Verifying Numerical Programs via Iterative Abstract Testing

Banghu Yin[1] Liqian Chen[1] Jiangchao Liu[1] Ji Wang[1] Patrick Cousot[2]

[1]National University of Defense Technology, Changsha, China
[2]New York University, New York, USA

SAS 2019@Porto, 2019-10-10

Overview

- Motivation
- Approach
- Experiment
- Conclusion

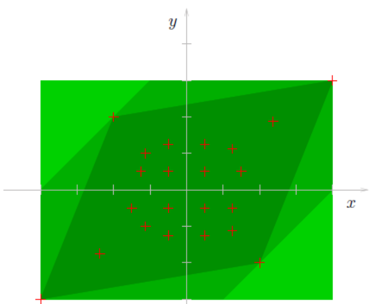
Program Verification

- Given a program P, and an assertion ψ :
  - if the assertion ψ is true, give a proof
  - if the assertion ψ is false, give a counter example

```
void cumsum(int n)
{
   n=random;
   x=0; y=0;
   while(x<n){
      x=x+1;
      y=y+x;
   }
   assert(y!=100);
   x = 1/(y-100);
}
```

Abstract Interpretation (AI)

- AI: a framework to design static analyses that are
  - **sound** by construction (no behavior is omitted)
    - no false negative
  - **approximate** (trade-off between precision & efficiency)
    - rates of false positives vs. scalability



precision **efficiency**

Abstract Interpretation (AI)

- Abstract interpretation for verification
  - generate sound program invariants
  - check the target property using invariants

```
void cumsum(int n)
{
   x=0; y=0;
   while(x<n){
      x=x+1;
      y=y+x;
   }
   assert(y!=100);
  //  x = 1/(y-100);
}
```

the Box abstract domain

```
void cumsum(int n)
{
   x=0; y=0;  // x:[0,+oo],y:[0, +oo]
   while(x<n){ //x:[0,+oo],y:[0, +oo], n:[1,+oo]
      x=x+1;  //x:[1,+oo],y:[0, +oo], n:[1,+oo]
      y=y+x;  //x:[1,+oo],y:[1, +oo], n:[1,+oo]
   } //x:[0,+oo],y:[0, +oo]
   assert(y!=100);
  //  x = 1/(y-100);
}
```

Abstract Interpretation (AI)

- **Problems**: (simple) AI-based verification approaches
  - do not make full use of target property
  - are hardly able to generate counter-examples
    - hard to prove false assertions
  - may get too conservative over-approximations
    - hard to prove true (non-simple) assertions

Abstract Interpretation (AI)

- **Problems**: (simple) AI-based verification approaches
  - do not make full use of target property
  - are hardly able to generate counter-examples
    - hard to prove false assertions
  - may get too conservative over-approximations
    - hard to prove true (non-simple) assertions
    - main cause of precision loss in AI
      - **expressiveness limitation**
        - in expressing disjunctive, non-linear properties
      - **widening**
        - often aggressively weakens unstable predicates

Main Idea

- "Iterative Abstract Testing" for verification
    - iteratively perform forward & backward AI
        - with input space partitioning
        - using backward AI to refine the given input space
            - making use of the target property
    - use bounded exhaustive testing to complement AI
        - to verify an input sub-space involving limited number of inputs
        - to generate counter-examples for false assertions

Main Idea

- "Iterative Abstract Testing" for verification
  - iteratively perform forward & backward AI
    - with input space partitioning
    - using backward AI to refine the
      - making use of the target property
  - use bounded exhaustive testing to complement AI
    - to verify an input sub-space involv
    - to generate counter-examples for

