SymInfer: Inferring Program Invariants using Symbolic States

ThanhVu (Vu) Nguyen*,
Matthew B. Dwyer*, Willem Visser†

*University of Nebraska-Lincoln, †Stellenbosch University

ASE 2017
Program invariants are asserted properties, such as relations among variables that always hold at certain locations in a program.

- Pre/Post conditions, loop invariants, assertions
Introduction

Program invariants are asserted properties, such as relations among variables that always hold at certain locations in a program

- Pre/Post conditions, loop invariants, assertions

Numerical invariants, e.g., relations among numerical variables

- E.g., \( x = 2y + 3, 0 \leq idx \leq |arr| - 1, x \leq y^2, x = qy + r \)

- Nonlinear polynomial invariants: \( x \leq y^2, x = qy + r, \ldots \)
Program invariants are asserted properties, such as relations among variables that always hold at certain locations in a program

- Pre/Post conditions, loop invariants, assertions

Numerical invariants, e.g., relations among numerical variables

- E.g., \( x = 2y + 3, 0 \leq idx \leq |arr| - 1, x \leq y^2, x = qy + r \)
- Nonlinear polynomial invariants: \( x \leq y^2, x = qy + r, \ldots \)

Techniques for automatic invariant generation

- **Statically** examine program code, **dynamically** analyze concrete states (traces), or hybridization of dynamic inference and static checking
- **SymInfer**: hybridization using **symbolic states**
  - Symbolic states: obtained from symbolic execution, intermediate representation of states, consist of path conditions and local variables
  - Infer: use symbolic states to generate sample traces and infer invariants
  - Check: use symbolic states to check candidate invariants
Example: Numerical Invariants

```c
int cohendiv(int x, int y){
    assert(x>0 && y>0);
    int q=0; int r=x;
    while(r ≥ y){
        int a=1;
        int b=y;
        while[L1](r ≥ 2*b){
            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
    [L2]
    return q;
}
```

What does this program do? What properties hold at L1 and L2?

SymInfer automatically generates loop invariants at L1:

- \( x = qy + r \), \( b = ya \), \( y \leq b \), \( b \leq r \), \( r \leq x \), \( a \leq b \), \( 2 \leq a + y \)

Postconditions at L2:

- \( x = qy + r \), \( 1 \leq q + r \), \( r \leq y - 1 \), \( 0 \leq r \), \( r \leq x \)

Invariants describe program's semantic, e.g., \( x = qy + r \) for integer division and reveal useful information, e.g., remainder r is non-negative.
int cohenDiv(int x, int y){
    assert(x>0 && y>0);
    int q=0; int r=x;
    while(r ≥ y){
        int a=1;
        int b=y;
        while[L1](r ≥ 2*b){
            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
    [L2]
    return q;
}
Example: Numerical Invariants

```c
int cohendiv(int x, int y){
    assert(x>0 && y>0);
    int q=0; int r=x;
    while(r ≥ y){
        int a=1;
        int b=y;
        while[L1](r ≥ 2*b){
            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
    return q;
}
```

What does this program do? What properties hold at L1 and L2?

SymInfer automatically generates

- **loop invariants at L1:**
  \[ x = qy + r, \quad b = ya, \quad y \leq b, \]
  \[ b \leq r, \quad r \leq x, \quad a \leq b, \quad 2 \leq a + y \]

- **postconditions at L2:**
  \[ x = qy + r, \quad 1 \leq q + r, \]
  \[ r \leq y - 1, \quad 0 \leq r, \quad r \leq x \]

- Invariants describe program’s semantic, e.g., \( x = qy + r \) for integer division and reveal useful information, e.g., remainder \( r \) is non-negative
Symbolic States

```c
int cohendiv(int x, int y){
    assert(x>0 && y>0);
    int q=0; int r=x;
    while(r ≥ y){
        int a=1;
        int b=y;
        while[L1](r ≥ 2*b){
            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
    [L2]
    return q;
}
```

Run *symbolic execution* to obtain
- Path conditions over input variables
- Relationships among local variables
int cohendiv(int x, int y){
    assert(x>0 && y>0);
    int q=0; int r=x;
    while(r ≥ y){
        int a=1;
        int b=y;
        while[L1](r ≥ 2*b){
            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
    return q;
}
Symbolic States

Run *symbolic execution* to obtain

- **Path conditions** over input variables
- **Relationships among local variables**

