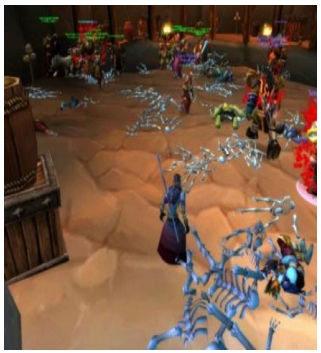
Automatic Invariant Generation and Program Repair

ThanhVu (Vu) Nguyen

University of Maryland, College Park

The Problem

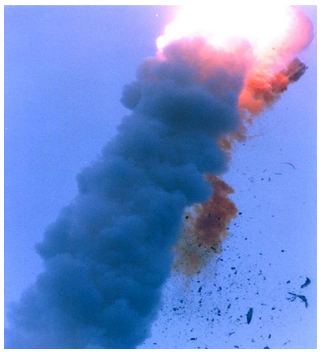
SOFTWARE BUGS



World of Warcraft bug

Therac-25 machines

X-rays overdose



Ariane-5 rocket self-destructs

North America blackout

The Cost

– Mozilla Developer

"Everyday, almost 300 bugs appear [..] far too many for only the Mozilla programmers to handle."

Software bugs annually cost 0.6% of the U.S GDP and $312 billion to the global economy

Average time to fix a security-critical error: 28 days

Program Analysis

Write Code

Check Code

Program Analysis

Write Code


Check Code


Automated program analysis techniques and tools can decrease debugging time by an average of 26% and $41 billion annually

Program Synthesis


Generate a program that meets a given specification

Program Verification


Check if a program satisfies a given specification

Invariant Generation and Template-based Synthesis

Invariant Generation

```
def intdiv(x, y):
  q = 0
  r = x
  while r ≥ y:
    a = 1
    b = y
    while [??] r ≥ 2b:
      a = 2a
      b = 2b
    r = r - b
    q = q + a
    [??]
    return q
```

- Discover invariant properties at certain program locations
- Answer the question *"what does this program do ?"*

Invariant Generation and Template-based Synthesis

Invariant Generation

```
def intdiv(x, y):
  q = 0
  r = x
  while r ≥ y:
    a = 1
    b = y
    while [??] r ≥ 2b:
      a = 2a
      b = 2b
    r = r - b
    q = q + a
    [??]
    return q
```

- Discover invariant properties at certain program locations
- Answer the question *"what does this program do ?"*

Template-based Synthesis

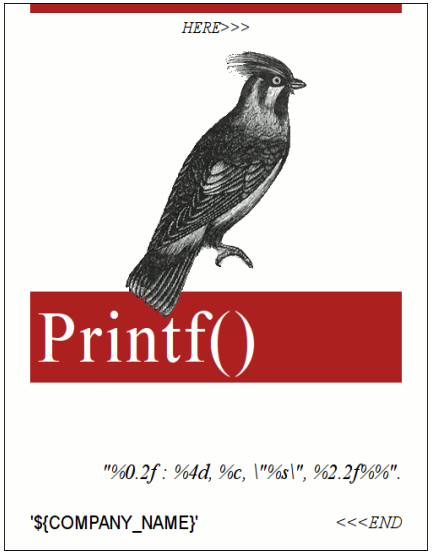
```
def intdiv(x, y):
  q = 0
  r = x
  while r [**] y:
    a = 1
    b = [**]
    while r ≥ 2b:
      a = [**]
      b = 2b
    r = r - b
    q = q + a

  return [**]
```

- Create code under specific templates from partially completed programs
- Can be used for automatic program repair

Outline

How We Analyze Programs

How We Analyze Programs

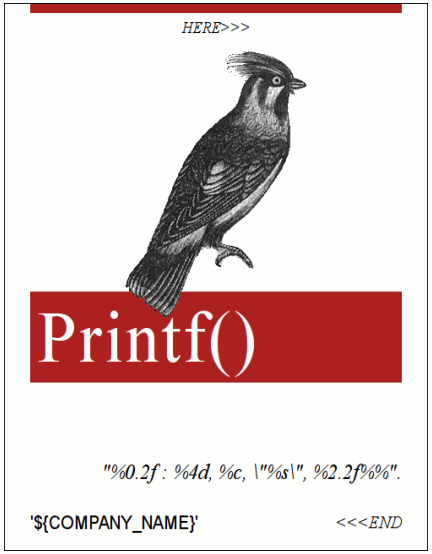



```
File  Edit  Options  Buffers  Tools  Help

def intdiv(x, y):

    q = 0
    r = x
    while r >= y:
        a = 1
        b = y

        while r >= 2*b:
            a = 2 * a
            b = 2 * b
        r = r - b
        q = q + a

    print "x %d, y %d, q %d, r %d" %(x,y,q,r)
    return q,r
```

```
x   0,  y   1,  q    0,  r  0
x   1,  y   1,  q    1,  r  0
x   1,  y   5,  q    0,  r  1
x   1,  y  10,  q    0,  r  1
x   3,  y   1,  q    3,  r  0
x   3,  y   4,  q    0,  r  3
x   3,  y   7,  q    0,  r  3
x   8,  y   1,  q    8,  r  0
x   8,  y   2,  q    4,  r  0
x   8,  y   9,  q    0,  r  8
x   8,  y  10,  q    0,  r  8
x  15,  y   1,  q   15,  r  0
x  15,  y   5,  q    3,  r  0
x  15,  y   7,  q    2,  r  1
x  20,  y   2,  q   10,  r  0
x  20,  y   7,  q    2,  r  6
x  20,  y  10,  q    2,  r  0
x 100,  y   1,  q  100,  r  0
x 100,  y   5,  q   20,  r  0
x 100,  y  10,  q   10,  r  0
```

```
-U:---  intdiv.py      All (18,0)     (Python)--5:
-U:---  intdiv.traces  All (21,0)
```

How We Analyze Programs




```
File  Edit  Options  Buffers  Tools  Help

def intdiv(x, y):

    q = 0
    r = x
    while r >= y:
        a = 1
        b = y

        while r >= 2*b:
            a = 2 * a
            b = 2 * b
        r = r - b
        q = q + a

    print "x %d, y %d, q %d, r %d" %(x,y,q,r)
    return q,r

-U:---  intdiv.py      All (18,0)     (Python)--5:
```

```
x   0,  y   1,  q    0,  r  0
x   1,  y   1,  q    1,  r  0
x   1,  y   5,  q    0,  r  1
x   1,  y  10,  q    0,  r  1
x   3,  y   1,  q    3,  r  0
x   3,  y   4,  q    0,  r  3
x   3,  y   7,  q    0,  r  3
x   8,  y   1,  q    8,  r  0
x   8,  y   2,  q    4,  r  0
x   8,  y   9,  q    0,  r  8
x   8,  y  10,  q    0,  r  8
x  15,  y   1,  q   15,  r  0
x  15,  y   5,  q    3,  r  0
x  15,  y   7,  q    2,  r  1
x  20,  y   2,  q   10,  r  0
x  20,  y   7,  q    2,  r  6
x  20,  y  10,  q    2,  r  0
x 100,  y   1,  q  100,  r  0
x 100,  y   5,  q   20,  r  0
x 100,  y  10,  q   10,  r  0

-U:---  intdiv.traces   All (21,0)
```

```
File  Edit  Options  Buffers  Tools  Python  Help

def intdiv(x, y):
    assert y != 0

    # .. compute result ..

    assert r >= 0
    assert x >= q

    return q,r

--:---  intdiv.py      All (11,0)
```

UDACITY – Software Testing course

"GCC: 9000 assertions,
LLVM: 13,000 assertions [..]
1 assertion per 110 loc"

Program Invariants

"invariants are asserted properties, such as relations among variables, at certain locations in a program"

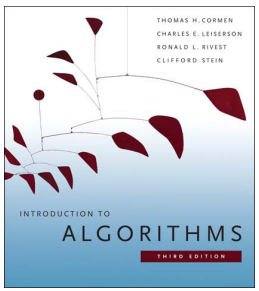

```
assert(x == 2*y);
assert(0 <= idx < |arr|);
```

Program Invariants

"invariants are asserted properties, such as relations among variables, at certain locations in a program"

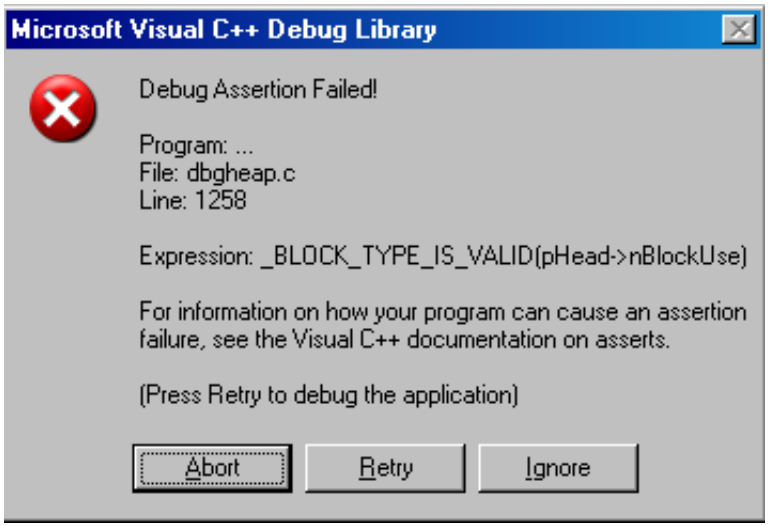

```
assert(x == 2*y);
assert(0 <= idx < |arr|);
```


```
int getDateOfMonth(int m){
    /*pre: 1 <= m <= 12*/
    ..

    /*post: 0 <= result <= 31*/
}
```



"a loop invariant is a condition that is true on entry into a loop and is guaranteed to remain true on every iteration of the loop [..]"

Uses of Invariants



- Understand and verify programs
- Formal proofs
- Debug (locate errors)
- Documentations

Approaches to Finding Invariants

Approaches to Finding Invariants

```
def intdiv(x,y):
    q = 0
    r = x
    while r ≥ y:
        a = 1
        b = y
        while r ≥ 2b:
            a = 2a
            b = 2b
        r = r - b
        q = q + a
    [L]
    return q,r
```

Static Analysis

- Analyze source code directly
- Pros: results guaranteed on any input, proofs of correctness or errors
- Cons: computationally intensive, deduce simple invariants

Approaches to Finding Invariants

```
def intdiv(x,y):
    q = 0
    r = x
    while r ≥ y:
        a = 1
        b = y
        while r ≥ 2b:
            a = 2a
            b = 2b
        r = r - b
        q = q + a
    [L]
    return q,r
```

x	y	q	r
0	1	0	0
1	1	1	0
3	4	0	3
8	1	8	0
15	5	3	0
20	2	10	0
100	1	100	0
	⋮	⋮	

Static Analysis

- Analyze source code directly
- Pros: results guaranteed on any input, proofs of correctness or errors
- Cons: computationally intensive, deduce simple invariants

Dynamic Analysis

- Analyze program traces
- Pros: fast, source code not required
- Cons: results depend on traces, might not hold for all runs

Three Challenging Classes of Invariants

Three Challenging Classes of Invariants

1. Polynomials
   - Relations over numerical variables
     $x = 3.5,\ x = 2y,\ x = qy + r,\ x^2 \geq y + z^3, |\texttt{arr}| \geq \texttt{idx} \geq 0,\ \ldots$
   - *Nonlinear* polynomials: required in scientific and engineering applications, implemented in Astrée analyzer for Airbus systems

