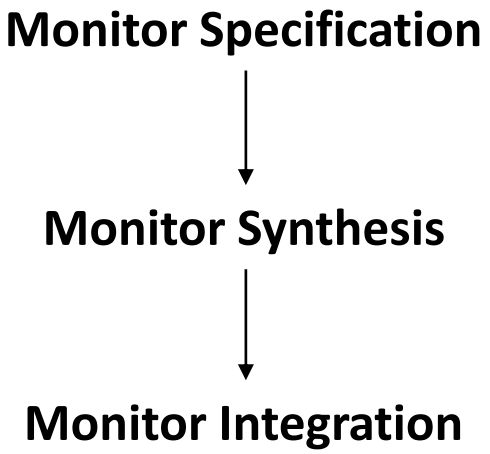
Monitor Oriented Programming with JavaMOP

Reading: Monitoring Oriented Programming - A Project Overview.

Monitor Oriented Programming

- Software reliability is a concern
  - Need to monitor the program behavior for violation of specification
- Monitor Oriented Programming
  - A generic framework
  - Checks specification with implementation at runtime

MOP Framework

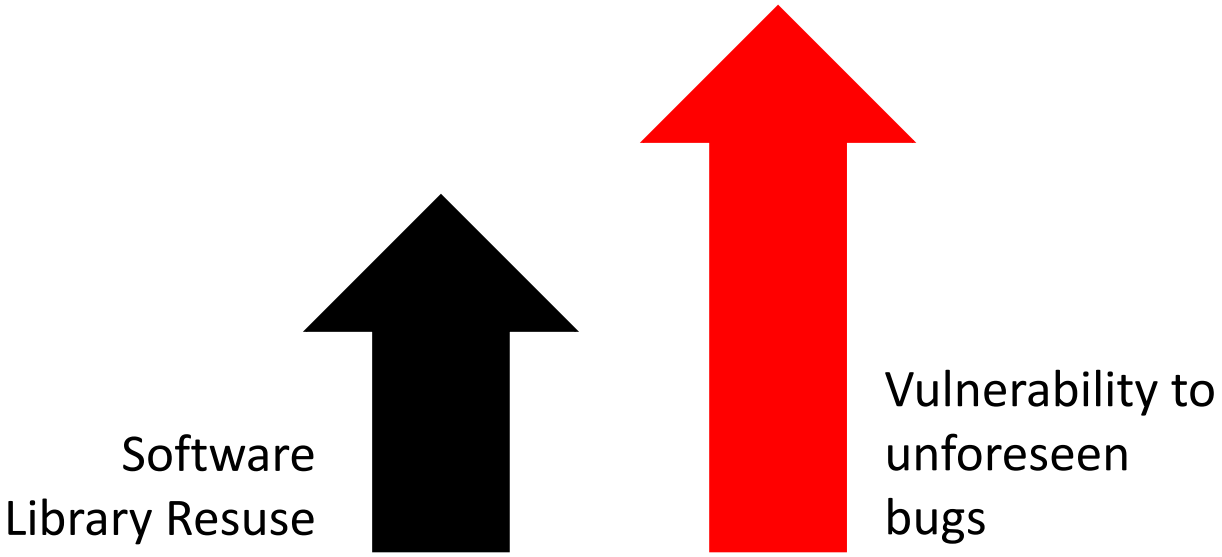


Choice of formalism
LTL, CTL, RE, …

Choice of languages
Java, C, …

Choice of join points
Method-level, Wildcard Support, Before/After Invocation …

The Problem


Software
Library Resuse
Vulnerability to
unforeseen
bugs

HashSet Documentation

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. **Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this** exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

HashSet with Iterator

```
private static void safe_HS() {
  HashSet set = new HashSet();
  for (int i=0; i<100; i++) {
    int rnd = (int) (Math.random() * 10000);
    set.add(new Integer(rnd));
  }

  Iterator iter = set.iterator();
  int sum = 0;
  while (iter.hasNext()) {
    sum += (Integer)iter.next();
  }
  System.out.println("sum: " + sum);
}
```

Safety Property: hasNext() must precede next()

Adds 100 random numbers to a HashSet.
Calculates and prints the sum of these numbers.

