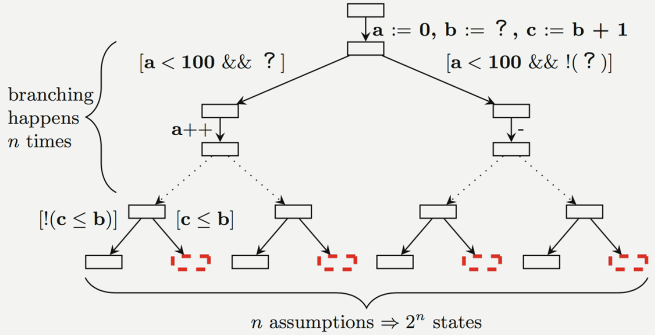
Symbolic Execution and Path Explosion

Symbolic Execution is so useful!

- Tracks explicit/symbolic values and constraints on symbolic values
- Handles unknown and non-deterministic values in dynamic and static analysis (external functions, unavailable libraries, `random()`)
- Test Case Generation, Error Localization, Fault Repair, Verification, Testing, …

But does not scale well. 🤔

```
a := 0;
b := ?;
c := b+1;
while a < 100 do
  if ? do
    a++;
if c <= b do
  error();
```


$\mathbf{a} := \mathbf{0}, \mathbf{b} := ?, \mathbf{c} := \mathbf{b} + \mathbf{1}$
$[\mathbf{a} < \mathbf{100}$ && $?]$
$[\mathbf{a} < \mathbf{100}$ && $!(?)]$
branching happens n times
$\mathbf{a}$++
-
$[!(\mathbf{c} \leq \mathbf{b})]$
$[\mathbf{c} \leq \mathbf{b}]$
n assumptions $\Rightarrow 2^n$ states

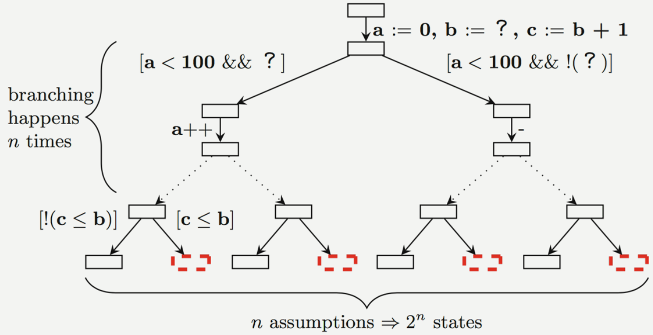
Symbolic Execution and Path Explosion

→ Path explosion due to amount of tracked information

→ But tracked information often unnecessary...

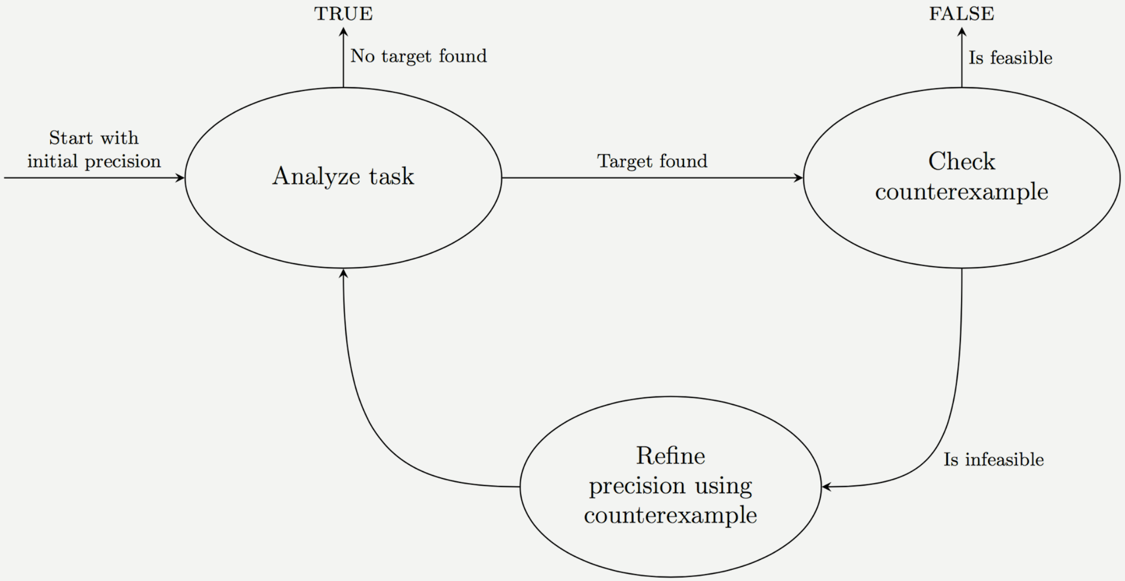
→ Use CEGAR to find out what has to be tracked 😮 ☝️

```
a := 0;
b := ?;
c := b+1;
while a < 100 do
  if ? do
    a++;
if c <= b do
  error();
```