Abstract execution

Concrete execution

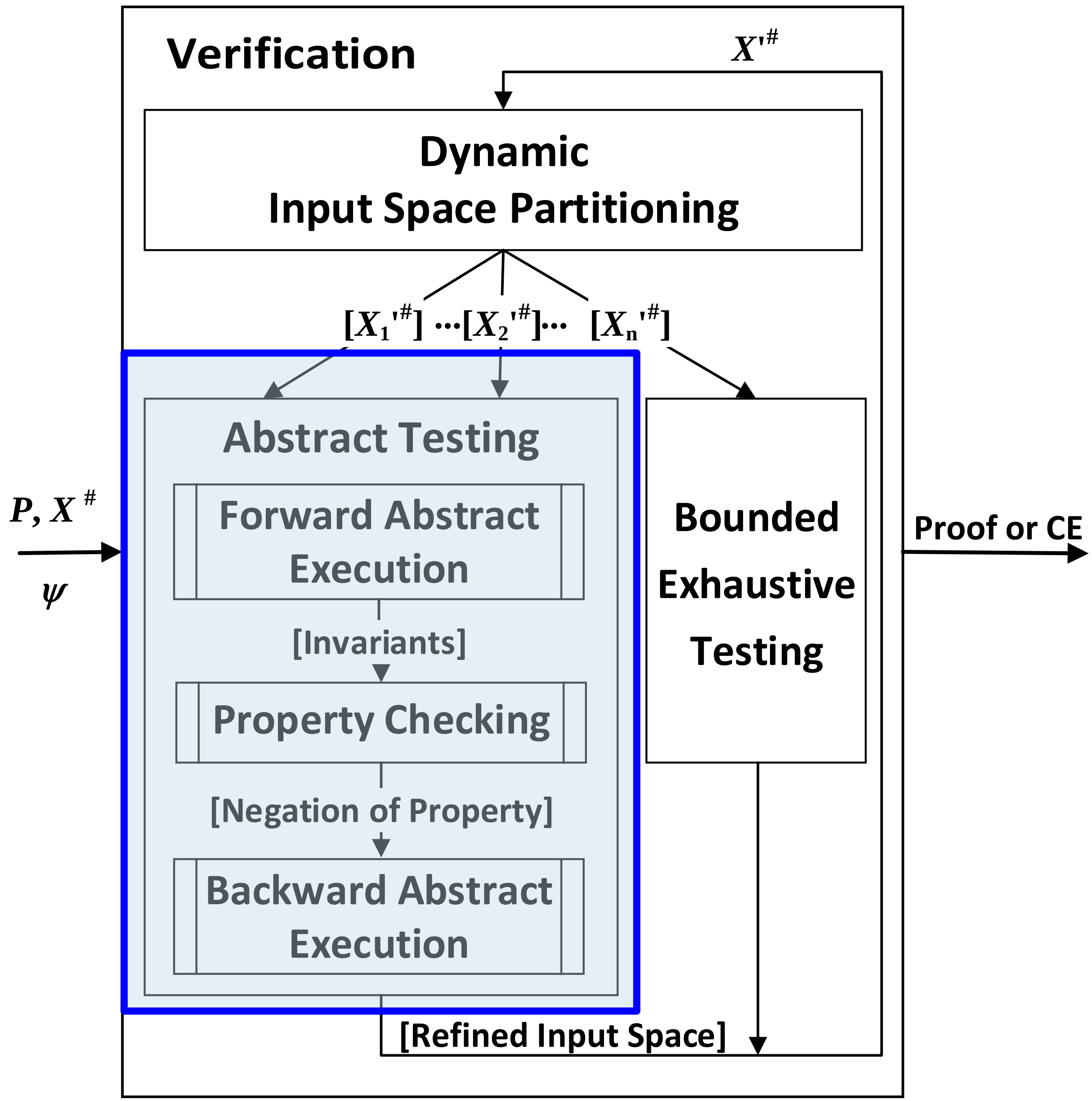
The whole process continues until

- a **counter-example** is found or
- the whole input space is **verified** against the property