Bad Iterator

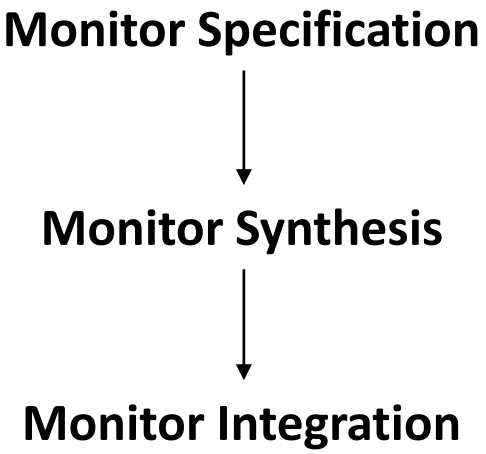
```
public static void badIterator(){
        HashSet<Integer> set = new HashSet<Integer>();
        for(int i = 0; i < 100; ++i){
            set.add(new Integer(i));
        }
        Iterator i = set.iterator();

        int sum = 0;
        for(int j = 0; j < 100; ++j){
          if (i.hasNext()) {
              sum += (Integer)i.next();
              set.add(new Integer(j));
           }
        }
        System.out.println(sum);
}
```

Not allowed to change the set directly while using an iterator.

Questions

- Can we inspect the code at runtime and report if a HashSet is updated directly when an iterator is iterating?
- Can we check if the next() method is called without calling hasNext() method?
- Can we measure time taken by add() method over several runs and ensure that it is operating in constant time?

Recall…. MOP Framework



Choice of formalism
LTL, CTL, RE, …

Choice of languages
Java, C, …

Choice of join points
Method-level, Wildcard Support, Before/After Invocation …

A Monitor Specification

```
import java.io.*;
import java.util.*;

suffix HasNext(Iterator i) {
        event hasnext
              before(Iterator i) :
              call(* Iterator.hasNext())
              && target(i) {}
        event next before(Iterator i) :
              call(* Iterator.next())
              && target(i) {}

        ere : next next

        @match {
            System.out.println(
              "@:" + __LOC + " ! hasNext not called before next");
            __RESET;
        }
}
```

Uses AspectJ syntax for a pointcut

//hasNext is the event id

//i can be used in the advice.

// The specification

__LOC is the line number where the current event is generated

__RESET resets the monitor to its initial state;

Monitoring

- Generates an execution trace
  - Only consisting of events that the user is interested in.
- On validation/violation of the given property,
  - appropriate actions will be triggered.
- In our case
  - Generates a trace
    - hasNext next hasNext next ….
  - suffix match looks for the suffix “next next”

JavaMOP Syntax

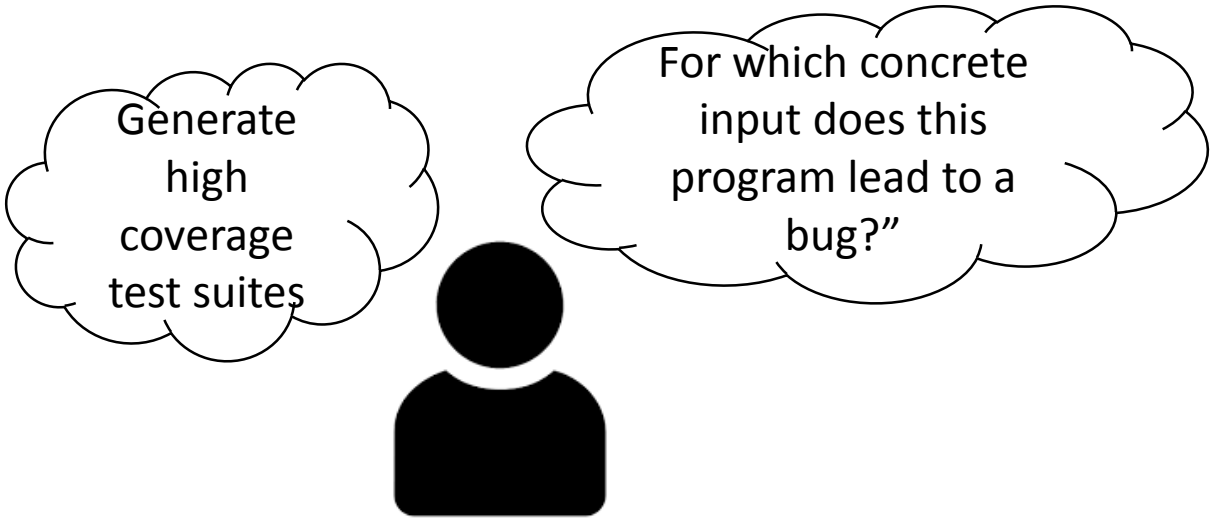
- For detailed syntax, visit http://fsl.cs.illinois.edu/index.php/JavaMOP_Syntax

JavaMOP Tutorial

Symbolic Execution

Reading: Symbolic Execution for Software Testing: Three Decades Later, Cadar and Sen.

