3.33pt

Counterexample-guided Approach to Finding Numerical Invariants

ThanhVu (Vu) Nguyen*,

Timos Antonopoulos†, Andrew Ruef‡, Michael Hicks‡

*University of Nebraska, †Yale University, ‡University of Maryland

FSE 2017

Introduction

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

- Assertions
- Pre/Post conditions
- Loop invariants

Introduction

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

- Assertions
- Pre/Post conditions
- Loop invariants

Techniques for automatic invariant generation

- *Static*: examine program code, compute sound results, but can be expensive and limited to simple invariants
- *Dynamic*: analyze exec traces, produce expressive invariants, but unsound

Introduction

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

- Assertions
- Pre/Post conditions
- Loop invariants

Techniques for automatic invariant generation

- *Static*: examine program code, compute sound results, but can be expensive and limited to simple invariants
- *Dynamic*: analyze exec traces, produce expressive invariants, but unsound

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: $x \leq y^2, x = qy + r, \ldots$

Invariants can help understanding programs

```
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?

Invariants can help understanding programs

```
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?

- loop invariants at L1:

$$\begin{array}{ll} x = qy + r & b = ya \\ y \leq b & b \leq r \\ r \leq x & a \leq b \\ 2 \leq a + y & \end{array}$$

- postconditions at L2:

$$\begin{array}{ll} x = qy + r & \\ 1 \leq q + r & r \leq y - 1 \\ 0 \leq r & r \leq x \end{array}$$

Invariants can help understanding programs

```
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?

- loop invariants at L1:

$$\begin{array}{ll} x = qy + r & b = ya \\ y \leq b & b \leq r \\ r \leq x & a \leq b \\ 2 \leq a + y & \end{array}$$

- postconditions at L2:

$$\begin{array}{ll} x = qy + r & \\ 1 \leq q + r & r \leq y - 1 \\ 0 \leq r & r \leq x \end{array}$$

Describe the semantic the program (e.g., $x = qy + r$ for integer division) and reveal useful information (e.g., remainder r is non-negative)

Invariants can help analyze program complexities

```
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

Invariants can help analyze program complexities

```
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: $t = O(N + NM + P)$
- Both are nonlinear invariants

Invariants can help verify programs

```
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]
  assert(a + c == b + d);
}
```

```
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]
  if (n > 0){
    assert(0 <= m && m < n);
  }
}
```

Invariants can help verify programs

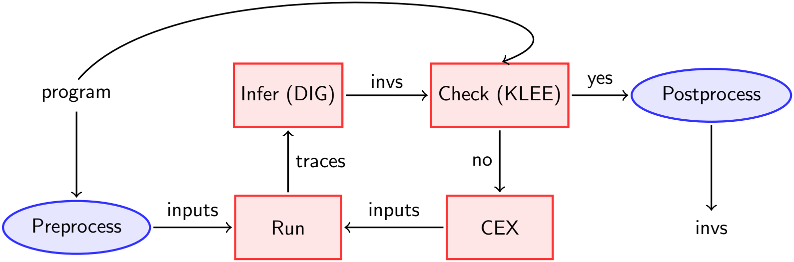
```
void f(int u1, int u2) {
  assert(u1 > 0 && u2 > 0);
  int a = 1, b = 1, c = 2, d = 2;
  int x = 3, y = 3;
  int i1 = 0, i2 = 0;
  while (i1 < u1) {
    i1++;
    x = a + c; y = b + d;
    if ((x + y) % 2 == 0) {
      a++; d++;
    } else { a--;}
    i2 = 0;
    while (i2 < u2 ) {
      i2++; c--;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

```
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]
  if (n > 0){
    assert(0 <= m && m < n);
  }
}
```

Assertions hold if matched or implied by discovered invariants at L

NumInv: a CEGIR approach to numerical invariants



- Focus on polynomial invariants over numerical variables
- Use CounterExample-Guided Invariant Generation (CEGIR) approach
  - Dynamic Inferrence: use DIG's algorithms to infer *nonlinear equalities* and *linear inequalities* from traces
  - Static Checking: use KLEE to check candiate invariants and generate counterexample inputs