Three Challenging Classes of Invariants

1. Polynomials
   - Relations over numerical variables
     $x = 3.5,\ x = 2y,\ x = qy + r,\ x^2 \geq y + z^3, |\texttt{arr}| \geq \texttt{idx} \geq 0,\ \ldots$
   - *Nonlinear* polynomials: required in scientific and engineering applications, implemented in Astrée analyzer for Airbus systems
2. Disjunctions
   - Represent branching behaviors in programs, e.g., search, sort
     $a \vee b\ (i = \texttt{even}) \Rightarrow (A[i] = B[i]),\ \texttt{if}\ (a = b)\ \texttt{then}\ (c = 5)\ \texttt{else}\ (c = d + 7)$
   - Many loops in OpenSSH require disjunctive invariants

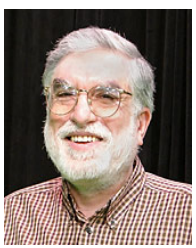
Three Challenging Classes of Invariants

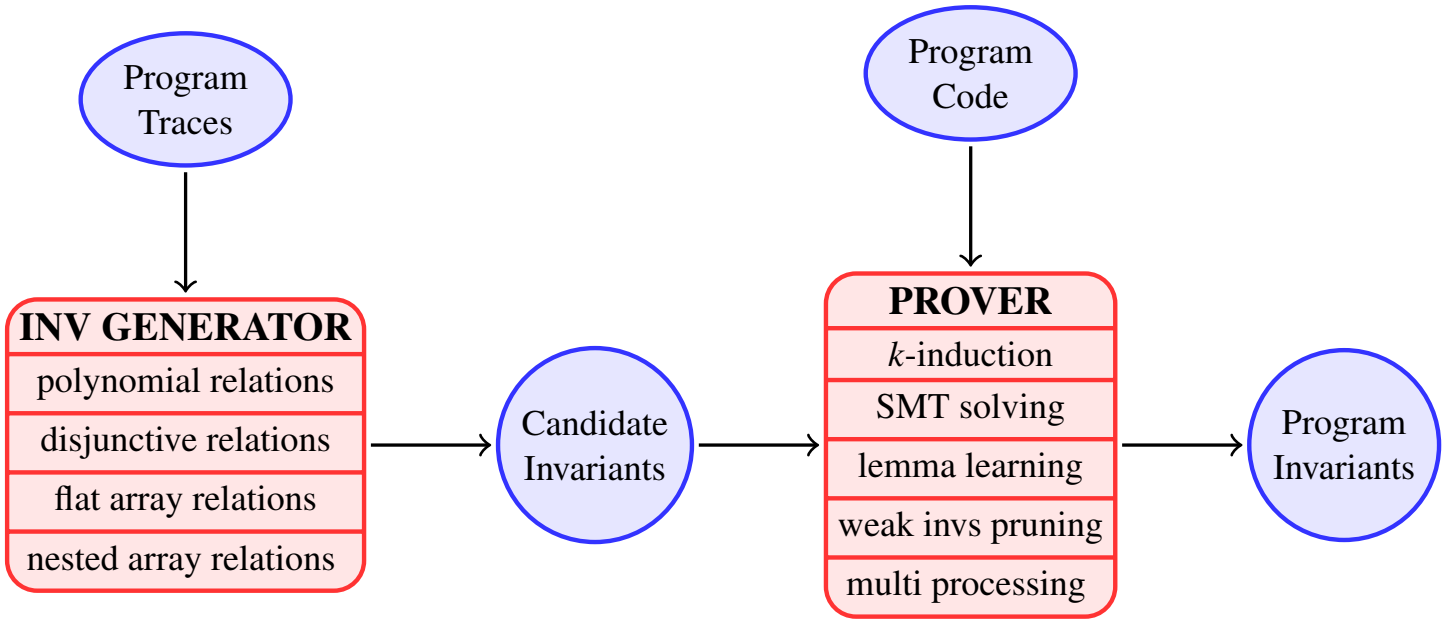
1. Polynomials
   - Relations over numerical variables
     $x = 3.5,\ x = 2y,\ x = qy + r,\ x^2 \geq y + z^3, |\texttt{arr}| \geq \texttt{idx} \geq 0,\ \ldots$
   - *Nonlinear* polynomials: required in scientific and engineering applications, implemented in Astrée analyzer for Airbus systems
2. Disjunctions
   - Represent branching behaviors in programs, e.g., search, sort
     $a \vee b\ (i = \texttt{even}) \Rightarrow (A[i] = B[i]),\ \texttt{if}\ (a = b)\ \texttt{then}\ (c = 5)\ \texttt{else}\ (c = d + 7)$
   - Many loops in OpenSSH require disjunctive invariants
3. Arrays
   - Relations among (multi-dimensional) array variables
     $A[i] = B[i],\ A[i][j] = B[i] + 3C[i] - 1.2,\ A[i] = B[C[i]][D[2i]],\ r = f(g(x), h(y, z))$
   - Popular data structure to implement strings, vectors, matrices, memory, ..

Three Challenging Classes of Invariants

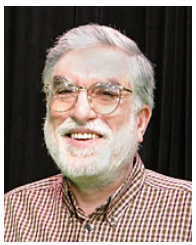
1. Polynomials
   - Relations over numerical variables
     $x = 3.5,\ x = 2y,\ x = qy + r,\ x^2 \geq y + z^3, |\texttt{arr}| \geq \texttt{idx} \geq 0,\ \ldots$
   - *Nonlinear* polynomials: required in scientific and engineering applications, implemented in Astrée analyzer for Airbus systems
2. Disjunctions
   - Represent branching behaviors in programs, e.g., search, sort
     $a \vee b\ (i = \texttt{even}) \Rightarrow (A[i] = B[i]),\ \texttt{if}\ (a = b)\ \texttt{then}\ (c = 5)\ \texttt{else}\ (c = d + 7)$
   - Many loops in OpenSSH require disjunctive invariants
3. Arrays
   - Relations among (multi-dimensional) array variables
     $A[i] = B[i],\ A[i][j] = B[i] + 3C[i] - 1.2,\ A[i] = B[C[i]][D[2i]],\ r = f(g(x), h(y,z))$
   - Popular data structure to implement strings, vectors, matrices, memory, ..

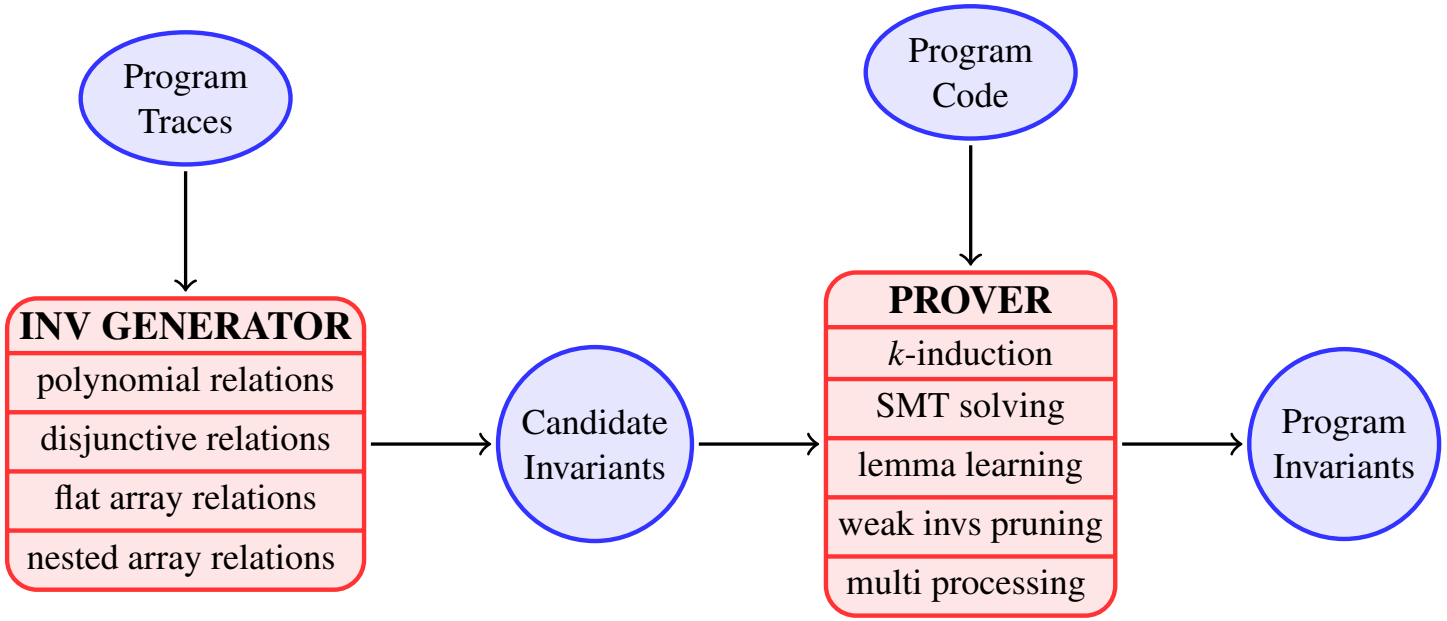
DIG: Dynamic Invariant Generation (TOSEM '14, ICSE '14, ICSE '12)





DIG: Dynamic Invariant Generation (TOSEM '14, ICSE '14, ICSE '12)





Goal: developing efficient methods to capture precise and provably correct program invariants

- Efficient: reformulate and solve using techniques such as equation solving and polyhedral construction
- Precise: employ expressive templates and infer invariants *directly* from traces
- Sound: integrate theorem proving to formally verify results

Outline

Example: Cohen Integer Division

```
def intdiv(x, y):
    q = 0
    r = x
    while r ≥ y:
        a = 1
        b = y
        while r ≥ 2b:
            [L]
            a = 2a
            b = 2b
        r = r - b
        q = q + a
    return q,r
```

Example: Cohen Integer Division

```
def intdiv(x, y):
    q = 0
    r = x
    while r ≥ y:
        a = 1
        b = y
        while r ≥ 2b:
            [L]
            a = 2a
            b = 2b
        r = r - b
        q = q + a
    return q,r
```

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

Invariants at L: $b = ya,\ x = qy + r,\ r \geq 2ya$

Polynomial Relations (ICSE '12)

DIG discovers polynomial relations of the forms

Equalities $c_0 + c_1x_1 + c_2x_n + c_3x_1x_2 + \cdots + c_mx_1^{d_1} \ldots x_n^{d_n} = 0$

Inequalities $c_0 + c_1x_1 + c_2x_n + c_3x_1x_2 + \cdots + c_mx_1^{d_1} \ldots x_n^{d_n} \geq 0, \quad c_i \in \mathbb{R}$

Examples

cubic $z - 6n = 6,\ \frac{1}{12}z^2 - y - \frac{1}{2}z = -1$

extended gcd $\gcd(a, b) = ia + jb$

sqrt $x + \varepsilon \geq y^2 \geq x - \varepsilon$

Polynomial Relations (ICSE '12)

DIG discovers polynomial relations of the forms

Equalities $c_0 + c_1x_1 + c_2x_n + c_3x_1x_2 + \cdots + c_mx_1^{d_1} \ldots x_n^{d_n} = 0$

Inequalities $c_0 + c_1x_1 + c_2x_n + c_3x_1x_2 + \cdots + c_mx_1^{d_1} \ldots x_n^{d_n} \geq 0, \quad c_i \in \mathbb{R}$