Objectives


Generate high coverage test suites
For which concrete input does this program lead to a bug?”

Error

- **For what concrete inputs of** (x,y) **will** testme(x,y) **result in an error?**

```
int twice (int v) {
    return 2 * v;
}

void testme (int x, int y) {
    z = twice (y );
    if (z == x) {
        if (x > y+10) ERROR;
    }
}
```

Brute force testing with random inputs has extremely low probability of hitting the ERROR.

Goals

- Cover as many program paths as possible in limited time.
- For each path,
  - Generate concrete values that lead to that path
  - Check for errors, assertion violations, exceptions, etc.,

Bugs Vs. Program Assertions

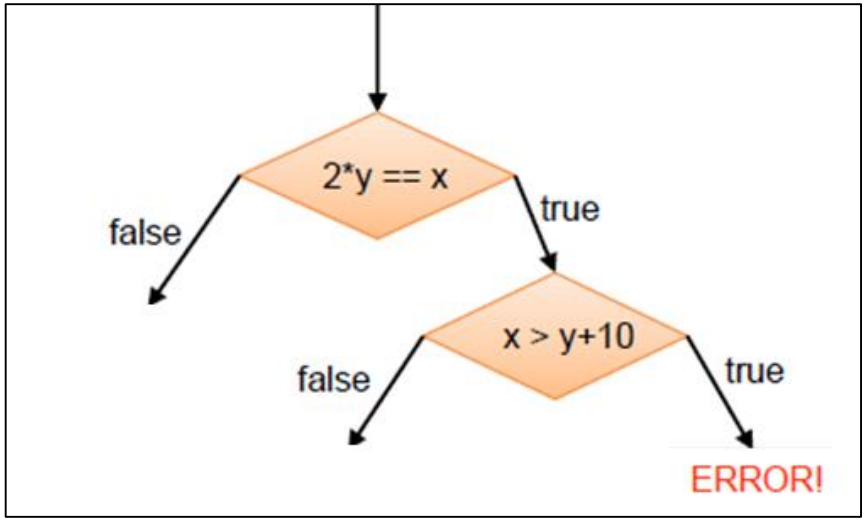
Null Pointers
Buffer Overflow
Array Index Out Of Bounds
…

Complex Program Properties
(Program Assertions)

```
int twice (int v) {
   return 2 * v;
}

void testme (int x, int y) {
   z = twice (y );
   if (z == x)
   {
      if (x > y+10)
         ERROR;
   }
}

int main() {
   x = sym input();
   y = sym input();
   testme (x, y );
   return 0;
}
```

Execution Tree

```
z = 2*y ;
if (z == x)
{
    if (x > y+10)
        ERROR;
}
```



Execution Tree

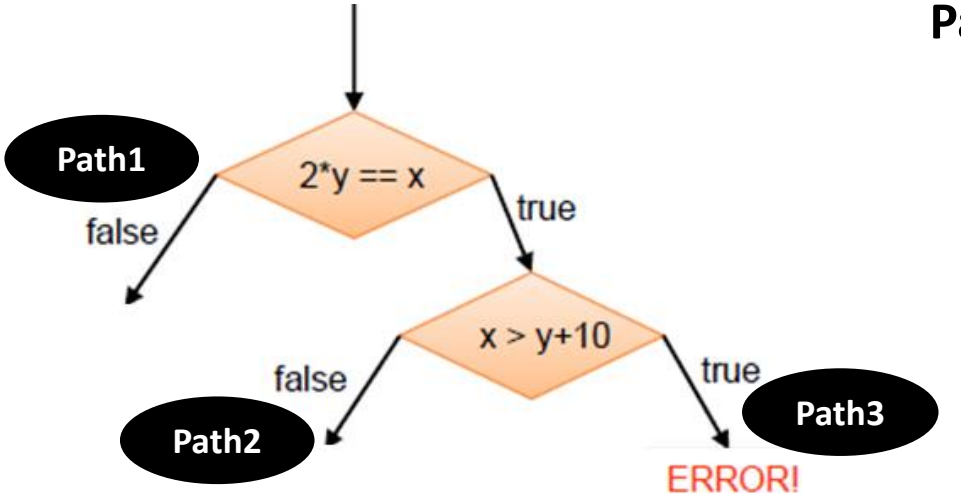
Symbolic variables

$(x_0 = 2y_0) \wedge (x_0 > y_0 + 10)$

A Path Constraint

Example taken from Symbolic Execution for Software Testing: Three Decades Later, Cadar and Sen.