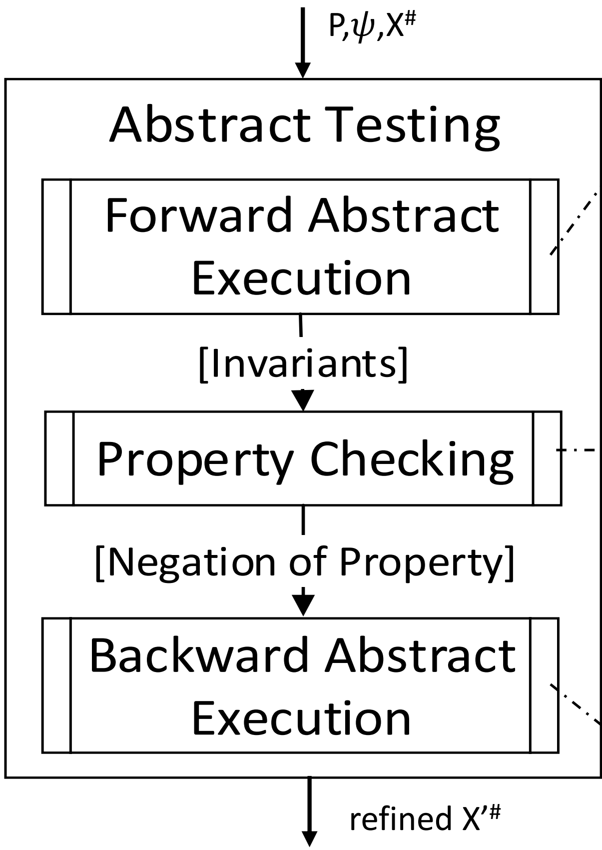
Overview

- Motivation
- Approach
- Experiment
- Conclusion

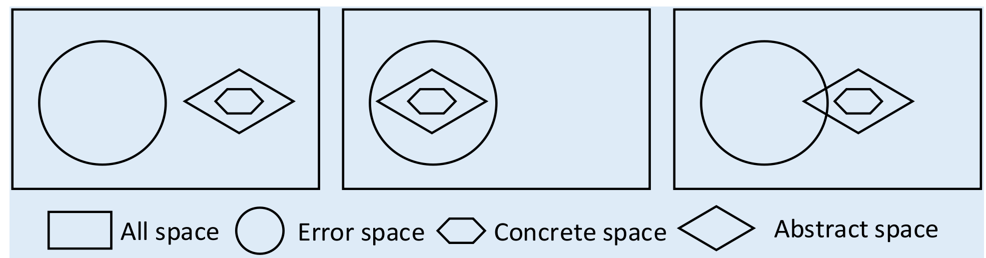
Main Framework



Abstract Testing [Cousot&Cousot, SSGRR 2000]