Examples

cubic $z - 6n = 6,\ \frac{1}{12}z^2 - y - \frac{1}{2}z = -1$

extended gcd $\gcd(a,b) = ia + jb$

sqrt $x + \varepsilon \geq y^2 \geq x - \varepsilon$

Method

- Equalities: solve equations
- Inequalities: construct polyhedra

Example: Cohen Integer Division

```
def intdiv(x, y):
    q = 0
    r = x
    while r ≥ y:
        a = 1
        b = y
        while r ≥ 2b:
            [L]
            a = 2a
            b = 2b
        r = r - b
        q = q + a
    return q,r
```

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

Invariants at L: $b = ya,\ x = qy + r,\ r \geq 2ya$

Finding 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

Finding 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

Finding Nonlinear Equations using Equation Solver

- Terms and degrees

$$V = \{r, y, a\};\ \texttt{deg} = 2$$

$$\downarrow$$

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

$$T = \{\ldots, \log(r), a^y, \sin(y), \ldots\}$$

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

Finding 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

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

Finding 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

$$c_1+c_2r+c_3y+c_4a+c_5ry+c_6ra+c_7ya+c_8r^2+c_9y^2+c_{10}a^2 = 0$$

- System of *linear* equations

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

Finding 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

$$c_1+c_2r+c_3y+c_4a+c_5ry+c_6ra+c_7ya+c_8r^2+c_9y^2+c_{10}a^2 = 0$$

- System of *linear* equations

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

- Solve for coefficients c_i

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

Geometric Invariant Inference (TOSEM '14)

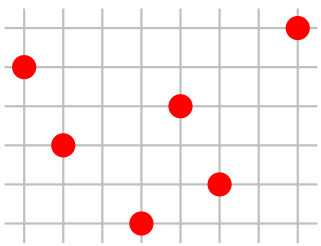
- Treat trace values as points in multi-dimensional space
- Build a convex hull (polyhedron) over the points
- Representation of a polyhedron: a conjunction of inequalities

Geometric Invariant Inference (TOSEM '14)

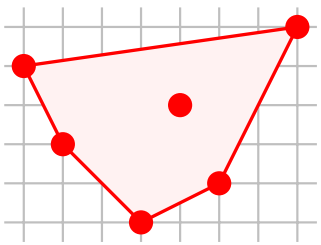
- Treat trace values as points in multi-dimensional space
- Build a convex hull (polyhedron) over the points
- Representation of a polyhedron: a conjunction of inequalities

x	y
-2	1
-1	-1
1	-3
2	0
3	-2
5	2

program traces



trace pts in 2D



polygon

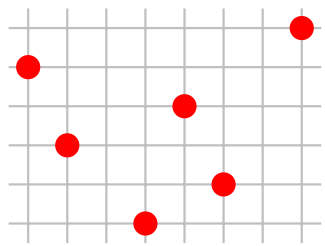
$c_1x + c_2y \geq c$

Geometric Invariant Inference (TOSEM '14)

- Treat trace values as points in multi-dimensional space
- Build a convex hull (polyhedron) over the points
- Representation of a polyhedron: a conjunction of inequalities

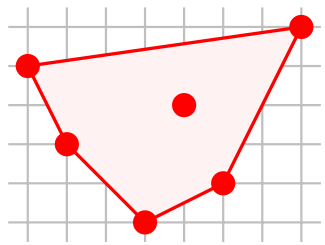
x	y
-2	1
-1	-1
1	-3
2	0
3	-2
5	2

program traces



trace pts

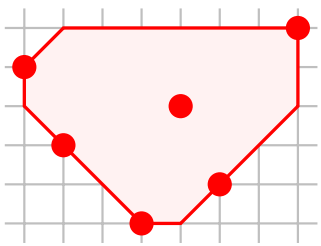
in 2D



polygon

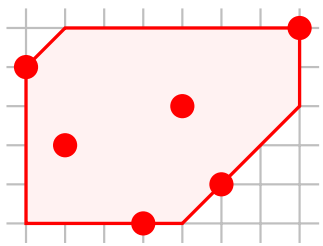
$c_1x + c_2y \geq c$

- Support simpler shapes (decreasing precision, increasing efficiency)



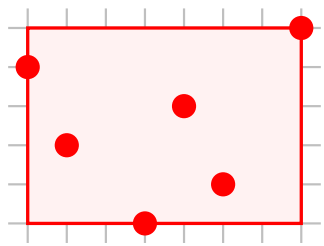
octagon

$\pm x \pm y \geq c$



zone

$x - y \geq c$



box

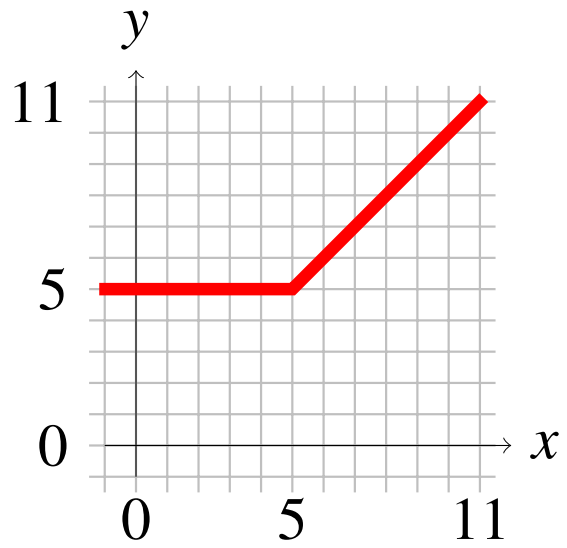
$\pm x, y \geq c$

Outline

Example: Disjunctive Invariants

```
def ex(x):
  y = 5
  if x > y: x = y
  while[L] x ≤ 10:
    if x ≥ 5:
      y = y + 1
    x = x + 1

  assert y ≡ 11
```

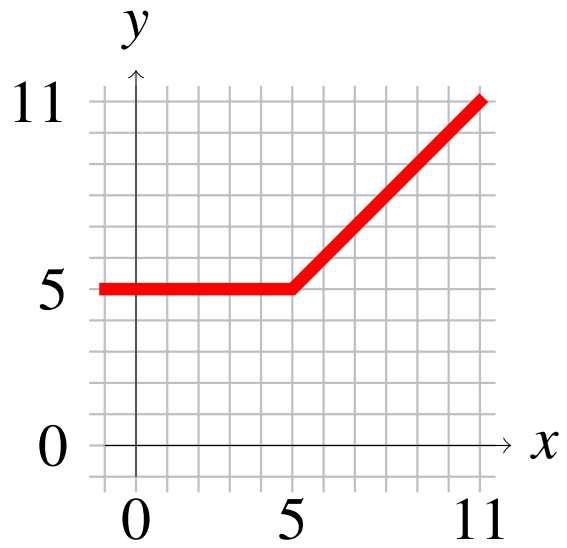
x	y
-1	5
⋮	⋮
5	5
6	6
⋮	⋮
11	11



Example: Disjunctive Invariants

```
def ex(x):
  y = 5
  if x > y: x = y
  while[L] x ≤ 10:
    if x ≥ 5:
      y = y + 1
    x = x + 1

  assert y ≡ 11
```

x	y
-1	5
⋮	⋮
5	5
6	6
⋮	⋮
11	11

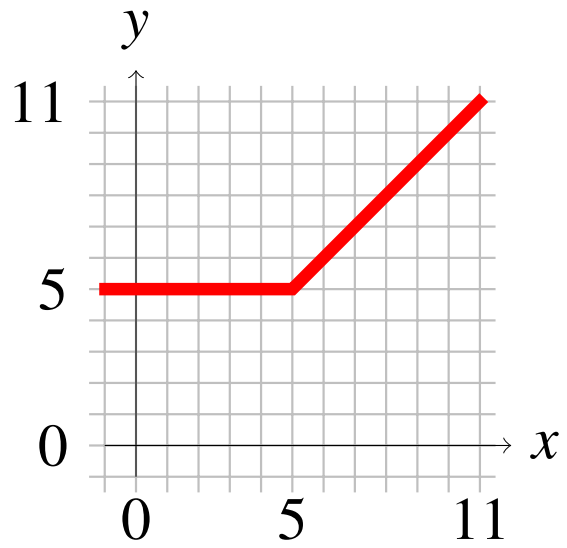


$$L : (x < 5 \wedge 5 = y) \ \vee \ (x \geq 5 \wedge x = y), 11 \geq x$$

Example: Disjunctive Invariants

```
def ex(x):
  y = 5
  if x > y: x = y
  while[L] x ≤ 10:
    if x ≥ 5:
      y = y + 1
    x = x + 1

  assert y ≡ 11
```

x	y
-1	5
⋮	⋮
5	5
6	6
⋮	⋮
11	11



$$L : (x < 5 \wedge 5 = y) \ \vee \ (x \geq 5 \wedge x = y), 11 \geq x$$

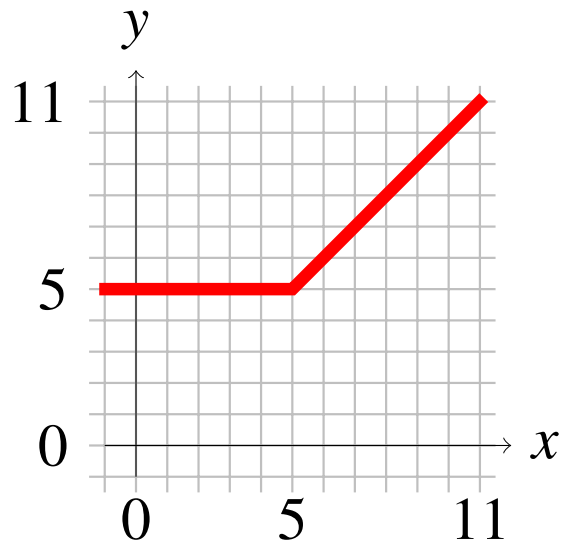
Disjunction of 2 cases:

1. if $x < 5$ then $y = 5$
2. if $x \geq 5$ then $x = y$

Example: Disjunctive Invariants

```
def ex(x):
  y = 5
  if x > y: x = y
  while[L] x ≤ 10:
    if x ≥ 5:
      y = y + 1
    x = x + 1

  assert y ≡ 11
```

x	y
-1	5
⋮	⋮
5	5
6	6
⋮	⋮
11	11



$$L : (x < 5 \wedge 5 = y) \ \vee \ (x \geq 5 \wedge x = y), 11 \geq x$$

Disjunction of 2 cases:

1. if $x < 5$ then $y = 5$
2. if $x \geq 5$ then $x = y$

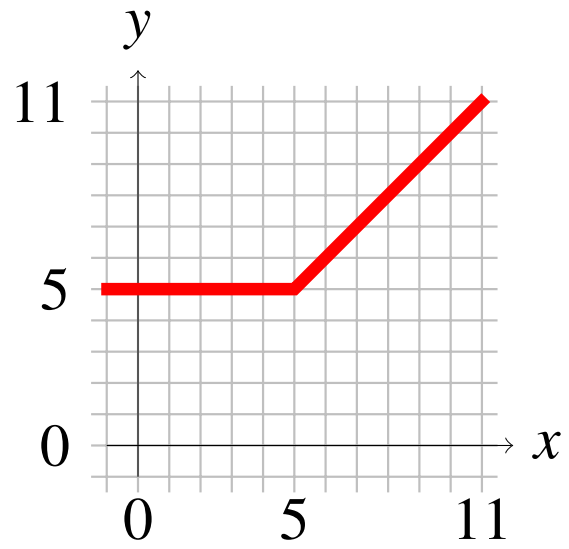
$\longleftrightarrow$ if $0 > x - 5$ then $0 = y - 5$ else $x - 5 = y - 5$

