Heap Cloning: Enabling Dynamic Symbolic Execution of Java Programs

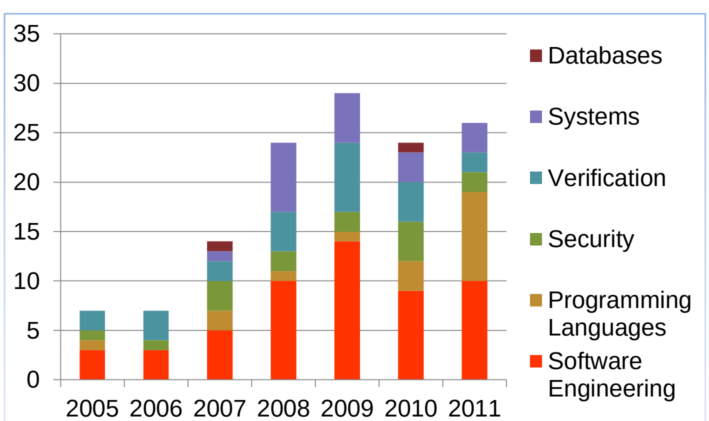
Saswat Anand
Mary Jean Harrold

Supported by NSF (CCF-0725202, CCF-0541048) and
IBM (Software Quality Innovation Award)

Aristotle Research Group

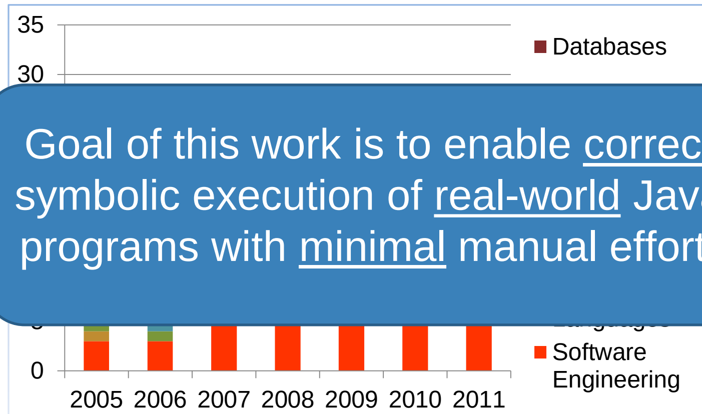
Georgia Tech College of Computing

Symbolic Execution



Estimated number of publications related to symbolic execution. The list of papers available at: http://sites.google.com/site/symexbib

Symbolic Execution



Goal of this work is to enable correct symbolic execution of real-world Java programs with minimal manual effort.

Estimated number of publications related to symbolic execution. The list of papers available at: http://sites.google.com/site/symexbib

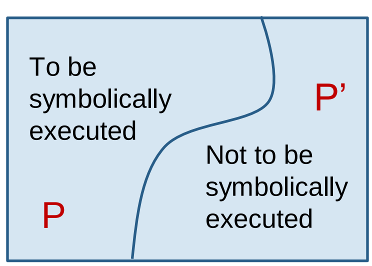
Outline

- Problem
- Heap Cloning
- Empirical evaluation
- Conclusion

Outline

- Problem
- Heap Cloning
- Empirical evaluation
- Conclusion

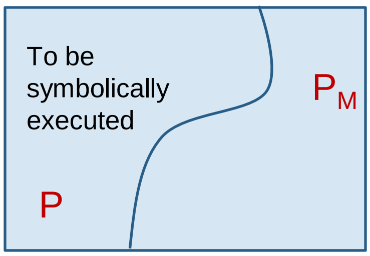
Symbolic Execution of Real-World Programs



Program

P’ not symbolically executed because:
1. P’ uses difficult-to-handle Java features such as native methods.
2. symbolic execution of P’ is *mostly* unnecessary.