**At L1:**

<table>
<thead>
<tr>
<th>Pathconds</th>
<th>Locals</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x \geq y \land y &gt; 0)</td>
<td>(q = 0 \land r = x \land a = 1 \land b = y)</td>
</tr>
<tr>
<td>(x \geq 2y \land y &gt; 0)</td>
<td>(q = 0 \land r = x \land a = 2 \land b = 2y)</td>
</tr>
<tr>
<td>(4y &gt; x \geq 2y + y \land y &gt; 0)</td>
<td>(q = 2 \land r = x - 2y \land a = 1 \land b = y)</td>
</tr>
</tbody>
</table>

*Symbolic states at L1*

- **Disjunctions of pathconds and locals**
  
  \[(x \geq y \land y > 0 \land q = 0 \land r = x \land a = 1 \land b = y) \lor (x \geq 2y \land y > 0 \land q = 0 \land r = x \land a = 2 \land b = 2y) \lor \ldots\]

- **An intermediate representation of states**

```c
int cohendiv(int x, int y){
    assert(x>0 && y>0);
    int q=0; int r=x;
    while(r \geq y){
        int a=1;
        int b=y;
        while[L1](r \geq 2*b){
            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
[L2]
    return q;
}
```
SymInfer: Invariants Inference using Symbolic States

- **Program**: Symbolic States
  - **Inputs**: Symbolic States
  - **Run**: Traces
    - **Infer**: Inputs
      - **CEX**: No
        - **Check**: Yes
          - **Invariants**
SymInfer: Invariants Inference using Symbolic States

- Use symbolic states for both inference and checking
- An iterative approach
  - Inferring: use symbolic states to generate traces, then apply DIG’s algorithms to infer numerical invariants from traces
  - Checking: use symbolic states to check candidate invariants and generate counterexample traces
Example: Dynamic Inference using DIG

```c
int cohendiv(int x, int y){
    assert(x>0 ; y>0);
    int q=0; int r=x;
    while(r >= y){
        int a=1; int b=y;
        while[L1](r >= 2*b){
            a = 2*a; b = 2*b;
        }
        r=r-b; q=q+a;
    }
    return q;
}
```

Traces:
\[
\begin{array}{cccccc}
 & x & y & a & b & q & r \\
1 & 15 & 2 & 1 & 2 & 0 & 15 \\
2 & 15 & 2 & 2 & 4 & 0 & 15 \\
3 & 15 & 2 & 1 & 2 & 4 & 7 \\
4 & 6 & 1 & 1 & 1 & 0 & 4 \\
5 & 6 & 1 & 2 & 2 & 0 & 4 \\
\end{array}
\]

Loop invariants at L1:

Equations:
\[
x = qy + r \\
b = ya \\
a = 2a + b = 2b \\
2 \leq a + y \leq b \\
y \leq b \leq r \leq x
\]
Example: Dynamic Inference using DIG

```c
int cohendiv(int x, int y)
{
    assert(x>0 ; y>0);
    int q=0; int r=x;
    while(r >= y){
        int a=1; int b=y;
        while[L1](r >= 2*b){
            a = 2*a; b = 2*b;
        }
        r=r-b; q=q+a;
    }
    return q;
}
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>a</th>
<th>b</th>
<th>q</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>
```

Loop invariants at L1:

Equations:

- `x = qy + r`
- `b = ya`

Inequalities:

- `2 ≤ a + y`
- `a ≤ b`
- `b ≤ r`
- `r ≤ x`
int cohendiv(int x, int y){
    assert(x>0 ; y>0);
    int q=0; int r=x;
    while(r >= y){
        int a=1; int b=y;
        while(L1)(r >= 2*b){
            a = 2*a; b = 2*b;
        }
        r=r-b; q=q+a;
    }
    return q;
}

Loop invariants at L1:

equations : \( x = qy + r \) \( b = ya \)

inequalities : \( 2 \leq a + y \) \( a \leq b \) \( y \leq b \)
\( b \leq r \) \( r \leq x \)
Infer Nonlinear Equations using Equation Solver

Terms and degrees

\[ V = \{ r, y, a \} \]
\[ \text{deg} = 2 \]

\[ T = \{ 1, r, y, a, ry, ra, ya, r^2, y^2, a^2 \} \]

Nonlinear equation template

\[ c_1 + c_2 r + c_3 y + c_4 a + c_5 ry + c_6 ra + c_7 ya + c_8 r^2 + c_9 y^2 + c_{10} a^2 = 0 \]

System of linear equations

trace 1 \rightarrow \{ r = 15, y = 2, a = 1 \}

\[ \begin{array}{c|cccc}
 x & y & a & b & q & r \\
 \hline
 15 & 2 & 1 & 2 & 0 & 15 \\
 15 & 2 & 2 & 4 & 0 & 15 \\
 15 & 2 & 1 & 2 & 4 & 7 \\
 4 & 1 & 1 & 1 & 0 & 4 \\
 4 & 1 & 2 & 2 & 0 & 4 \\
 \end{array} \]
Infer Nonlinear Equations using Equation Solver