$\max(0, x - 5) = y - 5$

Example: Disjunctive Invariants

```
def ex(x):
  y = 5
  if x > y: x = y
  while[L] x ≤ 10:
    if x ≥ 5:
      y = y + 1
    x = x + 1

  assert y ≡ 11
```

x	y
-1	5
⋮	⋮
5	5
6	6
⋮	⋮
11	11



$$L : (x < 5 \wedge 5 = y) \ \vee \ (x \geq 5 \wedge x = y), 11 \geq x$$

Disjunction of 2 cases:

1. if $x < 5$ then $y = 5$
2. if $x \geq 5$ then $x = y$

$\longleftrightarrow$ if $0 > x - 5$ then $0 = y - 5$ else $x - 5 = y - 5$

$\max(0, x - 5) = y - 5$

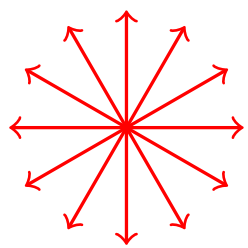
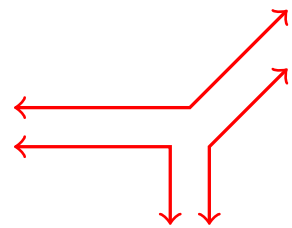
a linear relation .. in max-plus algebra

Max-plus Algebra

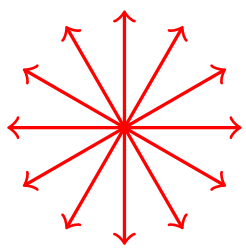
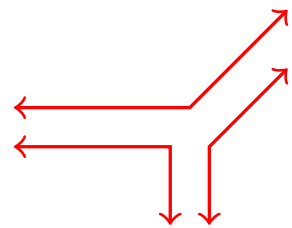
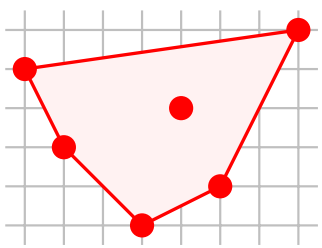
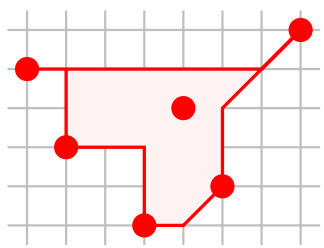
Max-plus Algebra

	Linear	**Max-plus**
Domain	$\mathbb{R}$	$\mathbb{R} \cup \{-\infty\}$
Addition	$+$	max
Multiplication	$\times$	$+$
Zero elem	0	$-\infty$
Unit elem	1	0
Relation form	$c_0 + c_1 t_1 + \cdots + c_n t_n \geq 0$	$\max(c_0, c_1 + t_1, \ldots, c_n + t_n) \geq \max(d_0, d_1 + t_1, \ldots, d_n + t_n)$

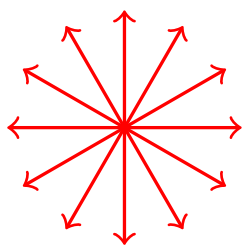
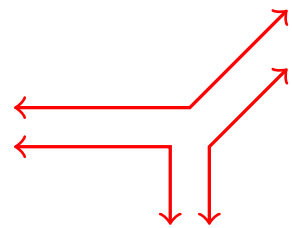
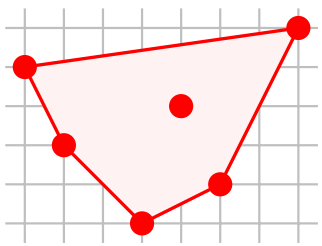
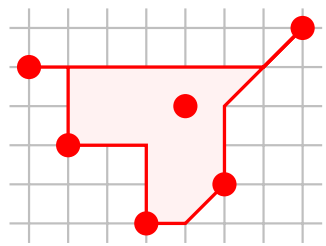
Max-plus Algebra

	Linear	**Max-plus**
Domain	$\mathbb{R}$	$\mathbb{R} \cup \{-\infty\}$
Addition	$+$	max
Multiplication	$\times$	$+$
Zero elem	0	$-\infty$
Unit elem	1	0
Relation form	$c_0 + c_1 t_1 + \cdots + c_n t_n \geq 0$	$\max(c_0, c_1 + t_1, \ldots, c_n + t_n) \geq \max(d_0, d_1 + t_1, \ldots, d_n + t_n)$
Line shapes		

Max-plus Algebra

	Linear	**Max-plus**
Domain	$\mathbb{R}$	$\mathbb{R} \cup \{-\infty\}$
Addition	$+$	max
Multiplication	$\times$	$+$
Zero elem	0	$-\infty$
Unit elem	1	0
Relation form	$c_0 + c_1 t_1 + \cdots + c_n t_n \geq 0$	$\max(c_0, c_1 + t_1, \ldots, c_n + t_n) \geq \max(d_0, d_1 + t_1, \ldots, d_n + t_n)$
Line shapes		
Convex hull		

Max-plus Algebra

	Linear	**Max-plus**
Domain	$\mathbb{R}$	$\mathbb{R} \cup \{-\infty\}$
Addition	$+$	max
Multiplication	$\times$	$+$
Zero elem	0	$-\infty$
Unit elem	1	0
Relation form	$c_0 + c_1 t_1 + \cdots + c_n t_n \geq 0$	$\max(c_0, c_1 + t_1, \ldots, c_n + t_n) \geq \max(d_0, d_1 + t_1, \ldots, d_n + t_n)$
Line shapes		
Convex hull		

Not convex in classical sense!

Disjunctive Invariants (ICSE '14)

DIG discovers disjunctive relations of the max-plus form

$$\max(c_0, c_1 + t_1, \ldots, c_n + t_n) \geq \max(d_0, d_1 + t_1, \ldots, d_n + t_n)$$

Examples

$$\begin{array}{lcccc} z = \max(x, y) & \equiv & (x < y \wedge z = x) & \vee & (x \geq y \wedge z = y) \\ \text{strncpy}(s, d, n) & \equiv & (n \geq |s| \wedge |d| = |s|) & \vee & (n < |s| \wedge |d| \geq n) \end{array}$$

Disjunctive Invariants (ICSE '14)

DIG discovers disjunctive relations of the max-plus form

$$\max(c_0, c_1 + t_1, \ldots, c_n + t_n) \geq \max(d_0, d_1 + t_1, \ldots, d_n + t_n)$$

Examples

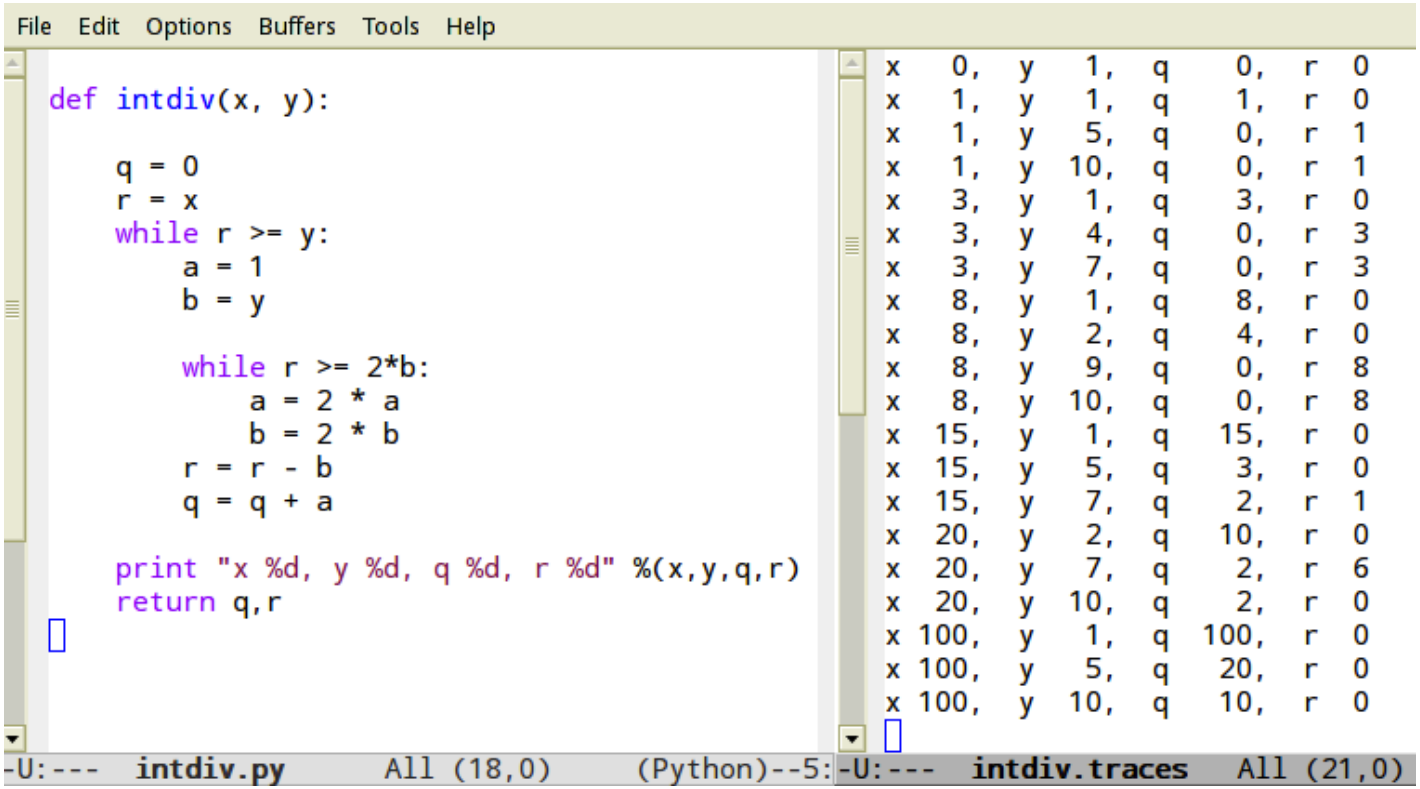
$$\begin{array}{lcccc} z = \max(x, y) & \equiv & (x < y \wedge z = x) & \vee & (x \geq y \wedge z = y) \\ \text{strncpy}(s, d, n) & \equiv & (n \geq |s| \wedge |d| = |s|) & \vee & (n < |s| \wedge |d| \geq n) \end{array}$$

Method

- Use terms to express variables
- Build a max-plus convex polyhedron and extract facets
- Introduce simpler max-plus shapes for lower computational complexity