a := 0, b := ? , c := b + 1
[a < 100 && ?]
[a < 100 && !(?)]
branching happens n times
a++
-
[!(c ≤ b)]
[c ≤ b]
n assumptions ⇒ 2^n states

Applying Counterexample-Guided Abstraction Refinement (CEGAR)

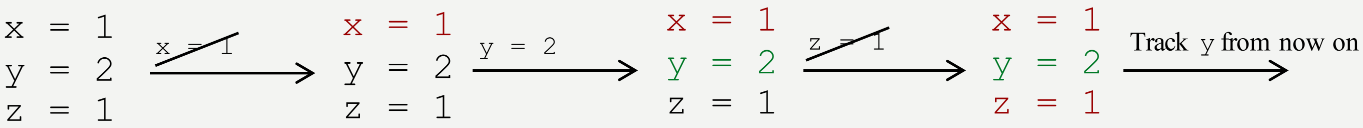
Applying CEGAR to Symbolic Execution



Applying CEGAR to Symbolic Execution

- Initially applied to model checking, already applied to explicit value analysis
- Precision refinement based on Craig interpolants
  1. Start at location 0 with initial interpolant
  2. Compute next value assignment and constraints based on previous interpolant
  3. Filter values needed to proof trace infeasible
  4. Filter constraints needed to proof trace infeasible
  5. Combine values and constraints to interpolant
  6. If not at last location on error trace, go to next location and continue at 2.
  7. Based on interpolants, adjust precision at every location for both value and constraints tracking

Example: Adjusting precision for value tracking



x = 1
y = 2
z = 1

~~x = 1~~

x = 1
y = 2
z = 1

y = 2

x = 1
y = 2
z = 1

~~z = 1~~

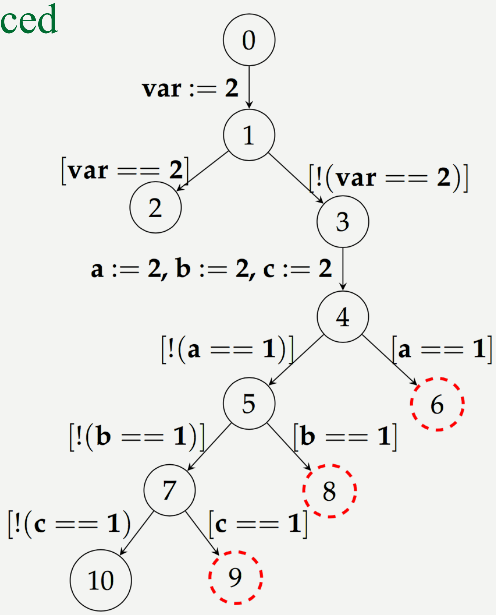
x = 1
y = 2
z = 1

Track y from now on

Applying CEGAR to Symbolic Execution

Further optimization: **Refinement selection**

- Choose good interpolants by computing them on sliced prefixes
- Different heuristics for prefix selection
- Influence behavior and performance significantly
- Example heuristics:
  - Variable domains
  - Interpolant width
  - Number of assumptions in prefix



Evaluation

Evaluation of Symbolic Execution with CEGAR

- Setup:
  - Cluster of Intel Xeon E5-2650 v2 CPUs at 2.60 GHz and 135 GB of memory
  - 2 CPU cores and 15 GB of memory for each verification task
  - 900s time limit
  - SV-COMP'16 task set
- Experiments:
  - Comparison of different refinement heuristics
  - Comparison of Symbolic Execution with and without CEGAR and Symbiotic 3 (based on KLEE)