Symbolic Execution



Path constraints for full path coverage

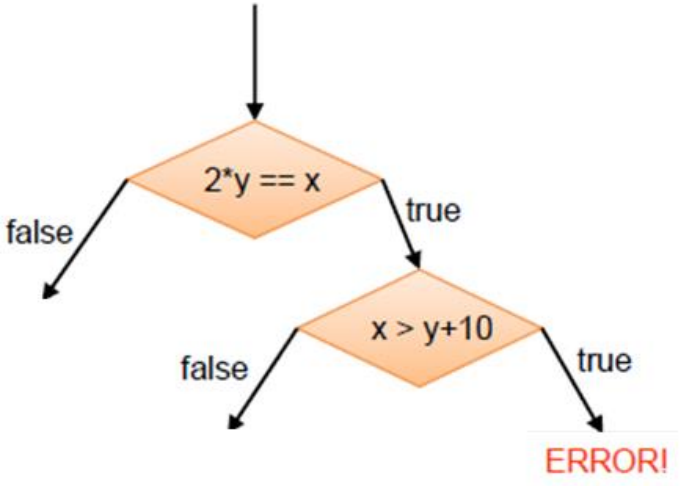
Path1: !C1,
Path2: C1 && C2,
Path3: C1 && !C2

where
C1: x = 2y
C2: x > (y+10)

Path Constraint
$x_0 = 2y_0$

→ **Constraint Solver** → {x = 2, y = 1}

Symbolic Execution

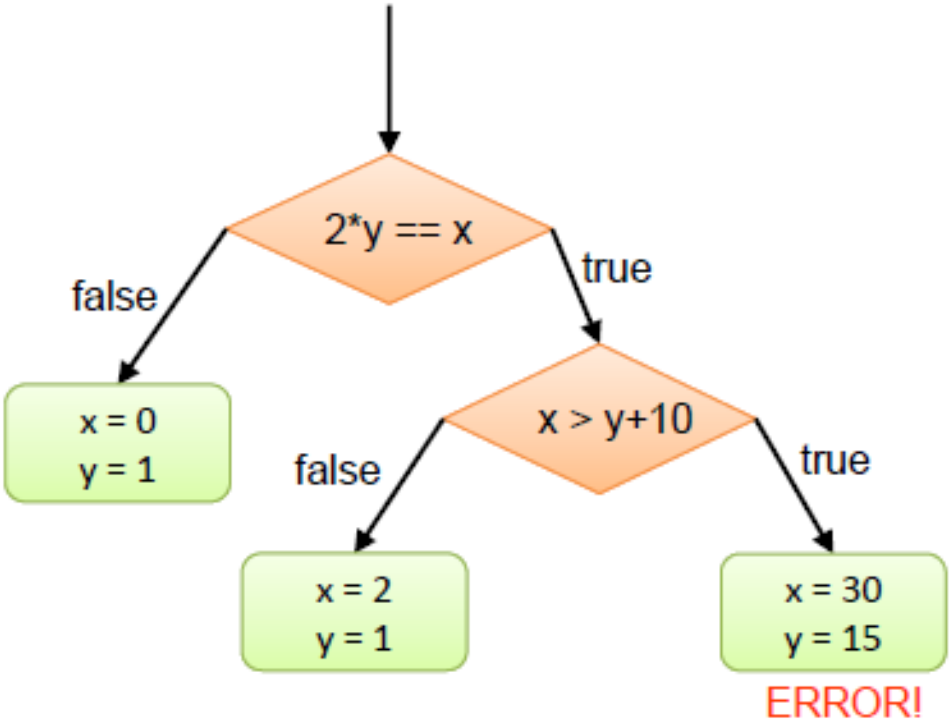


Path Constraints	Constraint Solver Output
Path1: !C1	{x = 22, y = 7}
Path2: C1 && C2	{x = 2, y = 1}
Path3: C1 && !C2	{x = 30, y = 15}

Has applications in Automated Test Generation

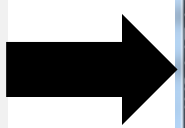
Execution Tree

```
int twice (int v) {
    return 2 * v;
}

void testme (int x, int y) {
    z = twice (y );
    if (z == x)
    {
        if (x > y+10)
            ERROR;
    }
}

testme (x, y );
```