Outline

Spurious Invariants

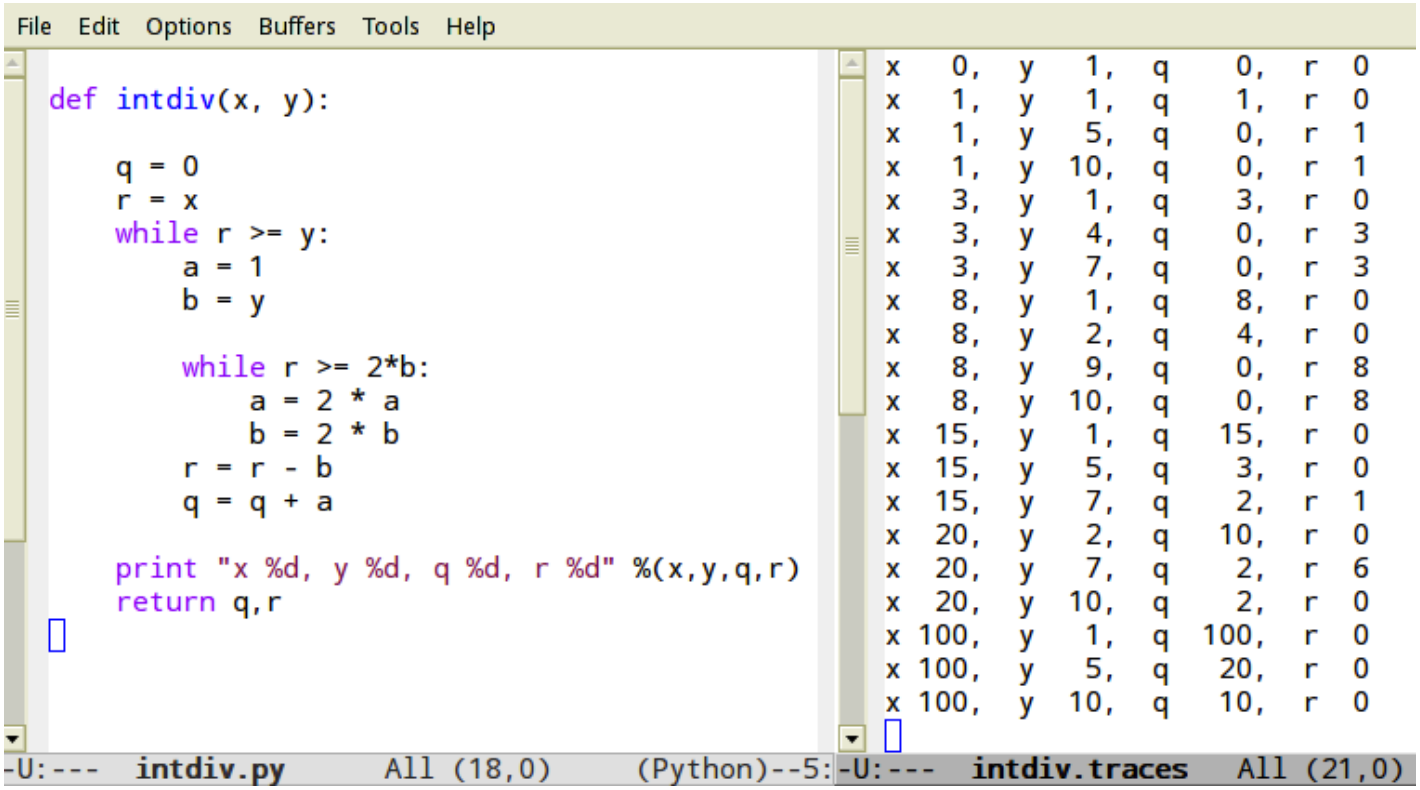


Valid results

- x, y, q, r are integers
- $r \geq 0$
- $x = q * y + r$

$\vdots$

Spurious Invariants



Valid results

- x, y, q, r are integers
- $r \geq 0$
- $x = q * y + r$

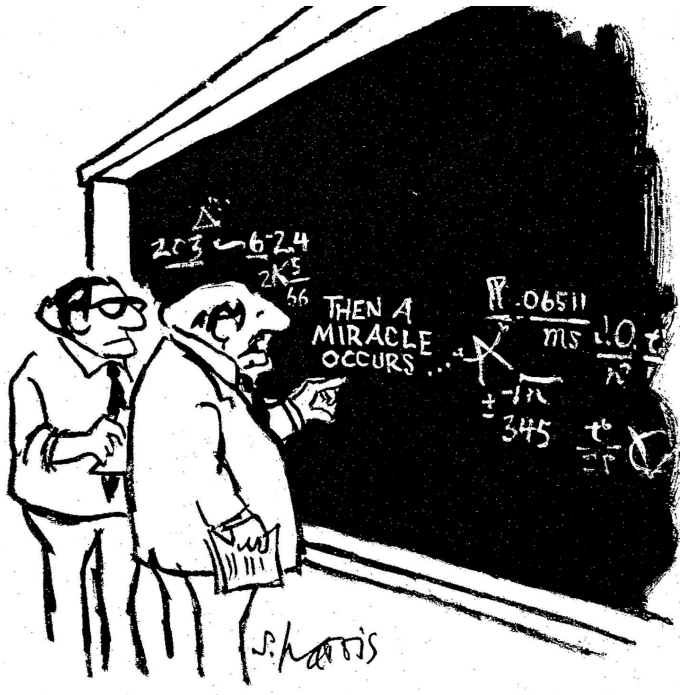
$\vdots$

Spurious results

- $100 \geq x \geq 0$
- $10 \geq y \geq 1$
- $100 \geq q - r \geq -8$

$\vdots$

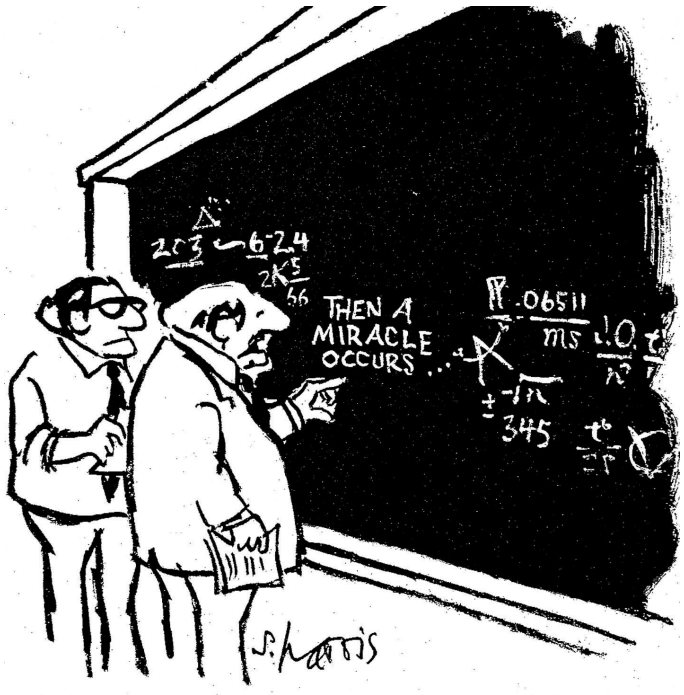
KIP: k-Induction Prover (ICSE '14)



KIP, an automatic theorem prover, for verifying invariants

1. Implement *k*-induction
2. Employ powerful constraint solving
3. Learn new lemmas
4. Use multi-processing
5. Identify strongest invariants

KIP: k-Induction Prover (ICSE '14)



KIP, an automatic theorem prover, for verifying invariants

1. Implement *k*-induction
2. Employ powerful constraint solving
3. Learn new lemmas
4. Use multi-processing
5. Identify strongest invariants

DIG + KIP ≡ hybridization of dynamic and static analysis

Idea: generating (or solving) is typically harder than checking

- Dynamically infer invariants from program traces
- Statically prove candidate results using program source

An efficient and sound technique to generate complex program invariants

Experimental Results

Benchmarks

- Nonlinear test suite: 27 programs require nonlinear invariants
- Disjunctive testsuite: 14 programs require disjunctive invariants

Setup

- Implemented in SAGE/Python (with Z3 backend solver)
- Invariants obtained at loop entrances and program exits

Experimental Results

Benchmarks

- Nonlinear test suite: 27 programs require nonlinear invariants
- Disjunctive testsuite: 14 programs require disjunctive invariants

Setup

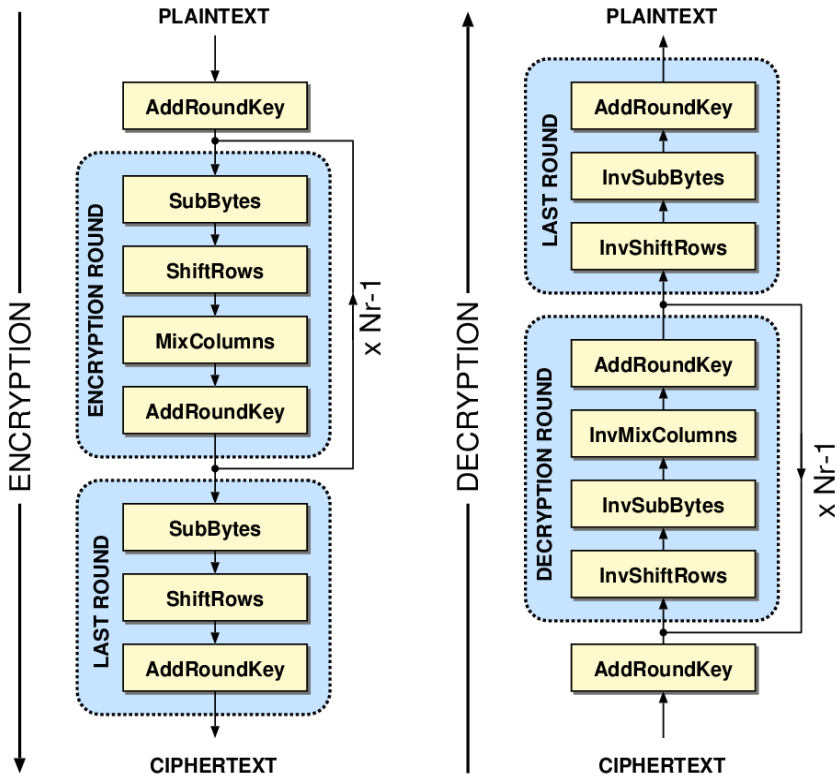
- Implemented in SAGE/Python (with Z3 backend solver)
- Invariants obtained at loop entrances and program exits

Results

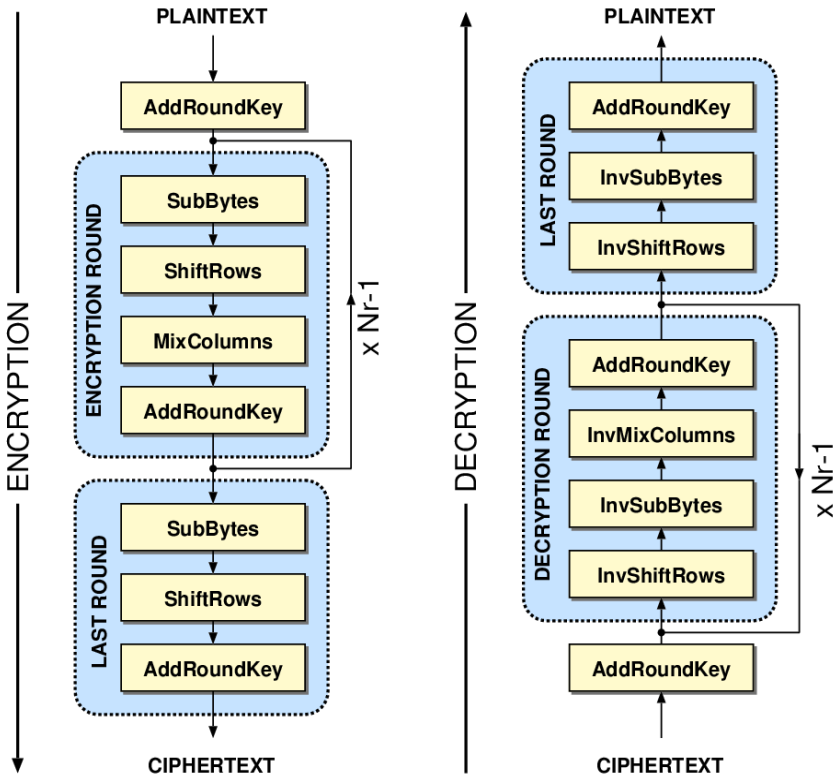
- All generated equalities are valid and most inequalities are spurious (and removed by KIP)
- Invariants generated are sufficiently strong to explain program behavior
  - Proved correctness of 36/41 programs, 2 mins per program
  - No spurious results
- Current dynamic analysis cannot find any of these invariants