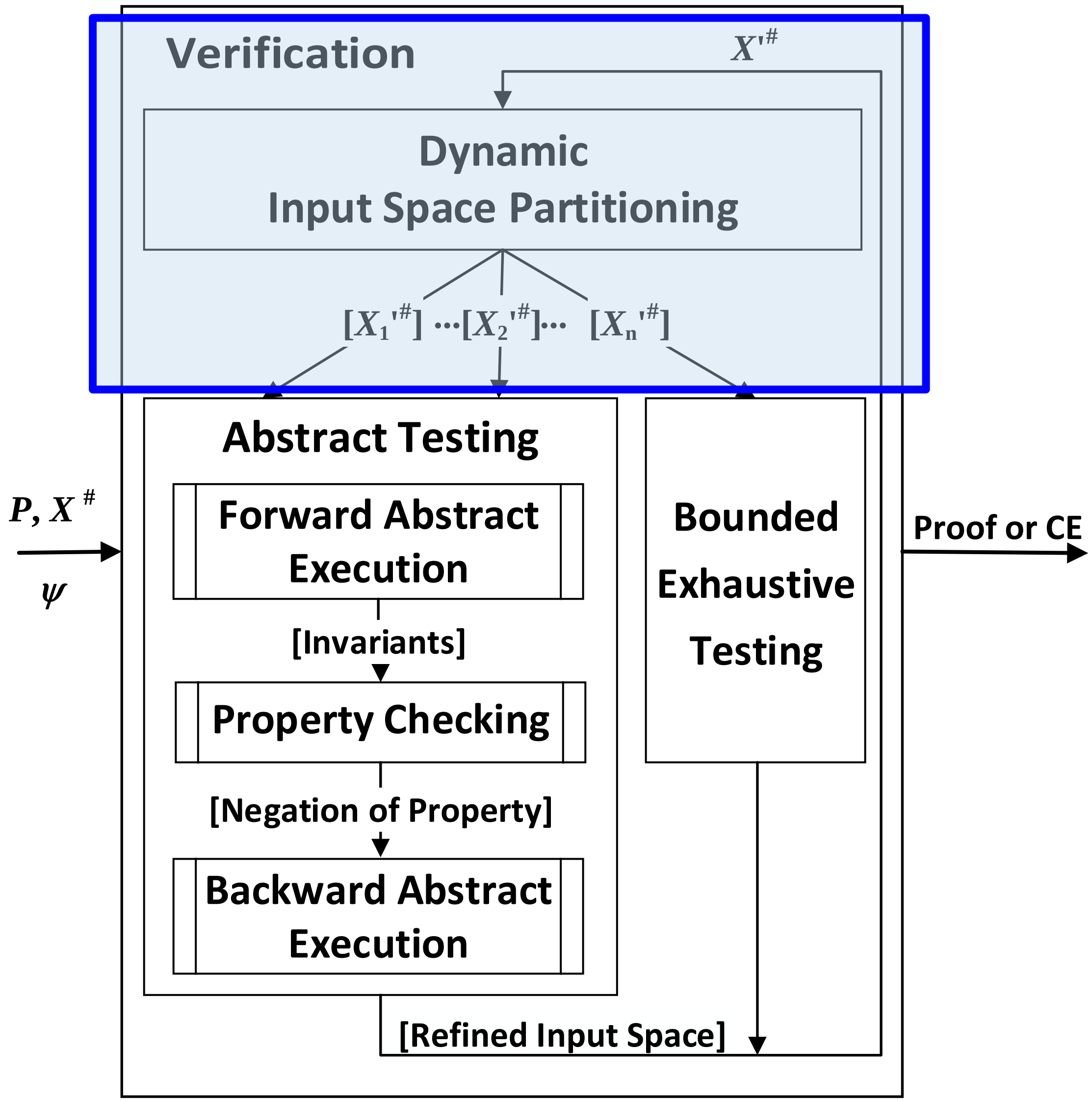
Perform **forward AI** for an input $X^{\#}$, which may contain unlimited test cases.



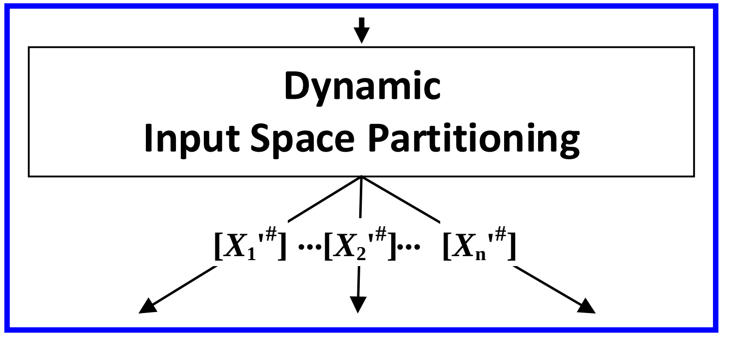
Perform **backward AI** from negation of ψ to achieve a **refined input** $X'^{\#}$, i.e. $X'^{\#} \sqsubseteq X^{\#}$.