Example: Dynamic Inference using DIG

```
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;
}
```

Example: Dynamic Inference using DIG

```
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:

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
			⋮		

Example: Dynamic Inference using DIG

```
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:

x	y	a	b	q	r
15	2	1	2	0	15
15	2	2	4	0	15
15	2	1	2	4	7
			$\vdots$		
4	1	1	1	0	4
4	1	2	2	0	4
			$\vdots$		

Loop invariants at L1:

$$\begin{array}{llll} \textit{equations}: & x = qy + r & b = ya & \\ \textit{inequalities}: & 2 \leq a + y & a \leq b & y \leq b \\ & b \leq r & r \leq x & \end{array}$$

Infer Nonlinear Equations using Equation Solver

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

Infer Nonlinear Equations using Equation Solver

- Terms and degrees

$$V = \{r, y, a\};\ \texttt{deg} = 2$$
$$\downarrow$$
$$T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\}$$

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

Infer Nonlinear Equations using Equation Solver

- Terms and degrees

$$V = \{r, y, a\};\ \texttt{deg} = 2$$
$$\downarrow$$
$$T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\}$$

- Nonlinear equation template

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

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

Infer Nonlinear Equations using Equation Solver

- Terms and degrees

$$V = \{r, y, a\};\ \texttt{deg} = 2$$
$$\downarrow$$
$$T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\}$$

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

- 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

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

Infer Nonlinear Equations using Equation Solver

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

- Terms and degrees

$$V = \{r, y, a\};\ \mathtt{deg} = 2$$
$$\downarrow$$
$$T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\}$$

- Nonlinear equation template

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

- System of *linear* equations

$$\begin{array}{rcl} \text{trace } 1 & \rightarrow & \{r = 15, y = 2, a = 1\} \\ \text{eq } 1 & \rightarrow & c_1 + 15c_2 + 2c_3 + c_4 + 30c_5 + 15c_6 + 2c_7 + 225c_8 + 4c_9 + c_{10} = 0 \\ & \vdots & \end{array}$$

- Solve for coefficients c_i

$$V = \{x, y, a, b, q, r\};\ \mathtt{deg} = 2 \quad \longrightarrow \quad x = qy + r, b = ya$$

Static Checking

- **Goal**: prove/refute candidate invariants using program code
- **Approach**: reduce invariant checking to reachability

Static Checking

- **Goal**: prove/refute candidate invariants using program code
- **Approach**: reduce invariant checking to reachability
  - Transform program and invariant into another program consist of a special location L'

```
...
[L] //is x=qy+r valid?
...
```

$\Longrightarrow$

```
...
if (!(x==qy+r)){
  [L′] //x=qy+r is invalid
  abort();
}
//x=qy+r is valid
...
```

Static Checking

- **Goal**: prove/refute candidate invariants using program code
- **Approach**: reduce invariant checking to reachability
  - Transform program and invariant into another program consist of a special location L'

```
...
[L] //is x=qy+r valid?
...
```

$\Longrightarrow$

```
...
if (!(x==qy+r)){
  [L′] //x=qy+r is invalid
  abort();
}
//x=qy+r is valid
...
```

  - L' reachable $\implies$ inv is spurious (inputs reaching L' represent cex's)
  - L' not reachable (within a time bound) $\implies$ NumInv *accepts* the invariant

Static Checking

- **Goal**: prove/refute candidate invariants using program code
- **Approach**: reduce invariant checking to reachability
  - Transform program and invariant into another program consist of a special location L'

```
...
[L] //is x=qy+r valid?
...
```

$\Longrightarrow$

```
...
if (!(x==qy+r)){
  [L'] //x=qy+r is invalid
  abort();
}
//x=qy+r is valid
...
```

  - L' reachable $\Longrightarrow$ inv is spurious (inputs reaching L' represent cex's)
  - L' not reachable (within a time bound) $\Longrightarrow$ NumInv *accepts* the invariant
- Use the symbolic execution tool KLEE to check reachability
  - *Unsound*: KLEE can timeout, but in practice is *very effective* in refuting bad invariants and finding cex's
  - Can use other test-input generation tools or verifiers instead of KLEE

Evaluation

Setup

- NumInv is implemented in SAGE/Python (with Z3 backend solver)
- Test machine: 10-core 2.4GHZ CPU, 128GB Ram, Linux OS

Benchmark

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

Example: Program Understanding

```
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?

Example: Program Understanding

```
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?