Symbolic Execution with User-specified Models



Program

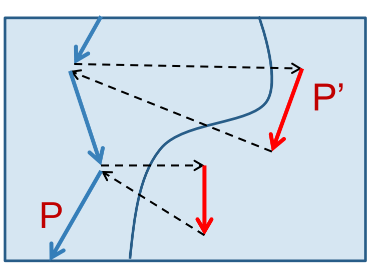
Replace P’ with user-specified models P_M

Impractical! Requires significant manual effort.

Dynamic Symbolic Execution

P: to be symbolically executed
P’: not to be symbolically executed

Symbolic execution
Concrete execution
Call/return



Program

No manual effort to write models

Example

```
class A {
   int f;
   A(int x) {
     this.f = x;
   }

  native void negate();

}
```

```
void negate() {
  this.f = -this.f;
}
```

Java version of native method negate

Example

```
class A {
   int f;
   A(int x) {
     this.f = x;
   }

  native void negate();

  void main() {
1   x = read();
2   m = new A(x);
3   if(m.f < 0)
4     m.negate();
5   if(m.f < 0)
6      error();
7   exit();
   }
}
```

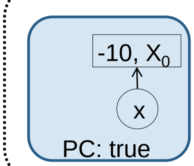
Task:
Perform dynamic symbolic execution along the path (1-2-3-4-5-7) that program takes for concrete input -10.

Requirement:
Native method `negate()` not to be symbolically executed

Dynamic Symbolic Execution

```
x = read();
m = new A(x);
if(m.f < 0)
  m.negate();
if(m.f < 0)
   error();
exit();
```

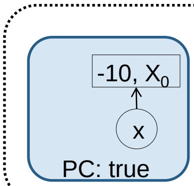
Continued on
next slide



state before: **m = new A(x)**

Dynamic Symbolic Execution

```
x = read();
m = new A(x);
if(m.f < 0)
  m.negate();
if(m.f < 0)
   error();
exit();
```



state before: **m = new A(x)**

Path Constraint (PC) of a path is a constraint of input values.

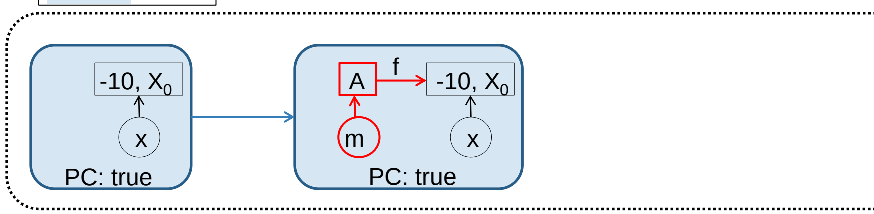
If PC is unsatisfiable, the path is infeasible.

If PC is satisfiable, any solution of PC is a program input that takes the corresponding program path.

Dynamic Symbolic Execution

```
x = read();
m = new A(x);
if(m.f < 0)
  m.negate();
if(m.f < 0)
   error();
exit();
```

Continued on
next slide



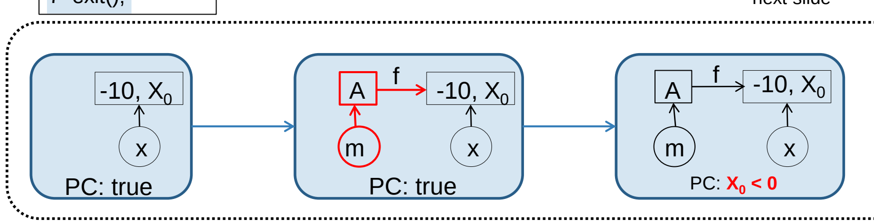
state before: **m = new A(x)** state before: **if(m.f < 0)**

Differences between two successive states are shown in **Red**.