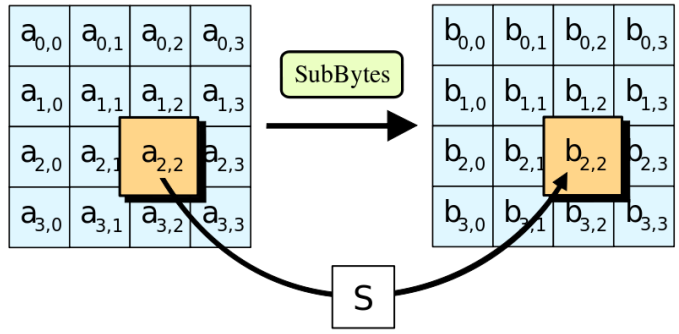
Outline

A Case Study: Advanced Encryption Standard (AES)



A Case Study: Advanced Encryption Standard (AES)






```
def SubBytes(S,a):
  #S is 1D array, a is 2D array
  b =
    [[S[a[0][0]],S[a[0][1]],
      S[a[0][2]],S[a[0][3]]],
     [S[a[1][0]],S[a[1][1]],
      S[a[1][2]],S[a[1][3]]],
     [S[a[2][0]],S[a[2][1]],
      S[a[2][2]],S[a[2][3]]],
     [S[a[3][0]],S[a[3][1]],
      S[a[3][2]],S[a[3][3]]]]

  [L]
  return b
```

[L]: b[i][j] = S[a[i][j]]

Nested Array Problem (TOSEM '14, ICSE '12)

The Array Nesting (AN) problem

Given an n-dimensional array a and set B of single dimensional arrays, does there exist a *nesting* $(b_1, \ldots, b_l)$ from B such that

$$a[i_1] \ldots [i_n] = b_1[\ldots[b_l[c_0 + c_1 i_1 + \cdots + c_n i_n]] \ldots] \; ?$$

Example: $a[i] = b_1[i + 3j + 5]$, $a[i][j][k] = b_1[b_2[i + 2j + 3k]]$

Nested Array Problem (TOSEM '14, ICSE '12)

The Array Nesting (AN) problem

Given an n-dimensional array a and set B of single dimensional arrays, does there exist a *nesting* $(b_1, \ldots, b_l)$ from B such that

$$a[i_1] \ldots [i_n] = b_1[\ldots[b_l[c_0 + c_1 i_1 + \cdots + c_n i_n]]\ldots] \; ?$$

Example: $a[i] = b_1[i + 3j + 5]$, $a[i][j][k] = b_1[b_2[i + 2j + 3k]]$

Complexity (**m**: number of array variables, **n**: size of the largest array variable)

- AN is strongly NP-Complete in **m** (reduction from *Exact Covering*)
- AN can be solved in polynomial time in **n** using reachability analysis
- Same complexity for the generalized version with multi-dimensional and repeating arrays, e.g., $\mathsf{a}[i][j] = 2\mathsf{b}_1[\mathsf{b}_2[2i + 3]][[\mathsf{b}_3[i][b_2[j]]]]$

Experimental Results

Nested Array Relations (functions are treated as a special type of arrays)

xor2Word	$R[i] = \mathsf{xor}(A[i], B[i]),\ A[i] = \mathsf{xor}(R[i], B[i]), B[i] = \mathsf{xor}(R[i], A[i])$
addRoundKey	$R[i][j] = \mathsf{xor}(T[i][j], H[i][j])$
multWord	$R[i] = T[\mathsf{mod}(L[A[i]] + L[B[i]], 255)]$

Experimental Results

Nested Array Relations (functions are treated as a special type of arrays)

xor2Word	$R[i] = \mathsf{xor}(A[i], B[i]),\ A[i] = \mathsf{xor}(R[i], B[i]), B[i] = \mathsf{xor}(R[i], A[i])$
addRoundKey	$R[i][j] = \mathsf{xor}(T[i][j], H[i][j])$
multWord	$R[i] = T[\mathsf{mod}(L[A[i]] + L[B[i]], 255)]$

Flat Array Relations (flattening array elements and solving equations)

block2State	$R[i][j] = T[4i + j]$
RotWord	$R = [W[1], W[2], W[3], W[0]]$
keySetupEnc8	$R[i][j] = \mathsf{cipherKey}[4i + j]$ for $i = 0, \ldots, 7;\ j = 0, \ldots, 3$

Experimental Results

Nested Array Relations (functions are treated as a special type of arrays)

xor2Word	$R[i] = \mathsf{xor}(A[i], B[i]),\ A[i] = \mathsf{xor}(R[i], B[i]), B[i] = \mathsf{xor}(R[i], A[i])$
addRoundKey	$R[i][j] = \mathsf{xor}(T[i][j], H[i][j])$
multWord	$R[i] = T[\mathsf{mod}(L[A[i]] + L[B[i]], 255)]$

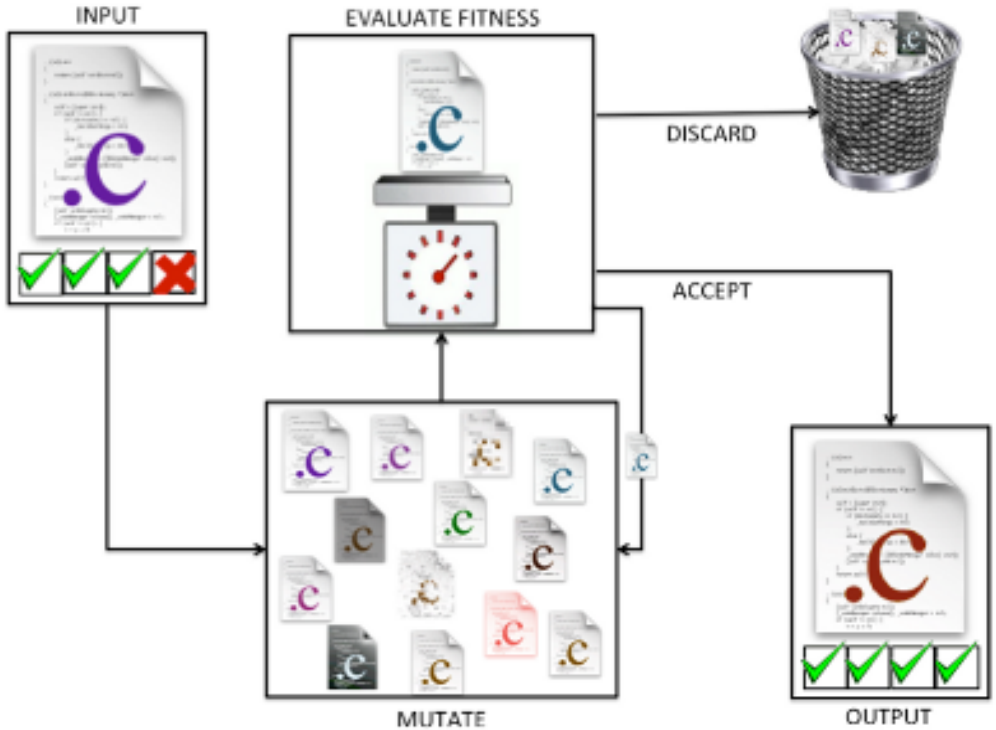
Flat Array Relations (flattening array elements and solving equations)

block2State	$R[i][j] = T[4i + j]$
RotWord	$R = [W[1], W[2], W[3], W[0]]$
keySetupEnc8	$R[i][j] = \mathsf{cipherKey}[4i + j]$ for $i = 0, \ldots, 7;\ j = 0, \ldots, 3$

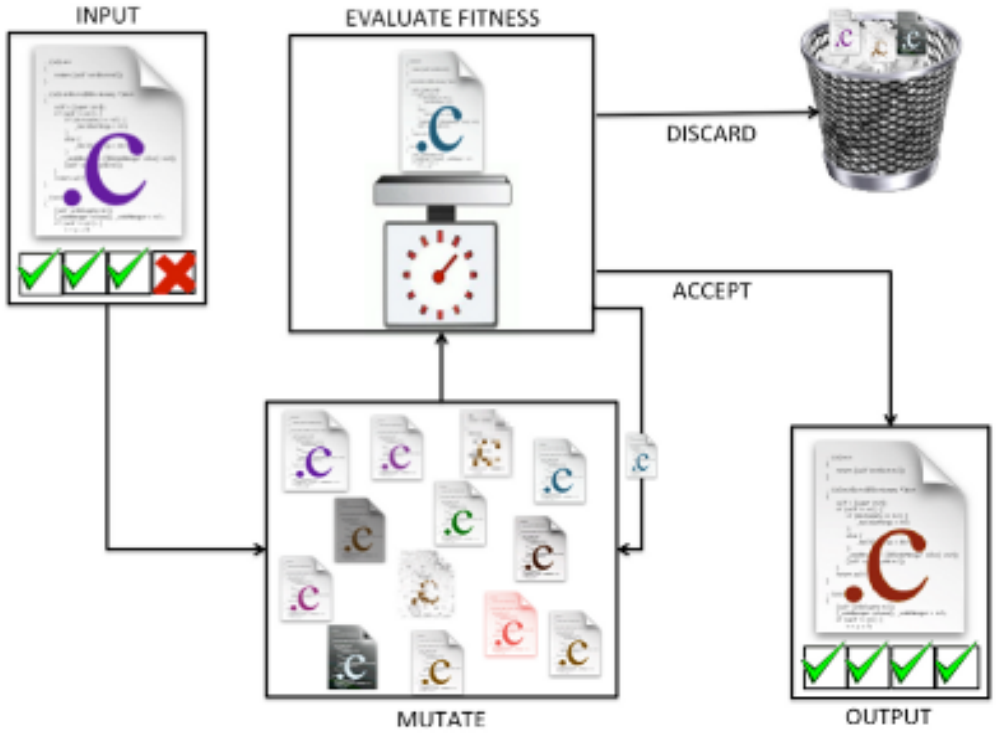
DIG found 60% of the documented array relations in AES under 15 minutes

Outline

GenProg (TSE '12, CACM '10, ICSE '09, GECCO '09)

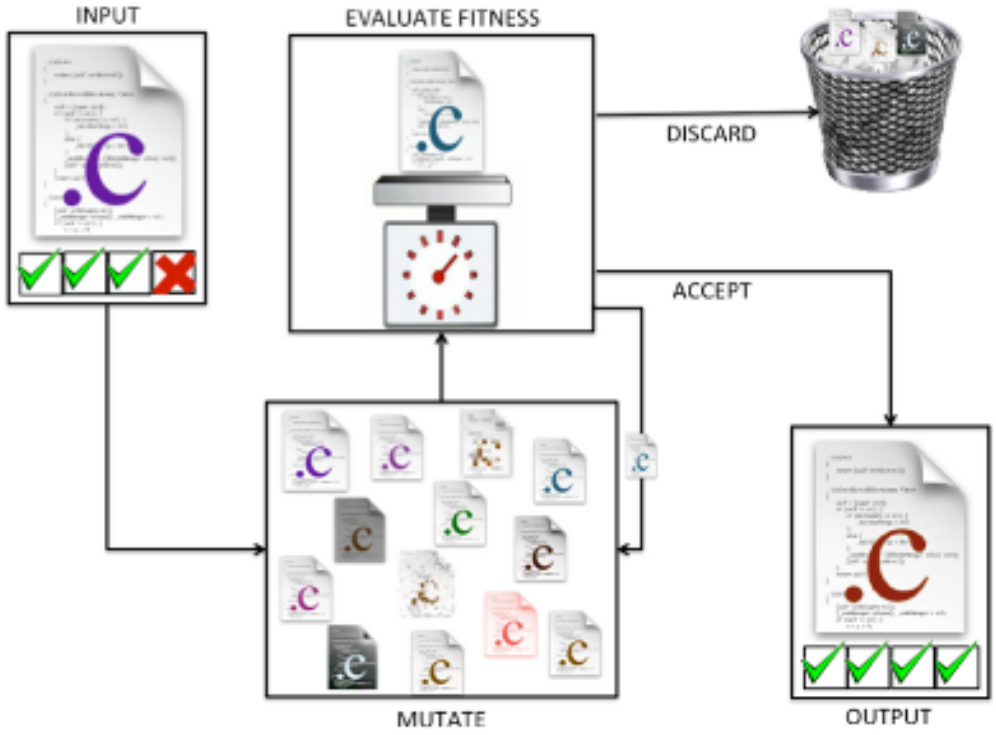


GenProg (TSE '12, CACM '10, ICSE '09, GECCO '09)