- loop invariants at L1:

$$
\begin{array}{ll}
x = qy + r & b = ya \\
y \leq b & b \leq r \\
r \leq x & a \leq b \\
2 \leq a + y &
\end{array}
$$

- postconditions at L2:

$$
\begin{array}{ll}
x = qy + r & \\
1 \leq q + r & r \leq y - 1 \\
0 \leq r & r \leq x
\end{array}
$$

Example: Program Understanding

```
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?

- loop invariants at L1:

$$\begin{array}{ll} x = qy + r & b = ya \\ y \leq b & b \leq r \\ r \leq x & a \leq b \\ 2 \leq a + y & \end{array}$$

- postconditions at L2:

$$\begin{array}{ll} x = qy + r & \\ 1 \leq q + r & r \leq y - 1 \\ 0 \leq r & r \leq x \end{array}$$

Indicate the exact semantic of integer division and reveal other useful correctness information (e.g., remainder is non-negative)

Results: Program Understanding

Prog	Locs	Invs	Time (s)	Correct
cohendiv	2	11	24.5	✓
divbin	2	12	116.8	✓
manna	1	5	30.8	✓
hard	2	13	71.4	✓
sqrt1	1	5	19.3	✓
dijkstra	2	14	89.3	✓
freire1	1	-	-	-
freire2	1	-	-	-
cohencu	1	5	22.5	✓
egcd1	1	9	284.5	✓
egcd2	2	-	-	-
egcd3	3	-	-	-
prodbin	1	7	45.1	✓
prod4br	1	11	87.3	✓
knuth	1	9	84.6	✓
fermat1	3	26	185.3	✓
fermat2	1	8	101.8	✓
lcm1	3	22	175.2	✓
lcm2	1	7	163.8	✓
geo1	1	7	24.4	✓
geo2	1	9	24.3	✓
geo3	1	7	32.3	✓
ps2	1	3	17.0	✓
ps3	1	4	17.8	✓
ps4	1	4	18.5	✓
ps5	1	4	19.3	✓
ps6	1	3	21.0	✓

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

Prog	Locs	Invs	Time (s)	Correct
cohendiv	2	11	24.5	✓
divbin	2	12	116.8	✓
manna	1	5	30.8	✓
hard	2	13	71.4	✓
sqrt1	1	5	19.3	✓
dijkstra	2	14	89.3	✓
freire1	1	-	-	-
freire2	1	-	-	-
cohencu	1	5	22.5	✓
egcd1	1	9	284.5	✓
egcd2	2	-	-	-
egcd3	3	-	-	-
prodbin	1	7	45.1	✓
prod4br	1	11	87.3	✓
knuth	1	9	84.6	✓
fermat1	3	26	185.3	✓
fermat2	1	8	101.8	✓
lcm1	3	22	175.2	✓
lcm2	1	7	163.8	✓
geo1	1	7	24.4	✓
geo2	1	9	24.3	✓
geo3	1	7	32.3	✓
ps2	1	3	17.0	✓
ps3	1	4	17.8	✓
ps4	1	4	18.5	✓
ps5	1	4	19.3	✓
ps6	1	3	21.0	✓

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: NumInv found correct invariants in 23/27 progs

- Most results equiv 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){
  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?

- Existing result: $t = O(N + NM + P)$

Example: Complexity Analysis

```
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?