[Cousot, Cousot. Abstract interpretation based program testing. In SSGRR 2000]

Input Space Partitioning



Input Space Partitioning



- Partitioning
  - given an abstract input $X'^{\#}$ that has not yet been proved, split it into a list of subspaces ($\{X_1'^{\#}, \ldots, X_n'^{\#}\}$)
- **Partitioning strategies**
  - partitioning by dichotomy --- guarantee the termination
    - $x \in [a, b] \rightarrow [a, (a+b)/2]$ and $[(a+b)/2, b]$
  - partitioning by predicates --- improve the efficiency
    - via a selection of predicates over symbolic input variables

Partitioning by predicates

- Basic idea:
  - introduce a symbolic input variable for every input variable
  - do Forward AI to generate invariants
  - do splitting based on predicates over symbolic input variables at conditional tests.

```
var x:int,y:int,x0:int;
begin
  x0=x;// insert temp var
  y=0;
  while x>0 do
   x=x-1;
   if x>=50 then
     y=y-1;
   else
     y=y-3;
   endif;
  done;
  if y==-100 then
   fail;
  endif;
end
```

Forward AI →

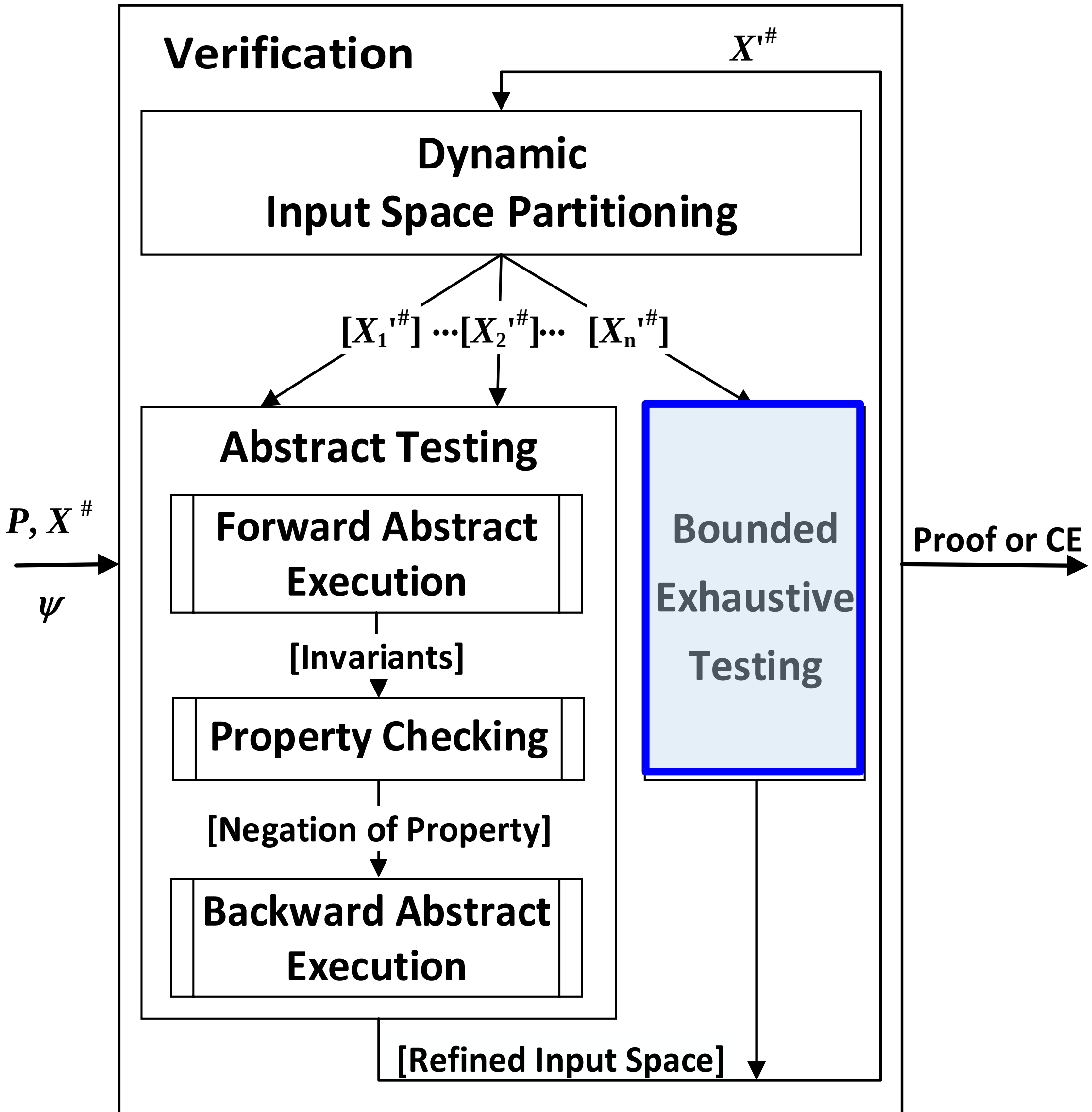
```
var x:int,y:int,x0:int;
begin
  x0=x;// x0=T
  y=0;
  while x>0 do //x0>=1
   x=x-1;
   if x>=50 then //x0>=51
     y=y-1;
   else  //x0>=1
     y=y-3;
   endif;
  done;
  if y==-100 then//x0<=100
   fail;
  endif;
end
```