1. Isolate faults
2. Mutate program statements and reuse existing code
3. Check repair candidates

GenProg (TSE '12, CACM '10, ICSE '09, GECCO '09)



1. Isolate faults
2. Mutate program statements and reuse existing code
3. Check repair candidates

Results: demonstrated on bugs in real-world software (repair 16 programs over 1.25 MLocs, 2 mins avg)

Using Reachability to Synthesize Repairs

Using Reachability to Synthesize Repairs

Verification as a Reachability problem

- Show a program state violating a given specification is not reachable
- Test-input generation: finds inputs that reach a program location

```
def P(x, y):
 if 2 × x == y:
  if x > y + 10:
     [L] #reachable, e.g., x = -20, y = -40

  return 0
```

Using Reachability to Synthesize Repairs

Verification as a Reachability problem

- Show a program state violating a given specification is not reachable
- Test-input generation: finds inputs that reach a program location

```
def P(x, y):
 if 2 × x == y:
  if x > y + 10:
     [L] #reachable, e.g., x = -20, y = -40

  return 0
```

Template-based Synthesis

- A practical form of synthesis that creates code under specific templates

```
def Q(i, u, d):
  if i:
    b = c0 + c1 × u + c2 × d # linear exp template
  else:
    b = u
  if (b > d): r = 1
  else: r = 0
  return r
```

Test suite

$$
\begin{array}{lcl}
Q(1, 0, 100) & = & 0 \\
Q(1, 11, 110) & = & 1 \\
Q(0, 100, 50) & = & 1 \\
Q(1, -20, 60) & = & 1 \\
Q(0, 0, 10) & = & 0 \\
Q(0, 0, -10) & = & 1
\end{array}
$$

- Applicable to automatic program repair: identify suspicious program statements and synthesize repairs for those statements

Using Reachability to Synthesize Repairs

Verification as a Reachability problem

- Show a program state violating a given specification is not reachable
- Test-input generation: finds inputs that reach a program location

```
def P(x, y):
 if 2 × x == y:
  if x > y + 10:
     [L] #reachable, e.g., x = -20, y = -40

  return 0
```

Template-based Synthesis

- A practical form of synthesis that creates code under specific templates

```
def Q(i, u, d):
  if i:
    b = c0 + c1 × u + c2 × d # linear exp template
  else:
    b = u
  if (b > d): r = 1
  else: r = 0
  return r
```

Test suite

$$
\begin{array}{lcl}
Q(1, 0, 100) & = & 0 \\
Q(1, 11, 110) & = & 1 \\
Q(0, 100, 50) & = & 1 \\
Q(1, -20, 60) & = & 1 \\
Q(0, 0, 10) & = & 0 \\
Q(0, 0, -10) & = & 1
\end{array}
$$

- Applicable to automatic program repair: identify suspicious program statements and synthesize repairs for those statements

Goal: connecting verification to synthesis to leverage existing verification techniques and tools to synthesize programs

From Reachability to Synthesis

Theorem: Template-based Synthesis is reducible to Reachability

Given a general instance of synthesis, create a specific instance of reachability consisting of a special location reachable iff synthesis has a solution

From Reachability to Synthesis

Theorem: Template-based Synthesis is reducible to Reachability

Given a general instance of synthesis, create a specific instance of reachability consisting of a special location reachable iff synthesis has a solution

```
def Q(i, u, d):
  if i:
    b = c0 + c1 × u + c2 × d #syn template
  else: b = u
  if (b > d): r = 1
  else: r = 0
  return r
```

Test suite

$$
\begin{array}{lcl}
Q(1, 0, 100) & = & 0 \\
Q(1, 11, 110) & = & 1 \\
Q(0, 100, 50) & = & 1 \\
Q(1, -20, 60) & = & 1 \\
Q(0, 0, 10) & = & 0 \\
Q(0, 0, -10) & = & 1
\end{array}
$$

Input: a synthesis instance

From Reachability to Synthesis

Theorem: Template-based Synthesis is reducible to Reachability

Given a general instance of synthesis, create a specific instance of reachability consisting of a special location reachable iff synthesis has a solution

```
def Q(i, u, d):
  if i:
    b = c0 + c1 × u + c2 × d #syn template
  else: b = u
  if (b > d): r = 1
  else: r = 0
  return r
```

Test suite

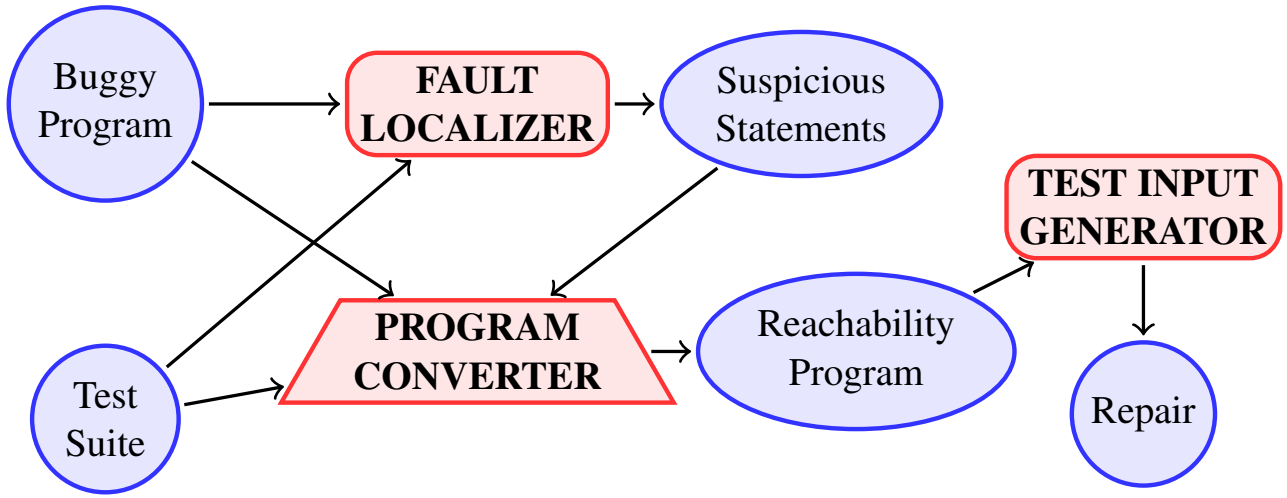
$$
\begin{array}{lcl}
Q(1, 0, 100) & = & 0 \\
Q(1, 11, 110) & = & 1 \\
Q(0, 100, 50) & = & 1 \\
Q(1, -20, 60) & = & 1 \\
Q(0, 0, 10) & = & 0 \\
Q(0, 0, -10) & = & 1
\end{array}
$$

Input: a synthesis instance

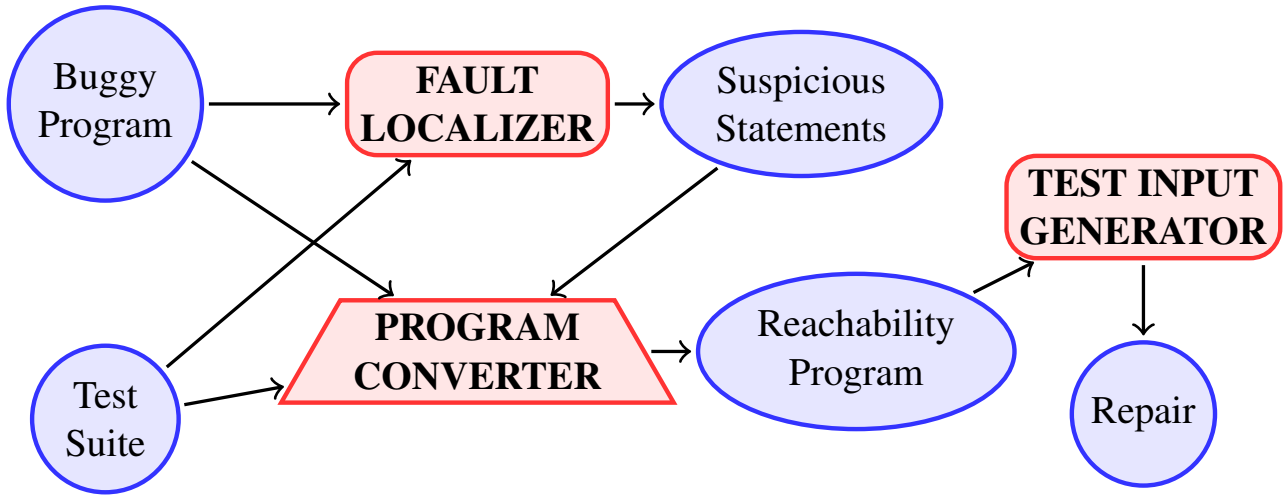
```
def pQ(i, u, d, c0, c1, c2):
  if i:
    b = c0 + c1 × u + c2 × d
  else: b = u
  if b > d: r = 1
  else: r = 0
  return r

def pmain(c0, c1, c2):
   e = pQ(1, 0,100, c0,c1,c2) == 0 and
       pQ(1,11,110, c0,c1,c2) == 1 and
       pQ(0,100,50, c0,c1,c2) == 1 and
       pQ(1,-20,60, c0,c1,c2) == 1 and
       pQ(0, 0, 10, c0,c1,c2) == 0 and
       pQ(0, 0,-10, c0,c1,c2) == 1

   if e:
      [L]  #pass the given test suite
   return 0
```

Output: a reachability instance, solvable using a test-input generation tool

CETI: Correcting Errors using Test Inputs (in submission)