Terms and degrees

\[ V = \{ r, y, a \}; \text{ deg } = 2 \]

\[ T = \{ 1, r, y, a, ry, ra, ya, r^2, y^2, a^2 \} \]

\[
\begin{array}{cccccc}
 x & y & | & a & b & q & r \\
 15 & 2 & | & 1 & 2 & 0 & 15 \\
 15 & 2 & | & 2 & 4 & 0 & 15 \\
 15 & 2 & | & 1 & 2 & 4 & 7 \\
 4 & 1 & | & 1 & 1 & 0 & 4 \\
 4 & 1 & | & 2 & 2 & 0 & 4 \\
\end{array}
\]
Infer Nonlinear Equations using Equation Solver

- Terms and degrees
  \[ V = \{ r, y, a \}; \quad \text{deg} = 2 \]
  \[ \downarrow \]
  \[ T = \{ 1, r, y, a, ry, ra, ya, r^2, y^2, a^2 \} \]

- Nonlinear equation template
  \[ c_1 + c_2 r + c_3 y + c_4 a + c_5 ry + c_6 ra + c_7 ya + c_8 r^2 + c_9 y^2 + c_{10} a^2 = 0 \]

\[
\begin{array}{c|cccc}
  x & y & a & b & q & r \\
  \hline
  15 & 2 & 1 & 2 & 0 & 15 \\
  15 & 2 & 2 & 4 & 0 & 15 \\
  15 & 2 & 1 & 2 & 4 & 7 \\
  \hline
  4 & 1 & 1 & 1 & 0 & 4 \\
  4 & 1 & 2 & 2 & 0 & 4 \\
\end{array}
\]
**Infer Nonlinear Equations using Equation Solver**

- **Terms and degrees**
  \[ V = \{r, y, a\}; \quad \text{deg} = 2 \]
  \[ T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\} \]

- **Nonlinear equation template**
  \[ c_1 + c_2 r + c_3 y + c_4 a + c_5 ry + c_6 ra + c_7 ya + c_8 r^2 + c_9 y^2 + c_{10} a^2 = 0 \]

- **System of linear equations**

  \[
  \begin{array}{c|cccc}
  \text{trace 1} & r & y & a & b & q & r \\
  \hline
  15 & 2 & 1 & 2 & 0 & 15 \\
  15 & 2 & 2 & 4 & 0 & 15 \\
  15 & 2 & 1 & 2 & 4 & 7 \\
  4 & 1 & 1 & 1 & 0 & 4 \\
  4 & 1 & 2 & 2 & 0 & 4 \\
  \end{array}
  \]

  \[
  \begin{align*}
  \text{eq 1} & \rightarrow \quad c_1 + 15 c_2 + 2 c_3 + c_4 + 30 c_5 + 15 c_6 + 2 c_7 + 225 c_8 + 4 c_9 + c_{10} = 0 \\
  \vdots & \end{align*}
  \]
Infer Nonlinear Equations using Equation Solver

- Terms and degrees
  \[ V = \{ r, y, a \}; \text{ deg} = 2 \]
  \[ T = \{ 1, r, y, a, ry, ra, ya, r^2, y^2, a^2 \} \]

- Nonlinear equation template
  \[ c_1 + c_2 r + c_3 y + c_4 a + c_5 ry + c_6 ra + c_7 ya + c_8 r^2 + c_9 y^2 + c_{10} a^2 = 0 \]

- System of linear equations
  \[ \text{trace 1} \rightarrow \{ r = 15, y = 2, a = 1 \} \]
  \[ \text{eq 1} \rightarrow c_1 + 15 c_2 + 2 c_3 + c_4 + 30 c_5 + 15 c_6 + 2 c_7 + 225 c_8 + 4 c_9 + c_{10} = 0 \]
  \[ \vdots \]

- Solve for coefficients \( c_i \)
  \[ V = \{ x, y, a, b, q, r \}; \text{ deg} = 2 \quad \rightarrow \quad x = qy + r, \; b = ya \]
Checking Using Symbolic States

General Idea

- **Goal**: prove/refute candidate invariants \((I)\) using symbolic states \((S)\)
- **Approach**: call SMT solver to check for validity of \(S \Rightarrow I\)
  - **valid**: invariant is valid and accepted
  - **invalid**: invariant is spurious and rejected, solver produces cex’s to help inference

Implementation: use JPF/SPF to obtain symbolic states bounded by depth \(k\): invariants only valid over symbolic states \(S\) computed with \(k\). If \(I\) is valid with \(S_k\), then check again if \(I\) is also valid with \(S_{k+1}\) to gain confidence.

