Numerical Program Analysis via Mathematical Execution

Zhendong Su

ETH Zurich
Floating-point code

- **Important**: bugs can lead to disasters
- **Challenging**: hard to get right
Why difficult?

- FP Math $\neq$ Real Math
- Non-linear relations
- Transcendental functions
  - $\sin$, $\log$, $\exp$, ...

Challenging for all known approaches

```java
double foo(double x){
    if (x<=1.0)
        x++;
    y = x*x;
    if (y<=4.0)
        x--;
    return x;
}
```
New perspective: ME

Analyzing numerical programs
• Coverage-based testing
• Boundary value analysis
• Numerical exception detection
Floating-point constraint solving

Mathematical Execution (ME)

Mathematical optimization (MO)

\[(p, \phi)\]

input \(x\) drives \(p\) to satisfy \(\phi\) \iff \(x\) minimizes \(r\)
FP constraints

Solving the floating-point constraint $\pi$

$$(\text{SIN}(x) \equiv x) \land (x \geq 10^{-10})$$

- Satisfiable if $x$ is floating-point

  For $x \in \mathbb{F}$, $\text{SIN}(x) = x \iff x \simeq 0$

- Unsatisfiable if $x$ is real

  For $x \in \mathbb{R}$, $\text{SIN}(x) = x \iff x = 0$
Step 1

Simulate $\pi$ with a floating-point program $R$

- $R(x) \geq 0$ for all $x$
- $R(x) = 0 \iff x \models \pi$
Step 2

Minimize $R$ as if it is a mathematical function

- Let $x^*$ be the minimum point

$$\pi \text{ satisfiable } \iff R(x^*) = 0$$
Construct $R$

Necessary Conditions to meet:
1. $R(x) \geq 0$ for all $x$
2. $R(x) = 0 \iff x \models \pi$

How?

<table>
<thead>
<tr>
<th>Constraint $\pi$</th>
<th>Program $R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x == y$</td>
<td>$(x - y)^2$</td>
</tr>
<tr>
<td>$x \leq y$</td>
<td>$x \leq y \ ? 0 : (x - y)^2$</td>
</tr>
<tr>
<td>$\pi_1 \land \pi_2$</td>
<td>$R_1 + R_2$</td>
</tr>
<tr>
<td>$\pi_1 \lor \pi_2$</td>
<td>$R_1 * R_2$</td>
</tr>
</tbody>
</table>

$R$ can be constructed from a CNF form
Minimize R

Unconstrained programming techniques:

- Local optimization
- Monte Carlo Markov Chain (MCMC)
- We use them as black-box
- Do not analyze $\pi$; execute $R$
Theoretical guarantees

Let $R$ satisfy (1) $R(x) \geq 0$, and (2) $R(x) = 0 \iff x \models \pi$, and $x^*$ be a minimum point of $R$. Then

$$\pi \text{ satisfiable } \iff R(x^*) = 0.$$  

Threats

- Floating-point inaccuracy when calculating with $R$
- Sub-optimal $x^*$
Example

\[(\text{SIN}(x) == x) \land (x \geq 10^{-10})\]

\[\implies (\text{SIN}(x) - x)^2 + \begin{cases} 
0 & \text{if } x \geq 10^{-10} \\
(x - 10^{-10})^2 & \text{otherwise}
\end{cases}\]

\[\implies x^* = 9.0 \times 10^{-9} \text{ (can be others)}\]
XSat & results

- Developed the ME-based XSat tool
- Evaluated against MathSat and Z3
- Used SMT-Comp 2015 FP benchmarks
- Result summary
  - **100%** consistent results
  - **700+X** faster than MathSat
  - **800+X** faster than Z3
Generalizations

- Coverage-based testing of FP code
- Boundary value analysis
- FP exception detection
- Path divergence detection
Coverage-based testing

Goal

To generate test inputs to cover all branches of a program like this:

- pointer operations: &, *
- type casting: (int*), (unsigned)
- bit operations ^, &, >>
- floating-point comparison

```c
double __ieee754_fmod(double x, double y) {
    ...
    Zero[] = {0.0, -0.0,};
    hx = *(1+(int*)&x);
    lx = *(int*)&x;
    hy = *(1+(int*)&y);
    ly = *(int*)&y;
    sx = hx&0x80000000;
    hx ^=sx;
    hy &= 0x7fffffff;

    if((hy|ly)==0||(hx>=0x7ff00000)||(hy|(ly-ly)>>31)>0x7ff00000))
        return (x*y)/(x*y);
    if(hx<=hy) {
        if((hx<hy)||(lx<ly)) return x;
        if(lx==ly)
            return Zero[(unsigned)sx>>31];
    }

    if(hx<0x00100000) {
        if(hx==0) {
```
State-of-the-art & Challenges

**Symbolic execution**
- Path explosion
- Constraint solving

**Search-based testing**
- Fitness function
- Search strategies

**Our approach**
- No path issues
- No need to solve constraints
- Effective for FP programs
Our approach

Valid inputs

Inputs that cover a new branch

Step 1: Derive a program F00_R from F00 s.t.