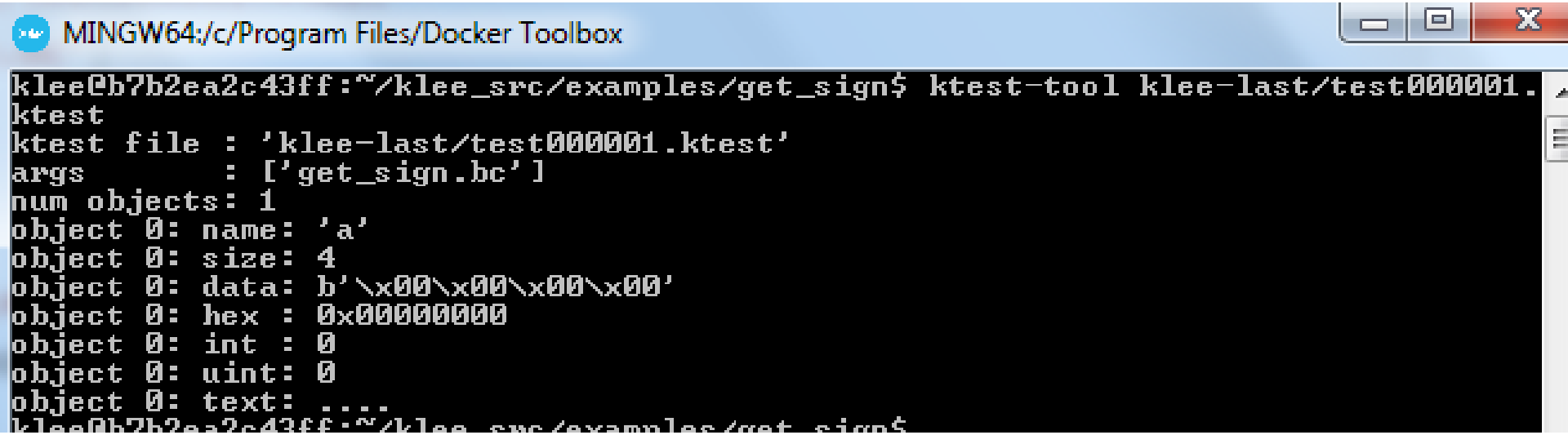
KLEE Tutorial

```c
int get_sign(int x) {
  if (x == 0)
    return 0;

  if (x < 0)
    return -1;
  else
    return 1;
}
```



```
MINGW64:/c/Program Files/Docker Toolbox
    return 1;
}

int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
klee@b7b2ea2c43ff:~/klee_src/examples/get_sign$ clang -I
m -c -g -O0 -Xclang -disable-O0-optnone get_sign.c
klee@b7b2ea2c43ff:~/klee_src/examples/get_sign$ clang -e
get_sign.c:5:10: fatal error: 'klee/klee.h' file not fou
#include <klee/klee.h>
         ^~~~~~~~~~~~~

1 error generated.
klee@b7b2ea2c43ff:~/klee_src/examples/get_sign$ clang -I
m -c -g -O0 -Xclang -disable-O0-optnone get_sign.c
klee@b7b2ea2c43ff:~/klee_src/examples/get_sign$ klee get
KLEE: output directory is "/home/klee/klee_src/examples/
KLEE: Using STP solver backend

KLEE: done: total instructions = 33
KLEE: done: completed paths = 3
KLEE: done: generated tests = 3
klee@b7b2ea2c43ff:~/klee_src/examples/get_sign$
```

KLEE generated Tests



Disadvantages of Symbolic Execution

- Constraints should be simple enough that a constraint solver is able to (efficiently) solve them.
  - E.g., (2^x) % large_prime == 13
- In the case of loops or recursion, we may need to put bounds on the number of iterations.
- Several custom functions may be uninterpreted.

Concolic Execution

- Concolic = Concrete + Symbolic
- Maintains both symbolic states as well as concrete states.

```
y= passwordHash(x);
if  (z == x)
{
        …
 } else {
        ERROR;
 }
```

Constraint solver cannot execute the uninterpreted function passwordHash(x)

Concolic Execution

```
hash = passHash(pass);
if (x == hash)
{
        …PATH1…
 } else {
        …PATH2…
 }
```

Input	Concrete Execution
{pass = a, x = 1}	Execute passHash("a"). Let it be 4000900977878888 Leads to PATH2.
{pass = a, x = 4000900977878888}	Leads to PATH1.