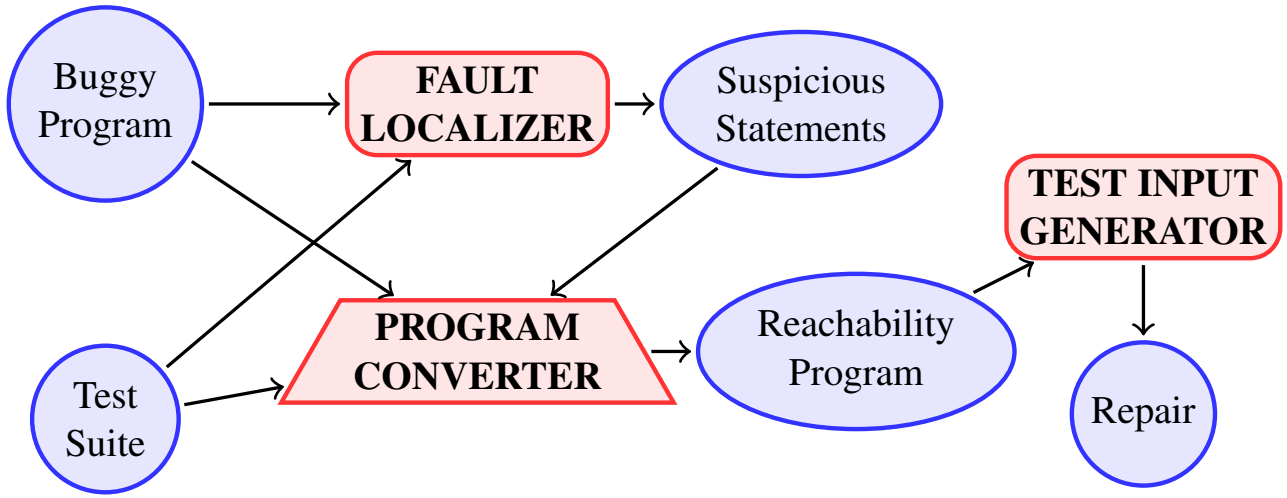


CETI: Correcting Errors using Test Inputs (in submission)



- **CETI**: automatic program repair using test-input generation
  1. Obtain suspicious statements using an existing fault localization tool
  2. Apply synthesis templates to create template-based synthesis instances
  3. Convert to reachabilty programs using reduction theorem
  4. Employ an off-the-shelf test-input generator to solve reachability, i.e., creating repairs

CETI: Correcting Errors using Test Inputs (in submission)



- **CETI**: automatic program repair using test-input generation
  1. Obtain suspicious statements using an existing fault localization tool
  2. Apply synthesis templates to create template-based synthesis instances
  3. Convert to reachabilty programs using reduction theorem
  4. Employ an off-the-shelf test-input generator to solve reachability, i.e., creating repairs
- Can fix errors that other repair techniques cannot

Outline

Analyzing Configurable Software (in submission)

- Modern software is highly configurable
- Different configurations can lead to different program behaviors

Analyzing Configurable Software (in submission)

- Modern software is highly configurable
- Different configurations can lead to different program behaviors

```
#options: s,t,u,v,x,y,z ∈ {0...4}
max_z = 3;
if x and y:
  #L0
  if not (0 < z < max_z):
    #L1

if s or t:
    #L2
    if u and v:
      return

#L3
```

- 7 options, 320 possible configs

Analyzing Configurable Software (in submission)

- Modern software is highly configurable
- Different configurations can lead to different program behaviors

```
#options: s, t, u, v, x, y, z ∈ {0...4}
max_z = 3;
if x and y:
  #L0
  if not (0 < z < max_z):
    #L1

if s or t:
    #L2
    if u and v:
      return

#L3
```

- 7 options, 320 possible configs

config	s	t	u	v	x	y	z	coverage
c_1	1	0	1	1	1	1	4	$L0, L1$
c_2	0	0	1	1	1	1	0	$L0, L3$
c_3	0	1	1	1	1	0	3	$L2$
	⋮							

Analyzing Configurable Software (in submission)

- Modern software is highly configurable
- Different configurations can lead to different program behaviors

```
#options: s, t, u, v, x, y, z ∈ {0...4}
max_z = 3;
if x and y:
  #L0
  if not (0 < z < max_z):
    #L1

if s or t:
    #L2
    if u and v:
      return

#L3
```

- 7 options, 320 possible configs

config	s	t	u	v	x	y	z	coverage
c_1	1	0	1	1	1	1	4	$L0, L1$
c_2	0	0	1	1	1	1	0	$L0, L3$
c_3	0	1	1	1	1	0	3	$L2$
	⋮							

- interactions (reaching conditions):

interaction	covering
$x \wedge y$	$L0$
$x \wedge y \wedge (z \in \{0, 3, 4\})$	$L1$
$s \vee t$	$L2$
$(\neg s \wedge \neg t) \vee (\neg u \vee \neg v)$	$L3$

Analyzing Configurable Software (in submission)

- Modern software is highly configurable
- Different configurations can lead to different program behaviors

```
#options: s, t, u, v, x, y, z ∈ {0...4}
max_z = 3;
if x and y:
  #L0
  if not (0 < z < max_z):
    #L1

if s or t:
    #L2
    if u and v:
      return

#L3
```

- 7 options, 320 possible configs

config	s	t	u	v	x	y	z	coverage
c_1	1	0	1	1	1	1	4	$L0, L1$
c_2	0	0	1	1	1	1	0	$L0, L3$
c_3	0	1	1	1	1	0	3	$L2$
	⋮							

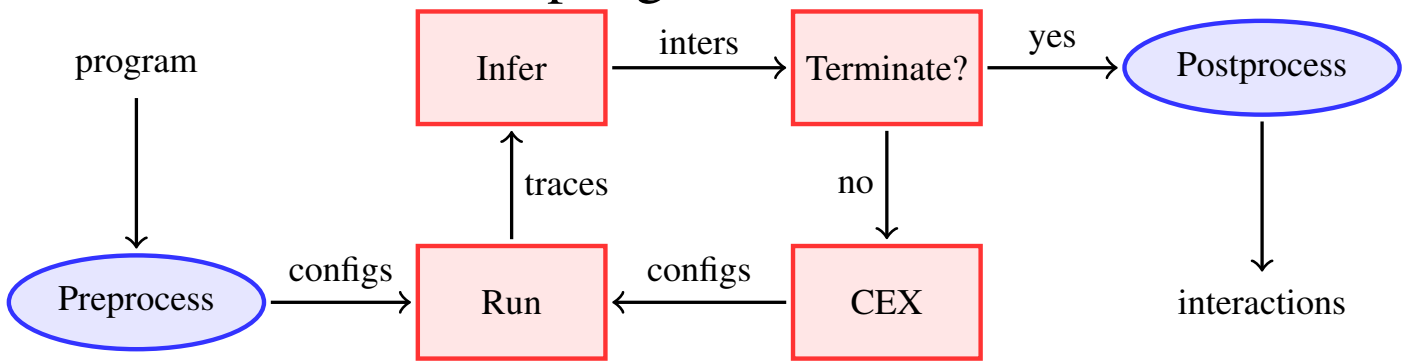
- interactions (reaching conditions):

interaction	covering
$x \wedge y$	$L0$
$x \wedge y \wedge (z \in \{0, 3, 4\})$	$L1$
$s \vee t$	$L2$
$(\neg s \wedge \neg t) \vee (\neg u \vee \neg v)$	$L3$

- Existing approaches do not scale and are restricted to simple interactions

iGen: a lightweight approach to finding interactions

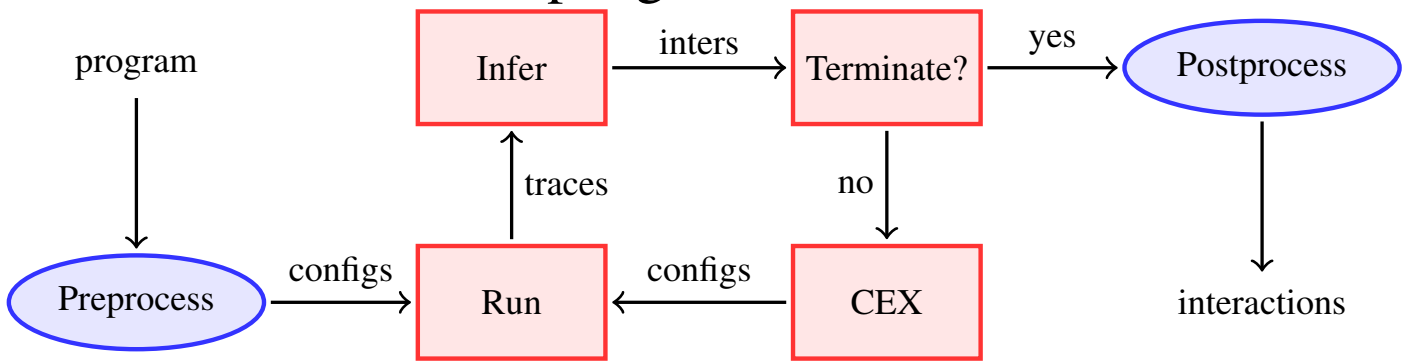
- Apply dynamic invariant analysis
- CEGIR: counter-example guided interaction refinement



- Analyze *negative* traces to compute more accurate interactions

iGen: a lightweight approach to finding interactions

- Apply dynamic invariant analysis
- CEGIR: counter-example guided interaction refinement



- Analyze *negative* traces to compute more accurate interactions

Results

- 25 open source software (GNU `coreutils`, Apache `httpd`, `unison`, `vsftpd`, etc) in different languages
- Generate accurate and useful interactions
  - Found bugs in the Perl implementation of GNU `coreutils`
  - Most interactions are small, but some are very large and complex
- More efficient than existing approaches (minutes vs weeks)

What's Next ?

What's Next ?

1. Invariant Generation
   - Apply invariant generations to analyze configurable software, feature-based interactions
   - DIG2: supports data structures including trees and lists and uses CEGIR for iterative invariant refinment
2. Dependent and Gradual Typing
   - Represent invariants as dependent types and apply current work to infer and check types
   - Use verification techniques (e.g., *k*-induction) to statically check dynamic languages (such as Python and Ruby)
3. Program Synthesis/Repair
   - Apply techniques in program synthesis to reachability, e.g., using repair tools to generate high quality test inputs
   - Combine CETI and other repair techniques, e.g., random search, to handle a wider range of errors

Conclusion

- Invariant Generation
  - **DIG**: treat invariants as set of equations and constraints and solve them to identify nonlinear polynomial relations, disjunctive invariants, and array properties
  - **KIP**: an automatic theorem prover for verifying candidate invariants
  - **iGen**: a lighweight dynamic analysis approach to finding program interactions
- Program Synthesis/Repair
  - **GenProg**: modify and repair programs using GA
  - **CETI**: apply equivalence theorem to reduce repair task to rechability problem, solvable using off-the-shelf test-input generators

References

- Source code, benchmarks, etc: http://www.cs.umd.edu/~tnguyen
- Invariant Generation
  - Nguyen, Kapur, Weimer, and Forrest. DIG: A dynamic invariant generator for polynomial and array invariants. TOSEM, 2014
  - Nguyen, Kapur, Weimer, and Forrest. Using dynamic analysis to generate disjunctive invariants. ICSE, 2014
  - Kapur, Zhang, Horbach, Zhao, Lu, and Nguyen. Geometric quantifier elimination heuristics for automatically generating octagonal and max-plus invariants. Automated Reasoning and Mathematics, 2013
  - Nguyen, Kapur, Weimer, and Forrest. Using dynamic analysis to discover polynomial and array invariants. ICSE, 2012 (Distinguished Paper)
- Program Repair
  - Geoues, Nguyen, Forrest, and Weimer. GenProg: A generic method for automated software repair. TSE, 2012
  - Weimer, Forrest, Geoues, and Nguyen. Automatic program repair with evolutionary computation. CACM, 2010 (Research Highlight)
  - Forrest, Weimer, Nguyen, and Geoues. A genetic programming approach to automated software repair. GECCO, 2009 (Best Paper)
  - Weimer, Nguyen, Geoues, and Forrest. Automatically finding patches using genetic programming. ICSE, 2012 (Distinguished Paper and Manfred Paul Award for Excellence in Software: Theory and Practice)
  - Nguyen, Weimer, Geoues, and Forrest. Using execution paths to evolve software patches. ICST, 2009 (Best Paper)