Predicates → x0>=1, x0>=51, x0<=100

↓ Do Splitting

(x0<1) V (1 <= x0 <=50) V (51<= x0 <= 100) V (100<x0)

Bounded Exhaustive Testing (BET)



Bounded Exhaustive Testing (BET)

- Basic idea
  - test all the concrete inputs in an abstract input $X^{\#}$ of small size
  - guarantee the completeness

```
void cumsum(int n)
{
   x=0; y=0;
   while(x<n){
      x=x+1;
      y=y+x;
   }
   assert(y!=100);
    x = 1/(y-100);
}
```

BET for
n=1,…,100

BET executes totally 1+2+3+…+100 = 5050 times of the loop body to prove the assertion

Bounded Exhaustive Testing (BET)

- Efficiency improvement
  - check necessary preconditions at locations after conditional tests during BET

```
void cumsum(int n)
{
  x=0; y=0;
  while(x<n){
    assert(y<=99);
    x=x+1;
    y=y+x;
  }
  assert(y!=100);
   x = 1/(y-100);
}
```

BET for n=1,…,100

E.g., when n=100, the assertion y<=99 will be violated after 14 iterations of the loop body.

So, BET executes totally 1+2+…+14+86*14 = 1309 times of the loop body to prove the assertion

Illustration via an Example

```
void ex(int n)
{
  x=0; y=0;
  while(x<n){
    if(x*y<20){
      x=x+1;
      y=y+2;
    }
    else {
      x=x+1;
      y=y+3;
    }
  }
  assert(y!=100);
}
```

Forward AI only

```
void ex(int n)
{
  x=0; y=0;
  while(x<n){
    if(x*y<20){
      x=x+1;
      y=y+2;
    }
    else {
      x=x+1;
      y=y+3;
    }
  }
  //x>=0, y>=2x, y<=3x, x=n
  assert(y!=100);
}
```

fail to prove

Exhaustive testing

Exhaustively enumerate
n ∈ [min_int, max_int]
to prove this assertion

too costly

Illustration via an Example

For n ∈ [min_int,33] do AI:

```
void ex(int n)
{
   n_0=n;
   x=0; y=0;
   while(x<n){
      if(x*y<20){
        x=x+1;
        y=y+2;}
      else {
         x=x+1;
         y=y+3; }
}
//x>=0, y>=2x, y<=3x, x<=33
assert(y!=100);
}
```

proved

For n ∈ [51,max_int] do AI:

```
void ex(int n)
{
   n_0=n;
   x=0; y=0;
   while(x<n){
      if(x*y<20){
        x=x+1;
        y=y+2;}
      else {
         x=x+1;
         y=y+3; }
}
//x>=51, y>=2x, y<=3x
assert(y!=100);
}
```

proved

For n∈[34,50] do BET

Limited cases testing:
for n∈[34,50] do
    Test ex(n);

proved

For the whole input domain n ∈ [min_int, max_int], assertion proved!