Concolic Execution

- Perform Symbolic Execution dynamically
- Run the program on concrete inputs.
  - One way is to start with random input values.
- Maintain a **concrete state** and a **symbolic state**

Hybrid Analysis

Limitations of Static Analysis

- Several program features make static analysis imprecise
  - Dynamic Binding
  - Threads
  - Polymorphism
  - Reuse of third party libraries
  - Native calls
  - Obfurscation
  - …

Limitations of Dynamic Analysis

- Statement Coverage
  - Not all path constraints can be solved.
- Path-specific analysis
  - Many paths may remain unexplored.
- Need to run the program (Can’t launch a rocket!)

Quick Review – Program Representations

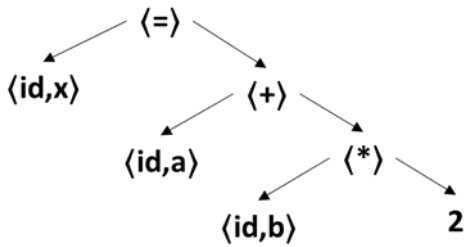
Lexemes

x = a + 2*b

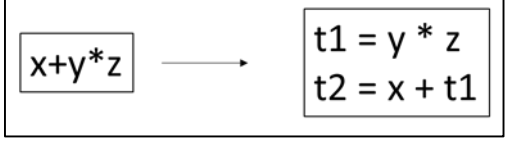
Tokens

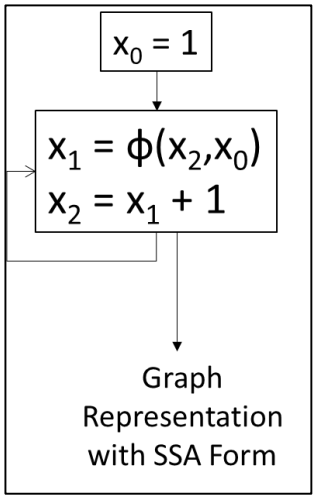
[(identifier, x), (operator, =), (identifier, a), (operator, +), (identifier, b), (operator, *), (literal, 2), (separator, ;)]

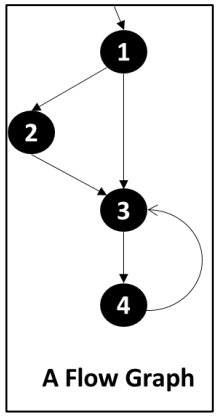
Abstract Syntax Tree

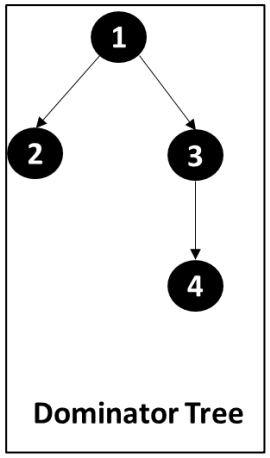


Three Address Code

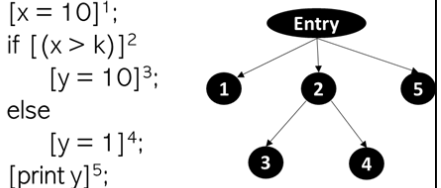


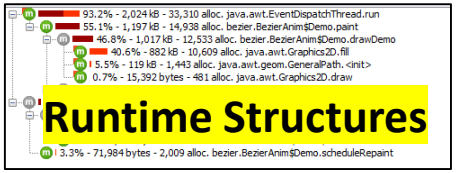






Control Dependence

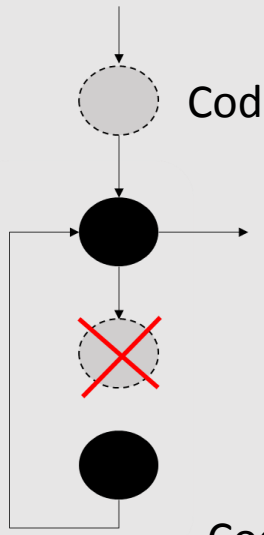




Tools

Eclipse JDT
Soot
JProfiler
Z3

Quick Review – Popular Compiler Optimizations



Loop Invariant Code Motion

```
int a = 30;
int b = 9 - (a / 5);
int c;

c = b * 4;
if (c > 10) {
  c = c - 10;
}
return c * (60 / a);
```