- $F00_R(x) \geq 0$ for all $x$, and
- $F00_R(x) = 0 \iff x$ covers a new branch

Step 2: Repeatedly minimize F00_R until $> 0$
void F00(double x) {
    if (x <= 1) {
        // branch $0_T$
        x = x + 1;
    } else {
        // branch $0_F$
    }
}

double square(double x) { return x * x; }

---

**Step 1.**

**F00_I: Instrumented program**

```c
void F00_I(double x) {
    double r;
    r = pen (l0, x, 1);
    l0: if (x <= 1) {
        // branch $0_T$
        x = x + 1;
    } else {
        // branch $0_F$
    }
    double y = square(x);
    l0: if (y == 4) {
        // branch $1_T$
    } else {
        // branch $1_F$
    }
}
```

---

**Step 2.**

**F00_R: Representing function**

```c
double F00_R(double x) {
    r = 1; F00_I(x); return r;
}
```

---

**Step 3.**

**Generated test inputs**

$\mathcal{X}$: A set of F00_R's global minimum points, which saturates (therefore covers) all branches of F00
double r; // global variable

void FOO_I(double x) {
    r = pen (l_0, \leq, x, 1);
    l_0: if (x \leq 1) {
        // branch 0_T
        x = x + 1;
    } else {
        // branch 0_F
    }
    double y = square (x);
    r = pen (l_1, =\!, y, 4);
    l_1: if (y =\! 4) {
        // branch 1_T
    } else {
        // branch 1_F
    }
}

if (neither 0_T or 0_F is saturated)
    return 0;
else if (0_F is saturated but 0_T is not)
    return (x \leq 1) ? 0 : (x - 1)^2;
else if (0_T is saturated but 0_F is not)
    return (x > 1) ? 0 : (x - 1)^2 + \varepsilon;
el else return r;

if (neither 1_T or 1_F is saturated)
    return 0;
else if (1_F is saturated but 1_T is not)
    return (y - 4)^2;
else if (1_T is saturated but 1_F is not)
    return (y \neq 4) ? 0 : \varepsilon;
el else return r;
**F00_I: Instrumented program**

```c
double r; // global variable
void F00_I(double x) {
    r = pen (l0, ≤, x, 1);
    l0: if (x <= 1) {
        // branch 0_T
        x = x + 1;
    } else {
        // branch 0_F
    }
    double y = square (x);
    r = pen (l1, ==, y, 4);
    l1: if (y == 4) {
        // branch 1_T
    } else {
        // branch 1_F
    }
}
```

```c
if (neither 0_T or 0_F is saturated) return 0;
else if (0_F is saturated but 0_T is not)
    return (x ≤ 1) ? 0 : (x-1)^2;
else if (0_T is saturated but 0_F is not)
    return (x > 1) ? 0 : (x-1)^2 + ε;
else return r;
```

```c
if (neither 1_T or 1_F is saturated)
    return 0;
else if (1_F is saturated but 1_T is not)
    return (y-4)^2;
else if (1_T is saturated but 1_F is not)
    return (y≠4) ? 0 : ε;
else return r;
```

**F00_R: Representing function**

```c
double F00_R(double x) {
    r = 1; F00_I(x); return r;
}
```

**Generated test inputs**

\( X \): A set of F00_R's global minimum points, which saturates (therefore covers) all branches of F00
Example

Generate an input set to cover \( \{0_T, 0_F, 1_T, 1_F\} \)

```c
void FOO (double x){
    /0: if (x < 1)
        x++;
        double y = x * x;
    /1: if (y == 4)
        ...
}
```
Step 1: Construct F00_R

<table>
<thead>
<tr>
<th>covered at $l_i$</th>
<th>pen($l_i$, op, a, b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>0</td>
</tr>
<tr>
<td>${i_F}$</td>
<td>$R_a \ op \ b$</td>
</tr>
<tr>
<td>${i_T}$</td>
<td>$R_{\neg\ (a \ op \ b)}$</td>
</tr>
<tr>
<td>${i_T, i_F}$</td>
<td>$r$</td>
</tr>
</tbody>
</table>

- $r$: global variable
- F00_R : $x \rightarrow r$
- $R_a \ op \ b$: Branch distance

```
r = 1 
```
```
Input $x$
```
```
r = pen($l_0$, $<$, $x$, 1) 
```
```
l_0: if $x < 1$
```
```
x = x + 1
```
```
r = pen($l_1$, $==$,$y$, 4) 
```
```
l_1: if $y == 4$
```
```
y = x * x
```
```
x = x + 1
```
```
r = pen($l_0$, $<$, $x$, 1) 
```
```
l_0: if $x < 1$
```
```
x = x + 1
```
```
r = pen($l_1$, $==$,$y$, 4) 
```
```
l_1: if $y == 4$
```
```
y = x * x
```
Branch distance $R_a \ op \ b$

A helper function to quantify how far $a$ and $b$ are from attaining branch $a \ op \ b$.