Overview

- Motivation
- Approach
- Experiment
- Conclusion

Benchmarks and EQs

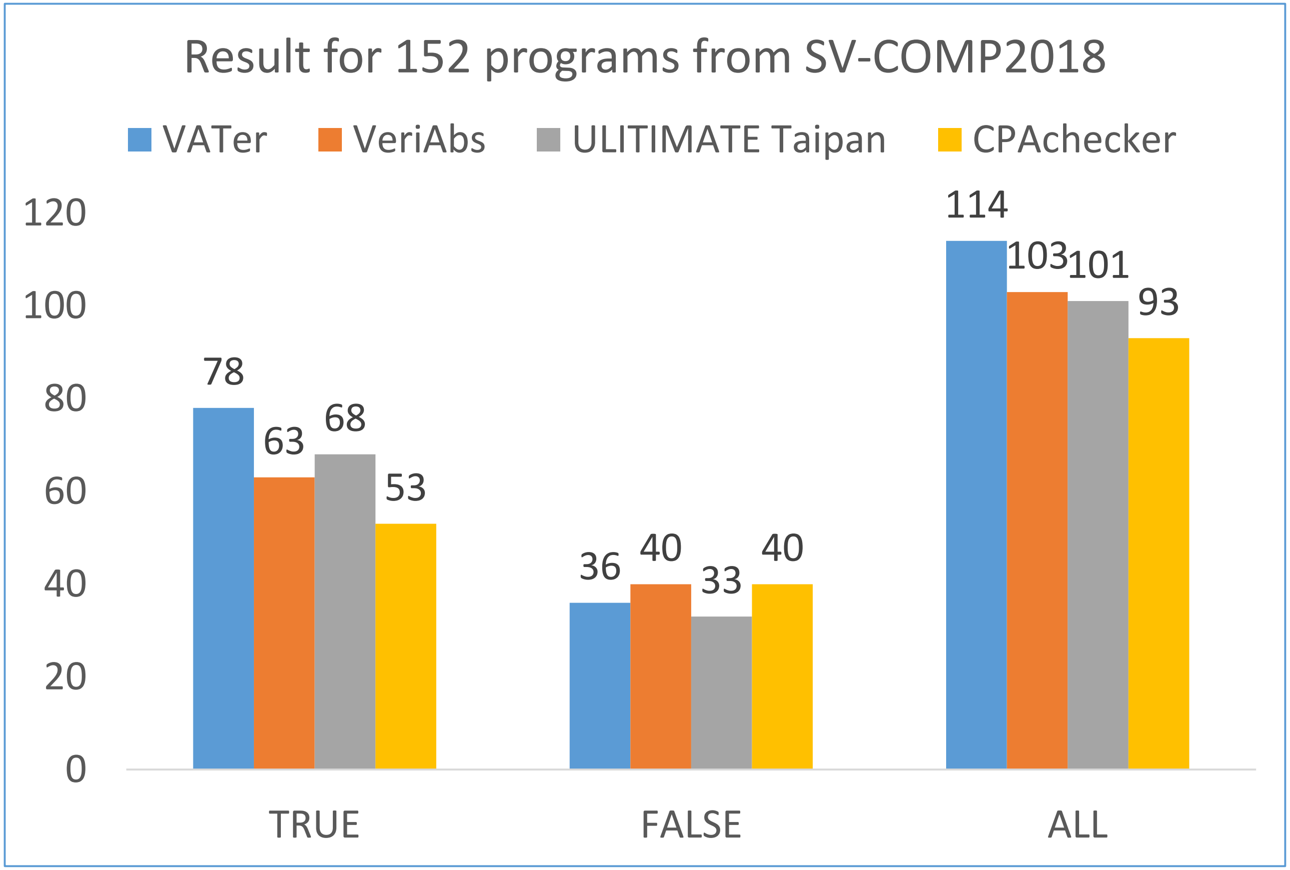
- Implementation: VATer
- Benchmarks
  - HOLA [Dillig et al, OOPSLA13], C4B [Carbonneaux et al, PLDI15] benchmark
    - 46 programs and 35 programs with true assertions
    - involving input-data dependent loops with disjunctive or non-linear prop.
  - SV-COMP 2018
    - all the 152 programs from six folders (in ReachSafety-Loops category)
- Experimental questions
  - EQ1: Ability of VATer in proving true assertions compared with AI-involved tools
  - EQ2: How does VATer work comparing with state-of-the-art verification tools?
  - EQ3: Usefulness of BET in VATer