=

```
return 4;
```

Constant Folding and Propagation

$$prog: I_{fixed} \times I_{dynamic} \rightarrow O$$

$$\downarrow$$

$$prog^{*}: I_{dynamic} \rightarrow O$$

Partial Evaluation

Before	After
y = x / 8	y = x >> 3
y = x * 64	y = x << 6
y = x * 2	y = x << 1

Strength Reduction

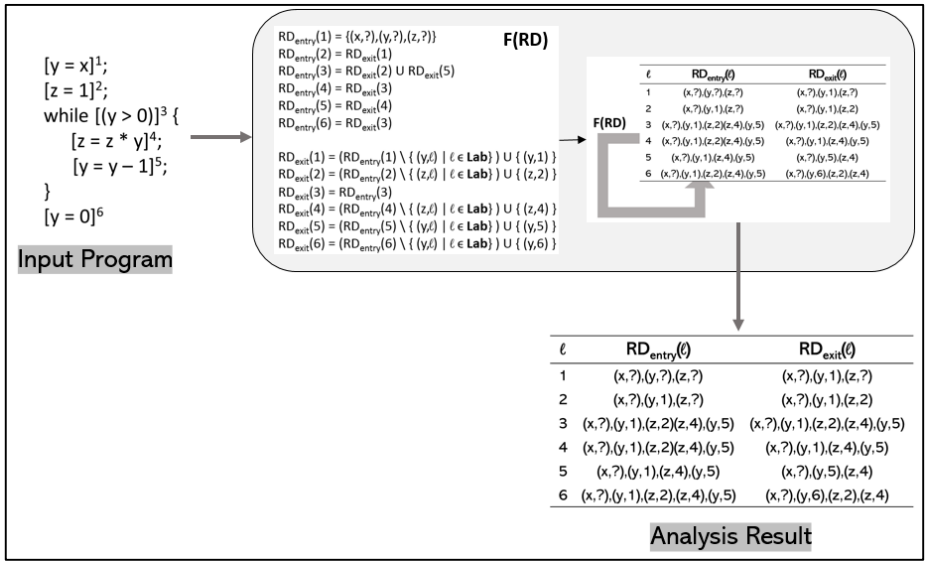
Peephole Optimization

```
float x,y,a,b,c,d;
…
x = (a/b)*c;
y = (a/b)*d;
```

Common Subexpressions

Quick Review

Data Flow as a System of Equations



Classic Four Analyses

$VB_{entry}(\ell)$	$(VB_{exit}(\ell) \setminus kill_{VB}(B^\ell)) \cup gen_{VB}(B^\ell)$	Backward Analyses
$VB_{exit}(\ell)$	$\emptyset$ if $\ell \in final(S_*)$ $\bigcap \{VB_{entry}(\ell') \mid (\ell', \ell) \in flow^R(S_*)\}$ otherwise	
$LV_{entry}(\ell)$	$(LV_{exit}(\ell) \setminus kill_{LV}(B^\ell)) \cup gen_{LV}(B^\ell)$	
$LV_{exit}(\ell)$	$\emptyset$ if $\ell \in final(S_*)$ $\bigcup \{LV_{entry}(\ell') \mid (\ell', \ell) \in flow^R(S_*)\}$ otherwise	
$RD_{entry}(\ell)$	$\{(x,?) \mid x \in Var_*\}$ if $\ell = init(S_*)$ $\bigcup \{RD_{exit}(\ell') \mid (\ell', \ell) \in flow(S_*)\}$ otherwise	Forward Analyses
$RD_{exit}(\ell)$	$(RD_{entry}(\ell) \setminus kill_{RD}(B^\ell)) \cup gen_{RD}(B^\ell)$	
$AE_{entry}(\ell)$	$\emptyset$ if $\ell = init(S_*)$ $\bigcap \{AE_{exit}(\ell') \mid (\ell', \ell) \in flow(S_*)\}$ otherwise	
$AE_{exit}(\ell)$	$(AE_{entry}(\ell) \setminus kill_{AE}(B^\ell)) \cup gen_{AE}(B^\ell)$	

Data Flow Analysis Framework

A Data Flow Analysis framework (P, ^, F) consists of:

- A bounded semilattice (P, ^)
  - Bounded since we need finite length ascending chain.
- A monotone transfer function
  - x <= y implies f(x) <= f(y)

forward or reverse flow.

union or intersection

init(S*) or final(S*)

$$Analysis_*(\ell) = \bigsqcup \{Analysis_*(\ell') \mid (\ell', \ell) \in F\} \sqcup \iota_E^\ell$$