Can be unsound (will not attempt all possible depths), but in practice is very effective in refuting bad invariants and finding cex’s.
Checking Using Symbolic States

General Idea

- **Goal**: prove/refute candidate invariants \( I \) using symbolic states \( S \)
- **Approach**: call SMT solver to check for validity of \( S \Rightarrow I \)
  - *valid*: invariant is valid and accepted
  - *invalid*: invariant is spurious and rejected, solver produces cex’s to help inference

Implementation: use JPF/SPF to obtain symbolic states

- Bounded by depth \( k \): invariants only valid over symbolic states \( S \) computed with \( k \)
- If \( I \) is valid with \( S_k \), then check again if \( I \) is also valid with \( S_{k+1} \) to gain confidence
- Can be *unsound* (will not attempt all possible depths), but in practice is *very effective* in refuting bad invariants and finding cex’s
Evaluation

Setup

- SymInfer works with Java programs, implemented in SAGE/Python (with JPF/SPF and Z3 SMT solver),
- Test machine: 10-core 2.4GHZ CPU, 128GB Ram, Linux OS

Benchmark (3 objectives)

1. Program Understanding: NLA testsuite, 27 programs with nonlinear invariants
2. Complexity Analysis: 19 programs collected from static complexity analysis work
3. Program Verification: HOLA benchmark, 46 programs with assertions, compare against PIE
Example: Program Understanding

```c
int cohendiv(int x, int y){
    assert(x>0 && y>0);
    int q=0; int r=x;
    while(r ≥ y){
        int a=1;
        int b=y;
        while[L1](r ≥ 2*b){
            a = 2*a;
            b = 2*b;
        }
        r=r-b;
        q=q+a;
    }
    [L2]
    return q;
}
```

What does this program do? What properties hold at L1 and L2?

SymInfer automatically generates

- loop invariants at L1:
  \[ x = qy + r, \quad b = ya, \quad y \leq b, \]
  \[ b \leq r, \quad r \leq x, \quad a \leq b, \quad 2 \leq a + y \]

- postconditions at L2:
  \[ x = qy + r, \quad 1 \leq q + r, \]
  \[ r \leq y - 1, \quad 0 \leq r, \quad r \leq x \]

- Invariants describe program’s semantic, e.g., integer division and reveal useful information, e.g., remainder is non-negative
### Results: Program Understanding

<table>
<thead>
<tr>
<th>Prog</th>
<th>Locs</th>
<th>Invs</th>
<th>Time (s)</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>cohendiv</td>
<td>2</td>
<td>10</td>
<td>21.05</td>
<td>✓</td>
</tr>
<tr>
<td>divbin</td>
<td>2</td>
<td>11</td>
<td>58.97</td>
<td>✓</td>
</tr>
<tr>
<td>manna</td>
<td>1</td>
<td>6</td>
<td>35.33</td>
<td>✓</td>
</tr>
<tr>
<td>hard</td>
<td>2</td>
<td>6</td>
<td>29.40</td>
<td>✓</td>
</tr>
<tr>
<td>sqrt1</td>
<td>1</td>
<td>5</td>
<td>20.03</td>
<td>✓</td>
</tr>
<tr>
<td>dijkstra</td>
<td>2</td>
<td>16</td>
<td>93.01</td>
<td>✓</td>
</tr>
<tr>
<td>freire1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>freire2</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cohen cu</td>
<td>1</td>
<td>4</td>
<td>21.90</td>
<td>✓</td>
</tr>
<tr>
<td>egcd1</td>
<td>1</td>
<td>14</td>
<td>122.22</td>
<td>✓</td>
</tr>
<tr>
<td>egcd2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>egcd3</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>prodbin</td>
<td>1</td>
<td>7</td>
<td>56.17</td>
<td>✓</td>
</tr>
<tr>
<td>prod4br</td>
<td>1</td>
<td>9</td>
<td>84.37</td>
<td>✓</td>
</tr>
<tr>
<td>knuth</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>fermat1</td>
<td>3</td>
<td>17</td>
<td>60.26</td>
<td>✓</td>
</tr>
<tr>
<td>fermat2</td>
<td>1</td>
<td>8</td>
<td>36.83</td>
<td>✓</td>
</tr>
<tr>
<td>lcm1</td>
<td>3</td>
<td>24</td>
<td>248.17</td>
<td>✓</td>
</tr>
<tr>
<td>lcm2</td>
<td>1</td>
<td>7</td>
<td>34.17</td>
<td>✓</td>
</tr>
<tr>
<td>geo1</td>
<td>1</td>
<td>8</td>
<td>158.27</td>
<td>✓</td>
</tr>
<tr>
<td>geo2</td>
<td>1</td>
<td>9</td>
<td>147.75</td>
<td>✓</td>
</tr>
<tr>
<td>geo3</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ps2</td>
<td>1</td>
<td>3</td>
<td>18.39</td>
<td>✓</td>
</tr>
<tr>
<td>ps3</td>
<td>1</td>
<td>3</td>
<td>19.69</td>
<td>✓</td>
</tr>
<tr>
<td>ps4</td>
<td>1</td>
<td>3</td>
<td>19.92</td>
<td>✓</td>
</tr>
<tr>
<td>ps5</td>
<td>1</td>
<td>3</td>
<td>46.19</td>
<td>✓</td>
</tr>
<tr>
<td>ps6</td>
<td>1</td>
<td>3</td>
<td>41.19</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Experiment