EQ1: Ability of proving true assertions compared with AI-involved tools

Benchmark	Interproc		SeaHorn		U Taipan		VATer			
	#V	T(s)	#V	T(s)	#V	T(s)	#V	T(s)	#AT	#BET
HOLA(46)	**17**	2.9	**34**	298.5	**38**	805.1	**44**	14.1	64	1
C4B(35)	**2**	0.1	**24**	274.9	**17**	1277.4	**32**	2.3	85	0
Total(81)	**19**	3.0	**58**	573.4	**51**	2082.6	**76**	16.4	149	1

- VATer can verify 57(3X) , 18(31%) and 25(49%) more **true assertions** than Interproc, SeaHorn and U Taipan
- This strengthening mainly comes from the **iterative abstract testing** through dynamic input partitioning.

EQ2: Comparing with state-of-the-art verification tools



Tools	Average Time(s)
VATer	1.9
VeriAbs	26.0
U Taipan	28.8
CPAChecker	55.4

Comparing with VeriAbs, U Taipan and CPAChecker, VATer achieves 11%, 13%, 22% improvement and has on average 13.6X, 15.2X, 29.2X speedups.

EQ3: Usefulness of BET in VATer

Folder	Assertions	IAT		VATer (IAT + BET)			
		#V	T(s)	#V	T(s)	#AT	#BET
Loops(67)	True(35)	21	4.2	23	5.0	23	2
	False(32)	7	2.4	18	55.0	145	11
Loop-new(8)	True(8)	4	4.7	7	5.8	7	3
	Fasle(0)	0	0	0	0	0	0
Loop-lit(16)	True(15)	9	2.3	13	2.7	15	4
	False(1)	0	0	1	0.2	1	1
Loop-inv(19)	True(18)	15	32.6	15	32.6	16	0
	False(1)	0	0	1	11.9	86	1
Loop-craft(7)	True(6)	2	0.2	4	0.6	4	2
	False(1)	0	0	0	0	0	0
Loop-acc(35)	True(19)	9	0.9	16	96.2	78	38
	False(16)	1	0.1	16	9.0	43	15
Total(152)	True(101)	60	44.9	78	142.9	143	49
	False(51)	8	2.5	36	76.1	275	28

- For the 101 programs with true assertions, VATer verifies 18 (30%) more programs than IAT
- For the 51 programs with false assertions, VATer generates counter-examples for 28 (3.5X) more programs than IAT

Overview

- Motivation
- Approach
- Experiment
- Conclusion

Summary

- A property-oriented verification approach based on iterative abstract testing with dynamic input partitioning
- Using bounded exhaustive testing to complement abstract testing based verification
- A tool called VATer based on the proposed approach, which achieves promising results

Future Work

- Other dynamic analysis techniques to complement abstract testing

- Parallel implementation: parallelizable by nature thanks to the partitioning

Thank you
Any Questions?