Evaluation of Symbolic Execution with CEGAR

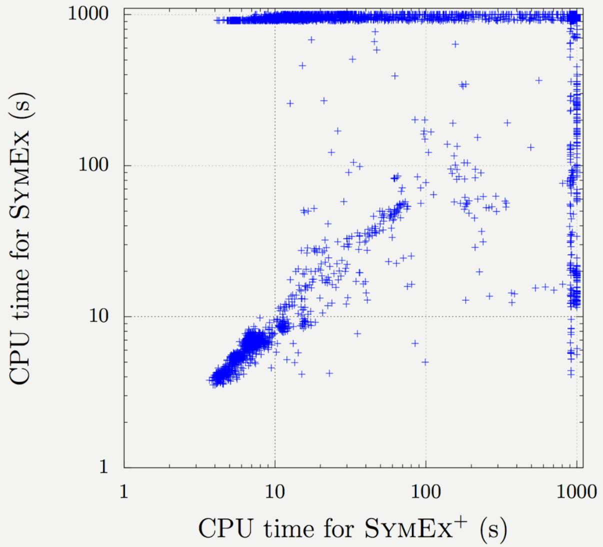
Table 1: Comparison of different refinement-selection heuristics in SymEx$^+$

Verdict	unsolved	solved	correct	correct	incorrect	incorrect
			TRUE	FALSE	TRUE	FALSE
No preference	4341	2336	1737	443	0	**156**
Domain good – width narrow	4444	2233	1702	531	0	171
Domain good – short	3906	**2771**	**2042**	567	0	162
Assumptions most – short	4028	2491	1892	**599**	0	158

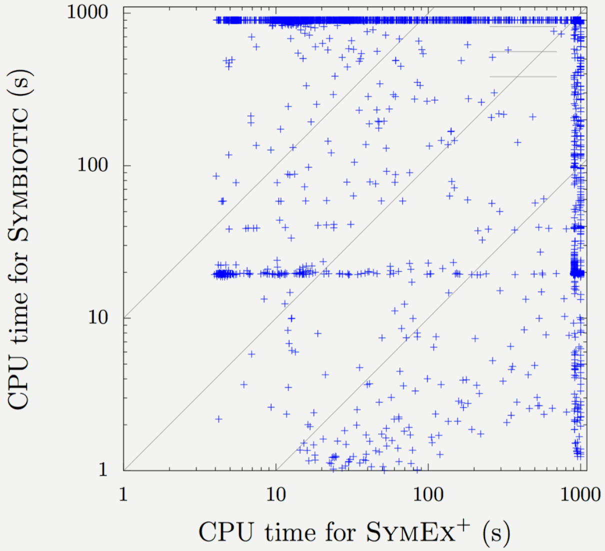
Table 2: Comparison of classical symbolic execution (SymEx) to SymEx$^+$ (both implemented in CPAchecker) and Symbiotic (an external tool)

Verdict	unsolved	solved	correct	correct	incorrect	incorrect
			TRUE	FALSE	TRUE	FALSE
SymEx	5756	921	171	**634**	1	115
SymEx$^+$	3906	**2771**	**2042**	567	0	162
Symbiotic	5388	1289	769	503	2	**15**