- **NLA suite**: 27 programs
- Require nonlinear invariants
- Use documented invariants (loop invariants and postconds) as ground truths
- **Goal**: obtain invariants and compare to ground truths
## Results: Program Understanding

<table>
<thead>
<tr>
<th>Prog</th>
<th>Locs</th>
<th>Invs</th>
<th>Time (s)</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>cohendiv</td>
<td>2</td>
<td>10</td>
<td>21.05</td>
<td>✓</td>
</tr>
<tr>
<td>divbin</td>
<td>2</td>
<td>11</td>
<td>58.97</td>
<td>✓</td>
</tr>
<tr>
<td>manna</td>
<td>1</td>
<td>6</td>
<td>35.33</td>
<td>✓</td>
</tr>
<tr>
<td>hard</td>
<td>2</td>
<td>6</td>
<td>29.40</td>
<td>✓</td>
</tr>
<tr>
<td>sqrt1</td>
<td>1</td>
<td>5</td>
<td>20.03</td>
<td>✓</td>
</tr>
<tr>
<td>dijkstra</td>
<td>2</td>
<td>16</td>
<td>93.01</td>
<td>✓</td>
</tr>
<tr>
<td>freire1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>freire2</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cohen cu</td>
<td>1</td>
<td>4</td>
<td>21.90</td>
<td>✓</td>
</tr>
<tr>
<td>egcd1</td>
<td>1</td>
<td>14</td>
<td>122.22</td>
<td>✓</td>
</tr>
<tr>
<td>egcd2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>egcd3</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>prodbin</td>
<td>1</td>
<td>7</td>
<td>56.17</td>
<td>✓</td>
</tr>
<tr>
<td>prod4br</td>
<td>1</td>
<td>9</td>
<td>84.37</td>
<td>✓</td>
</tr>
<tr>
<td>knuth</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>fermat1</td>
<td>3</td>
<td>17</td>
<td>60.26</td>
<td>✓</td>
</tr>
<tr>
<td>fermat2</td>
<td>1</td>
<td>8</td>
<td>36.83</td>
<td>✓</td>
</tr>
<tr>
<td>lcm1</td>
<td>3</td>
<td>24</td>
<td>248.17</td>
<td>✓</td>
</tr>
<tr>
<td>lcm2</td>
<td>1</td>
<td>7</td>
<td>34.17</td>
<td>✓</td>
</tr>
<tr>
<td>geo1</td>
<td>1</td>
<td>8</td>
<td>158.27</td>
<td>✓</td>
</tr>
<tr>
<td>geo2</td>
<td>1</td>
<td>9</td>
<td>147.75</td>
<td>✓</td>
</tr>
<tr>
<td>geo3</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ps2</td>
<td>1</td>
<td>3</td>
<td>18.39</td>
<td>✓</td>
</tr>
<tr>
<td>ps3</td>
<td>1</td>
<td>3</td>
<td>19.69</td>
<td>✓</td>
</tr>
<tr>
<td>ps4</td>
<td>1</td>
<td>3</td>
<td>19.92</td>
<td>✓</td>
</tr>
<tr>
<td>ps5</td>
<td>1</td>
<td>3</td>
<td>46.19</td>
<td>✓</td>
</tr>
<tr>
<td>ps6</td>
<td>1</td>
<td>3</td>
<td>41.19</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Experiment

- **NLA suite**: 27 programs
- Require nonlinear invariants
- Use documented invariants (loop invariants and postconds) as ground truths
- **Goal**: obtain invariants and compare to ground truths