- Existing result: $t = O(N + NM + P)$
- NumInv found a very *unexpected* inv:

$$P^2Mt + PM^2t - PMNt - M^2Nt$$
$$-PMt^2 + MNt^2 + PMt - PNt - 2MNt$$
$$+Pt^2 + Mt^2 + Nt^2 - t^3 - Nt + t^2 = 0$$

Example: Complexity Analysis

```
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?

- Existing result: $t = O(N + NM + P)$
- NumInv found a very *unexpected* inv:

$$\begin{aligned} P^2Mt + PM^2t - PMNt - M^2Nt \\ -PMt^2 + MNt^2 + PMt - PNt - 2MNt \\ +Pt^2 + Mt^2 + Nt^2 - t^3 - Nt + t^2 = 0 \end{aligned}$$

- Solve for t yields the most precise, unpublished bound:

$$\begin{aligned} t &= 0 && \text{when} \quad N = 0, \\ t &= P + M + 1 && \text{when} \quad N \leq P, \\ t &= N - M(P - N) && \text{when} \quad N > P \end{aligned}$$

- Nonlinear invariants can represent *disjunctive properties* capturing different complexity bounds

Results: Complexity Analysis

Prog	Invs	Time (s)	
cav09_fig1a	1	14.3	✓
cav09_fig1d	1	14.2	✓
cav09_fig2d	3	36.0	✓
cav09_fig3a	3	14.2	✓
cav09_fig5b	5	46.8	✓
pldi09_ex6	7	54.1	✓
pldi09_fig2 (triple)	6	93.5	✓✓
pldi09_fig4_1	3	44.2	✓
pldi09_fig4_2	5	43.7	✓
pldi09_fig4_3	3	37.5	✓
pldi09_fig4_4	4	56.6	-
pldi09_fig4_5	3	31.6	✓
popl09_fig2_1	2	211.7	✓✓
popl09_fig2_2	2	65.1	✓✓
popl09_fig3_4	4	54.7	✓
popl09_fig4_1	2	42.7	✓
popl09_fig4_2	2	158.3	✓✓
popl09_fig4_3	5	39.2	✓
popl09_fig4_4	3	34.2	✓

Experiment

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

Results: Complexity Analysis

Prog	Invs	Time (s)	
cav09_fig1a	1	14.3	✓
cav09_fig1d	1	14.2	✓
cav09_fig2d	3	36.0	✓
cav09_fig3a	3	14.2	✓
cav09_fig5b	5	46.8	✓
pldi09_ex6	7	54.1	✓
pldi09_fig2 (triple)	6	93.5	✓✓
pldi09_fig4_1	3	44.2	✓
pldi09_fig4_2	5	43.7	✓
pldi09_fig4_3	3	37.5	✓
pldi09_fig4_4	4	56.6	-
pldi09_fig4_5	3	31.6	✓
popl09_fig2_1	2	211.7	✓✓
popl09_fig2_2	2	65.1	✓✓
popl09_fig3_4	4	54.7	✓
popl09_fig4_1	2	42.7	✓
popl09_fig4_2	2	158.3	✓✓
popl09_fig4_3	5	39.2	✓
popl09_fig4_4	3	34.2	✓

Experiment

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

Results: Obtain equiv (14) or more precise bounds (4) in 18/19 progs

Example: Verification

```
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] //NumInv found:
  //b + 1 = c, a + 1 = d,
  //a + b <= 2, 2 <= a
  assert(a + c == b + d);
}
```

```
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] //NumInv 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 (23) or stronger (13) invariants in 36/46 programs

- Time: mean 30s, median 13s
- Nonlinear invariants can prove many nontrivial and *unsupported* properties

Conclusion

NumInv

- Use CEGIR for *numerical* invariant generation
  - Dynamic Inference: use DIG to compute nonlinear invariants
  - Static Checking: use KLEE to check candidate invariants and obtain cex's
- *Unsound*, but experience shows practical and effective in removing invalid results and can handle complex invariants

Conclusion

NumInv

- Use CEGIR for *numerical* invariant generation
  - Dynamic Inference: use DIG to compute nonlinear invariants
  - Static Checking: use KLEE to check candidate invariants and obtain cex's
- *Unsound*, but experience shows practical and effective in removing invalid results and can handle complex invariants

Results

- Discover necessary nonlinear invariants to understand programs
- Find useful invariants capturing nontrivial runtime complexity
- Compete well with existing work
- General polynomial invariants (e.g., nonlinear properties) can *surprisingly* represent/prove many nontrivial, complex, and *unsupported* properties

`https://bitbucket.org/nguyenthanhvuh/dig2/`