Evaluation of Symbolic Execution with CEGAR



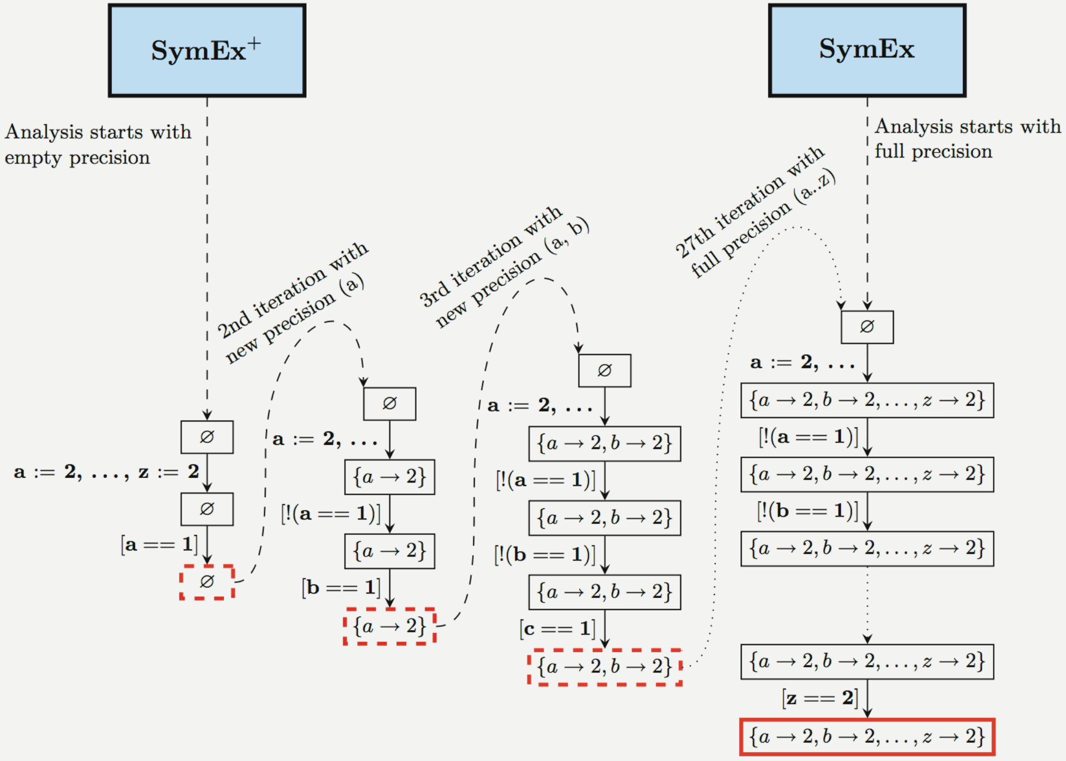
Comparison between Symbolic Execution with CEGAR and without CEGAR



Comparison between Symbolic Execution with CEGAR and Symbiotic 3

Evaluation of Symbolic Execution with CEGAR

Using CEGAR is not always better.



$\mathbf{SymEx}^{+}$

Analysis starts with empty precision

$\varnothing$

$\mathbf{a := 2, \ldots, z := 2}$

$\varnothing$

$[\mathbf{a == 1}]$

$\varnothing$

2nd iteration with new precision (a)

$\varnothing$

$\mathbf{a := 2, \ldots}$

$\{a \to 2\}$

$[!(\mathbf{a == 1})]$

$\{a \to 2\}$

$[\mathbf{b == 1}]$

$\{a \to 2\}$

3rd iteration with new precision (a, b)

$\varnothing$

$\mathbf{a := 2, \ldots}$

$\{a \to 2, b \to 2\}$

$[!(\mathbf{a == 1})]$

$\{a \to 2, b \to 2\}$

$[!(\mathbf{b == 1})]$

$\{a \to 2, b \to 2\}$

$[\mathbf{c == 1}]$

$\{a \to 2, b \to 2\}$

27th iteration with full precision (a..z)

$\mathbf{SymEx}$

Analysis starts with full precision

$\varnothing$

$\mathbf{a := 2, \ldots}$

$\{a \to 2, b \to 2, \ldots, z \to 2\}$

$[!(\mathbf{a == 1})]$

$\{a \to 2, b \to 2, \ldots, z \to 2\}$

$[!(\mathbf{b == 1})]$

$\{a \to 2, b \to 2, \ldots, z \to 2\}$

$\{a \to 2, b \to 2, \ldots, z \to 2\}$

$[\mathbf{z == 2}]$

$\{a \to 2, b \to 2, \ldots, z \to 2\}$

Conclusion

Conclusion

- CEGAR changes behavior of Symbolic Execution significantly
- Tracks only information really necessary for the analysis
- We choose which characteristics this information is supposed to have, using refinement selection

✓ Mitigates problem of path explosion

✓ Provides major performance boost for a significant amount of tasks

○ Challenge: Existence of many error paths with different error causes

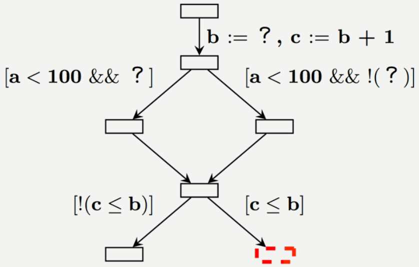
Conclusion

Symbolic Execution is so useful!

- Tracks explicit and symbolic values of execution
- Handles unknown and non-deterministic values in dynamic and static analysis (external functions, unavailable libraries, `random()`)
- Test Case Generation, Error Localization, Fault Repair, Verification, Testing, …

And it scales! 🙂

```
a := 0;
b := ?;
c := b+1;
while a < 100 do
  if ? do
    a++;
if c <= b do
  error();
```



Thank you! Questions?