### Results: SymInfer found correct invariants in 21/27 (✓) programs
- Most results equivalent to or stronger than (imply) ground truths
- Several unexpected and undocumented invariants
- Some invariants reveal “how” program works in details
Example: Complexity Analysis

void triple(int M, int N, int P){  Complexity of this program?
    assert (0 <= M);
    assert (0 <= N);
    assert (0 <= P);
    int i = 0, j = 0, k = 0;
    int t = 0;
    while(i < N){
        j = 0; t++;
        while(j < M){
            j++; k = i; t++;
            while (k < P){
                k++; t++;
            }
            i = k;
        }
        i++;
    }
    [L]
}
void triple(int M, int N, int P){
    assert (0 <= M);
    assert (0 <= N);
    assert (0 <= P);
    int i = 0, j = 0, k = 0;
    int t = 0;
    while(i < N){
        j = 0; t++;
        while(j < M){
            j++; k = i; t++;
            while (k < P){
                k++; t++;
            }
        i = k;
        }
    i++;
    }
    [L]
}

Complexity of this program?

- Use $t$ to count loop iterations
- At first glance: $t = O(MNP)$
- A more precise complexity bound: $t = O(N + NM + P)$
Example: Complexity Analysis

```c
void triple(int M, int N, int P){
    assert (0 <= M);
    assert (0 <= N);
    assert (0 <= P);
    int i = 0, j = 0, k = 0;
    int t = 0;
    while(i < N){
        j = 0; t++;
        while(j < M){
            j++; k = i; t++;
            while (k < P){
                k++; t++;
            }
            i = k;
        }
        i++;
    }
    [L]
}
```

Complexity of this program?

- Use \( t \) to count loop iterations
- At first glance: \( t = O(MNP) \)
- A more precise complexity bound: \( t = O(N + NM + P) \)
- SymInfer found a very unexpected inv:

\[
P^2Mt + PM^2t - PMNt - M^2Nt - PMt^2 + Mnt^2 + PMt - Pnt - 2Mnt + Pt^2 + Mt^2 + Nt^2 - t^3 - Nt + t^2 = 0
\]

Nonlinear invariants can represent disjunctive properties capturing different complexity bounds.
Example: Complexity Analysis

```c
void triple(int M, int N, int P){
    assert (0 <= M);
    assert (0 <= N);
    assert (0 <= P);
    int i = 0, j = 0, k = 0;
    int t = 0;
    while(i < N){
        j = 0; t++;
        while(j < M){
            j++; k = i; t++;
            while (k < P){
                k++; t++;
            }
            i = k;
        }
        i++;
    }
    [L]
}
```

Complexity of this program?

- Use \( t \) to count loop iterations
- At first glance: \( t = O(MNP) \)
- A more precise complexity bound: \( t = O(N + NM + P) \)
- SymInfer found a very unexpected inv:
  \[
  P^2 M t + PM^2 t - PMN t - M^2 N t - PMt^2 + MNt^2 + PMt - P N t - 2MNt + Pt^2 + Mt^2 + N t^2 - t^3 - N t + t^2 = 0
  \]
  Solve for \( t \) yields the most precise, unpublished bound:
  \[
  \begin{align*}
  t &= 0 & \text{when } N &= 0, \\
  t &= P + M + 1 & \text{when } N &\leq P, \\
  t &= N - M(P - N) & \text{when } N &> P
  \end{align*}
  \]
- Nonlinear invariants can represent disjunctive properties capturing different complexity bounds
## Results: Complexity Analysis

### Experiment

- 19 progs from static complexity work
- Obtain postconds representing complexity
- **Goal**: compare against results from prev work

<table>
<thead>
<tr>
<th>Prog</th>
<th>Invs</th>
<th>Time (s)</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>cav09_fig1a</td>
<td>1</td>
<td>12.41</td>
<td>✓</td>
</tr>
<tr>
<td>cav09_fig1d</td>
<td>1</td>
<td>12.44</td>
<td>✓</td>
</tr>
<tr>
<td>cav09_fig2d</td>
<td>3</td>
<td>58.40</td>
<td>✓</td>
</tr>
<tr>
<td>cav09_fig3a</td>
<td>3</td>
<td>8.75</td>
<td>✓</td>
</tr>
<tr>
<td>cav09_fig5b</td>
<td>6</td>
<td>49.44</td>
<td>✓</td>
</tr>
<tr>
<td>pldi09_ex6</td>
<td>6</td>
<td>57.00</td>
<td>✓</td>
</tr>
<tr>
<td>pldi09_fig2</td>
<td>6</td>
<td>60.60</td>
<td>✓✓</td>
</tr>
<tr>
<td>pldi09_fig4_1</td>
<td>3</td>
<td>56.24</td>
<td>✓</td>
</tr>
<tr>
<td>pldi09_fig4_2</td>
<td>5</td>
<td>28.32</td>
<td>✓</td>
</tr>
<tr>
<td>pldi09_fig4_3</td>
<td>3</td>
<td>59.19</td>
<td>✓</td>
</tr>
<tr>
<td>pldi09_fig4_4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>pldi09_fig4_5</td>
<td>3</td>
<td>103.70</td>
<td>✓</td>
</tr>
<tr>
<td>popl09_fig2_1</td>
<td>2</td>
<td>50.86</td>
<td>✓✓</td>
</tr>
<tr>
<td>popl09_fig2_2</td>
<td>2</td>
<td>53.48</td>
<td>✓✓</td>
</tr>
<tr>
<td>popl09_fig3_4</td>
<td>4</td>
<td>58.62</td>
<td>✓</td>
</tr>
<tr>
<td>popl09_fig4_1</td>
<td>4</td>
<td>65.19</td>
<td>✓</td>
</tr>
<tr>
<td>popl09_fig4_2</td>
<td>2</td>
<td>51.24</td>
<td>✓✓</td>
</tr>
<tr>
<td>popl09_fig4_3</td>
<td>5</td>
<td>31.57</td>
<td>✓</td>
</tr>
<tr>
<td>popl09_fig4_4</td>
<td>3</td>
<td>36.89</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Results: Complexity Analysis