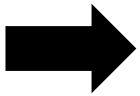
$$\text{where } \iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases}$$

$$Analysis_*(\ell) = f_\ell(Analysis_*(\ell))$$

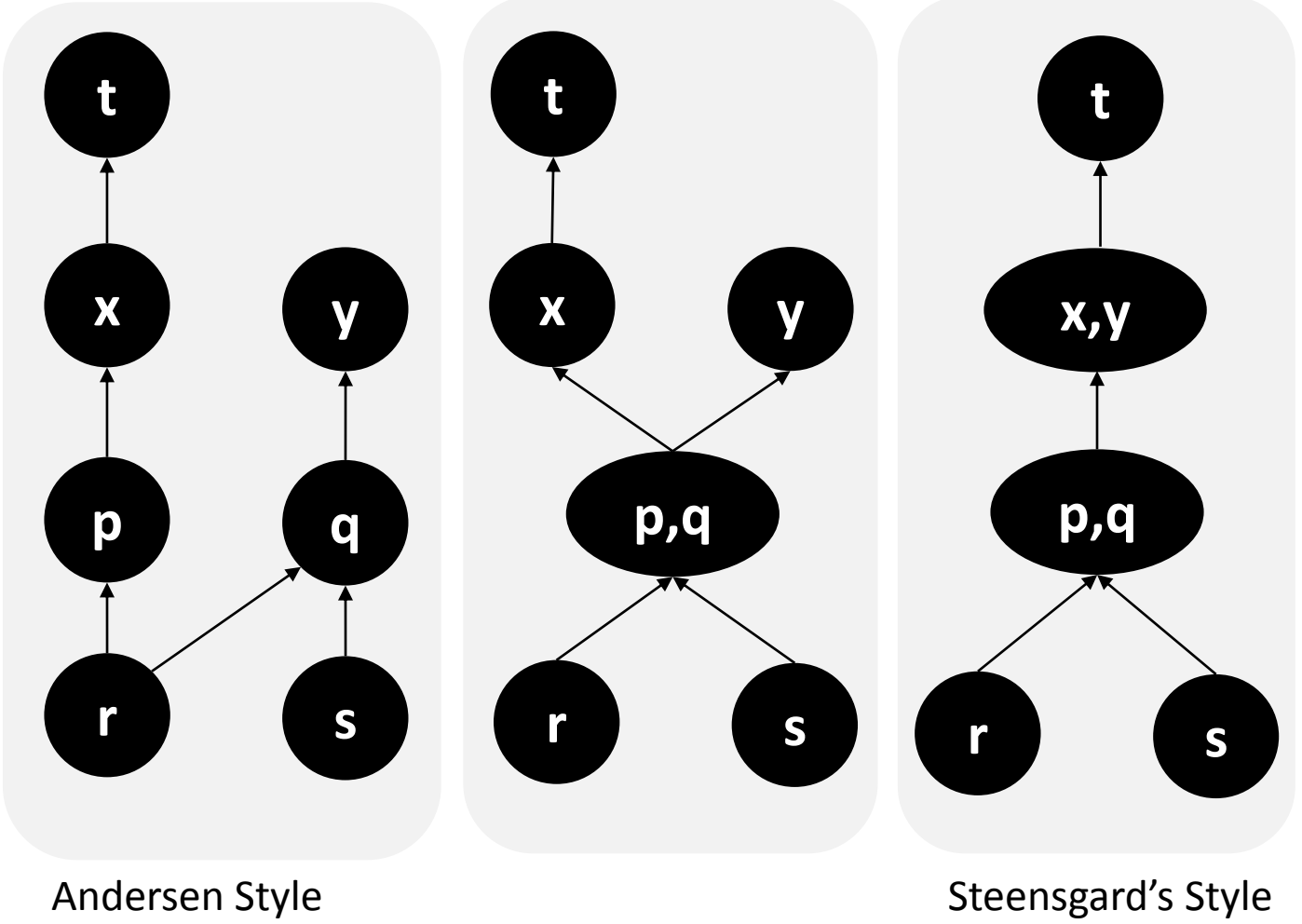
Soot

Points-To Analysis

p = &x
r = &p
q = &y
s = &q
r = s
*p = &t



Points-to Graphs



Andersen Style

Steensgard's Style

Topics in Static Analysis

- Symbolic Execution
- Non-Bit Vector Analyses (General Frameworks)

Dynamic Analysis

- Aspect Oriented Programming
- Monitor Oriented Programming
- Concolic Execution

Research Trends

- Program Slicing

```
read(N)
x=0
s=0
p=1
for (i=0; i < N; i++)
    s =s + 1
    p = p * (i+1)
    x = x+2
endfor
if (N>=0 && s != N)
   x= 1
endif
write(x)
```

```
read(N)
x=0
s=0
p=1
for (i=0; i < N; i++)
    s =s + 1
    p = p * (i+1)
    x = x+2
endfor
if (N>=0 && s != N)
   x= 1
endif
write(x)
```

Thank You!