$R_{a==b}$ defined as $(a - b)^2$
$R_{a\geq b}$ defined as $(a \geq b) \ ? 0 : (a - b)^2$
Step 2: Minimize F00_R

<table>
<thead>
<tr>
<th>covered at $l_i$</th>
<th>pen($l_i, op, a, b$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>0</td>
</tr>
<tr>
<td>${i_F}$</td>
<td>$R_{a ; op ; b}$</td>
</tr>
<tr>
<td>${i_T}$</td>
<td>$R\neg(a ; op ; b)$</td>
</tr>
<tr>
<td>${i_T, i_F}$</td>
<td>$r$</td>
</tr>
</tbody>
</table>

- No branch is covered
- Any input is a minimum point
- Assume $x^* = 0.7$
Step 2: Minimize F00_R

<table>
<thead>
<tr>
<th>covered at $l_i$</th>
<th>$pen(l_i, op, a, b)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>0</td>
</tr>
<tr>
<td>${i_F}$</td>
<td>$R_a \ op \ b$</td>
</tr>
<tr>
<td>${i_T}$</td>
<td>$R_{-}(a \ op \ b)$</td>
</tr>
<tr>
<td>${i_T, i_F}$</td>
<td>$r$</td>
</tr>
</tbody>
</table>

- $1_F, 0_T$ are covered
- F00_R attains minimum at -3 or 2
- Assume $x^* = -3$
Step 2: Minimize F00_R

<table>
<thead>
<tr>
<th>covered at $l_i$</th>
<th>pen($l_i$, $op$, $a$, $b$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>0</td>
</tr>
<tr>
<td>${i_F}$</td>
<td>$R_a \ op \ b$</td>
</tr>
<tr>
<td>${i_T}$</td>
<td>$R_{\neg}(a \ op \ b)$</td>
</tr>
<tr>
<td>${i_T, i_F}$</td>
<td>$r$</td>
</tr>
</tbody>
</table>

- $1_F, 1_T, 0_T$ are covered
- F00_R attains minimum at $\geq 1$
- Assume $x^* = 5.1$

Diagram:

1. Input $x$
2. $r = (x \geq 1) ? 0 : (x - 1)^2$
3. $l_0$: if $x < 1$
   - $0_T$
   - $x = x + 1$
4. $y = x \ast x$
5. $r = r$
6. $l_1$: if $y == 4$
   - $1_T$
   - $1_F$

Blue: covered
Step 2: Minimize F00_R

<table>
<thead>
<tr>
<th>covered at $l_i$</th>
<th>pen($l_i, op, a, b$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>0</td>
</tr>
<tr>
<td>${i_F}$</td>
<td>$R_{a \ op \ b}$</td>
</tr>
<tr>
<td>${i_T}$</td>
<td>$R_{\neg (a \ op \ b)}$</td>
</tr>
<tr>
<td>${i_T, i_F}$</td>
<td>$r$</td>
</tr>
</tbody>
</table>

- All branches are covered
- $\forall x, F00_R(x) = 1$
- Termination

![Control Flow Diagram](image)

Blue: covered
Our implementation CoverMe

- **Front-end**: Program transformation in LLVM
- **Back-end**: Basinhopping
Experiments

Benchmarks: Fdlibm

- Sun’s math library
- Reference for Java SE 8’s math library
- Used in Matlab, JavaScript and Android
- Heavy on branches (max=114, avg=23)
Results

CoverMe covers

- $\approx 90\%$ branches in 7 seconds
- $\approx 18\%$ more branches than AFL with $1/10$ time
- $\approx 40\%$ more branches than Austin with speedups of several orders of magnitudes
ME in the long run

- Offers a new general analysis paradigm
- Complements existing approaches
  - Random concrete execution (CE)
  - Symbolic execution (SE)
  - Abstract execution (AE)
New perspective: ME

Analyzing numerical programs
- Coverage-based testing
- Boundary value analysis
- Numerical exception detection

Floating-point constraint solving

Mathematical Execution (ME)

Mathematical optimization (MO)

(input $x$ drives $p$ to satisfy $\phi$ $\iff$ $x$ minimizes $r$)