<table>
<thead>
<tr>
<th>Prog</th>
<th>Invs</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cav09_fig1a</td>
<td>1</td>
<td>12.41</td>
</tr>
<tr>
<td>cav09_fig1d</td>
<td>1</td>
<td>12.44</td>
</tr>
<tr>
<td>cav09_fig2d</td>
<td>3</td>
<td>58.40</td>
</tr>
<tr>
<td>cav09_fig3a</td>
<td>3</td>
<td>8.75</td>
</tr>
<tr>
<td>cav09_fig5b</td>
<td>6</td>
<td>49.44</td>
</tr>
<tr>
<td>pldi09_ex6</td>
<td>6</td>
<td>57.00</td>
</tr>
<tr>
<td>pldi09_fig2</td>
<td>6</td>
<td>60.60</td>
</tr>
<tr>
<td>pldi09_fig4_1</td>
<td>3</td>
<td>56.24</td>
</tr>
<tr>
<td>pldi09_fig4_2</td>
<td>5</td>
<td>28.32</td>
</tr>
<tr>
<td>pldi09_fig4_3</td>
<td>3</td>
<td>59.19</td>
</tr>
<tr>
<td>pldi09_fig4_4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>pldi09_fig4_5</td>
<td>3</td>
<td>103.70</td>
</tr>
<tr>
<td>popl09_fig2_1</td>
<td>2</td>
<td>50.86</td>
</tr>
<tr>
<td>popl09_fig2_2</td>
<td>2</td>
<td>53.48</td>
</tr>
<tr>
<td>popl09_fig3_4</td>
<td>4</td>
<td>58.62</td>
</tr>
<tr>
<td>popl09_fig4_1</td>
<td>4</td>
<td>65.19</td>
</tr>
<tr>
<td>popl09_fig4_2</td>
<td>2</td>
<td>51.24</td>
</tr>
<tr>
<td>popl09_fig4_3</td>
<td>5</td>
<td>31.57</td>
</tr>
<tr>
<td>popl09_fig4_4</td>
<td>3</td>
<td>36.89</td>
</tr>
</tbody>
</table>

### Experiment
- 19 progs from static complexity work
- Obtain postconds representing complexity
- **Goal**: compare against results from prev work

### Results: Obtain equivalent (14 ✓) or more precise bounds (4 ✓✓) in 18/19 progs
Example: Verification

```c
void f(int u1, int u2) {
    assert(u1 > 0 && u2 > 0);
    int a = 1, b = 1, c = 2, d = 2;
    int x = 3, y = 3;
    int i1 = 0, i2 = 0;
    while (i1 < u1) {
        i1++;
        x = a + c; y = b + d;
        if ((x + y) % 2 == 0) {
            a++; d++;
        } else { a--;
        }
        i2 = 0;
        while (i2 < u2) {
            i2++; c--; b--;
        }
    }
    [L] //SymInfer found:
    //b + 1 = c, a + 1 = d,
    //a + b <= 2, 2 <= a
    assert(a + c == b + d);
}
```

```c
void g(int n, int u1) {
    assert(u1 > 0);
    int x = 0;
    int m = 0;
    while (x < n) {
        if (u1) {
            m = x;
        }
        x = x + 1;
    }
    [L] //SymInfer found:
    //m^2 = nx - m - x, mn = x^2 - x
    // -m <= x, x <= m + 1, n <= x
    if (n > 0){
        assert(0 <= m && m < n);
    }
}
```
Results: Verification

Experiment

- HOLA benchmark: 46 programs
- Various assertions (mostly postconds)

Goal:

- Obtain and compare invariants: if match or imply assertions, then assertions hold
- Also compare with existing tool PIE
Results: Verification

Experiment

- HOLA benchmark: 46 programs
- Various assertions (mostly postconds)

Goal:

- Obtain and compare invariants: if match or imply assertions, then assertions hold
- Also compare with existing tool PIE

Results: Found equiv or stronger invariants in 40/46 programs

- Time: median 9.3s, mean 5.4s
- Nonlinear invariants can prove many nontrivial and unsupported properties
documentation, code, benchmark programs

https://bitbucket.org/nguyenthanhvuh/symtraces/
Inferring Octagonal Inequalities

- Basic CEGIR does not work well for inequalities (e.g., $t \leq 1000$)
  - E.g., real inv: $t \leq 1000$
  - Basic CEGIR: iter 1: $t \leq 2$, iter: 2 $t \leq 3$, iter 3: $t \leq 7$, ...
  - Not terminating if $t$ has no bounds
Inferring Octagonal Inequalities

- Basic CEGIR does not work well for inequalities (e.g., $t \leq 1000$)
  - E.g., real inv: $t \leq 1000$
  - Basic CEGIR: iter 1: $t \leq 2$, iter: 2 $t \leq 3$, iter 3: $t \leq 7$, ...
  - Not terminating if $t$ has no bounds

- **Approach**: Divide and Conquer
  - Only consider invariants within fixed bounds, e.g., $-k \leq t \leq k$, where $k = 100000$
Inferring Octagonal Inequalities

- Basic CEGIR does not work well for inequalities (e.g., $t \leq 1000$)
  - E.g., real inv: $t \leq 1000$
  - Basic CEGIR: iter 1: $t \leq 2$, iter 2: $t \leq 3$, iter 3: $t \leq 7$, ...
  - Not terminating if $t$ has no bounds

- **Approach**: Divide and Conquer
  - Only consider invariants within fixed bounds, e.g., $-k \leq t \leq k$, where $k = 100000$
  - Finding upperbound (ub): check if $t \leq k$
  - If no (i.e., $t > k$) then will not find ub of $t$
Inferring Octagonal Inequalities

- Basic CEGIR does not work well for inequalities (e.g., $t \leq 1000$)
  - E.g., real inv: $t \leq 1000$
  - Basic CEGIR: iter 1: $t \leq 2$, iter: 2 $t \leq 3$, iter 3: $t \leq 7$, ...
  - Not terminating if $t$ has no bounds

- **Approach:** Divide and Conquer
  - Only consider invariants within fixed bounds, e.g., $-k \leq t \leq k$, where $k = 100000$
  - Finding upperbound (ub): check if $t \leq k$
  - If no (i.e., $t > k$) then will not find ub of $t$
  - Otherwise use divide and conquer to find ub of $t$ within range $[-k, k]$
    - Compute mid value $mv = (-k + k)/2$, check if $t \leq mv$
    - If yes, find ub of $t$ within the range $[-k, mv]$
    - If no, find ub of $t$ within the range $[mv, k]$
Inferring Octagonal Inequalities

- Basic CEGIR does not work well for inequalities (e.g., $t \leq 1000$)
  - E.g., real inv: $t \leq 1000$
  - Basic CEGIR: iter 1: $t \leq 2$, iter: 2 $t \leq 3$, iter 3: $t \leq 7$, ...
  - Not terminating if $t$ has no bounds

- **Approach**: Divide and Conquer
  - Only consider invariants within fixed bounds, e.g., $-k \leq t \leq k$, where $k = 100000$
  - Finding upperbound (ub): check if $t \leq k$
  - If no (i.e., $t > k$) then will not find ub of $t$
  - Otherwise use divide and conquer to find ub of $t$ within range $[-k, k]$
    - Compute mid value $mv = (-k + k)/2$, check if $t \leq mv$
    - If yes, find ub of $t$ within the range $[-k, mv]$
    - If no, find ub of $t$ within the range $[mv, k]$

- Support **octagonal** invariants: term $t$ represent $x, y, x - y, x + y, -x - y, ...$
Using Symbolic States for Invariant Inference

- Reusability: pre-compute and reuse symbolic states at $L$, e.g., for checking
- Expressiveness: a symbolic state (e.g., $x \geq 0, y \geq x$) represents many concrete states and also encodes relationships among variables (e.g., $y \geq x$)
- Diversity: each symbolic state represents a different program “path”, produce better traces
- Usability and Optimization: encoded logical formulas, checked with different solvers and optimized (e.g., perform slicing when checking)