Dynamic Symbolic Execution

```
x = read();
m = new A(x);
if(m.f < 0)
  m.negate();
if(m.f < 0)
   error();
exit();
```

Continued on
next slide



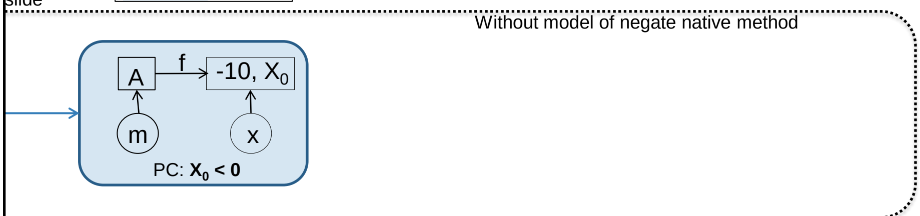
state before: **m = new A(x)** state before: **if(m.f < 0)** state before: **m.negate()**

Differences between two successive states are shown in **Red**.

Dynamic Symbolic Execution

```
x = read();
m = new A(x);
if(m.f < 0)
  m.negate();
if(m.f < 0)
   error();
exit();
```

Continued from prev. slide



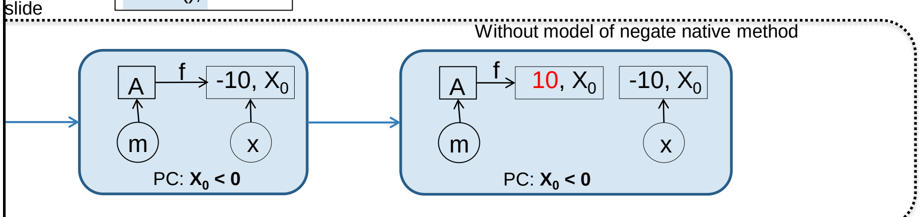
state before: **m.negate()**

Differences between two successive states are shown in **Red**.

Dynamic Symbolic Execution

```
x = read();
m = new A(x);
if(m.f < 0)
  m.negate();
if(m.f < 0)
   error();
exit();
```

Continued from prev. slide



state before: **m.negate()**

states before: **if(m.f < 0)**

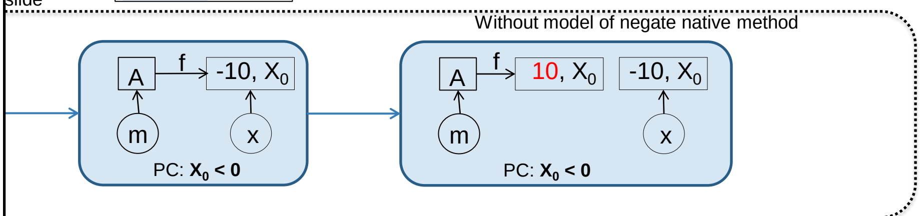
Differences between two successive states are shown in **Red**.

Dynamic Symbolic Execution

```
x = read();
m = new A(x);
if(m.f < 0)
  m.negate();
if(m.f < 0)
  error();
exit();
```

Continued from prev. slide

Concrete value of the field changed from -10 to 10. But, symbolic value X_0 did not change.



state before: **m.negate()**

states before: **if(m.f < 0)**

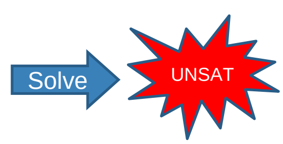
Differences between two successive states are shown in **Red**.

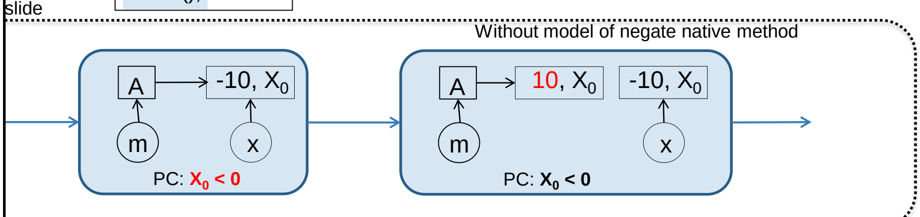
Dynamic Symbolic Execution

```
x = read();
m = new A(x);
if(m.f < 0)
  m.negate();
if(m.f < 0)
  error();
exit();
```

Continued from prev. slide

PC: $X_0 < 0$ **& X_0 >= 0**



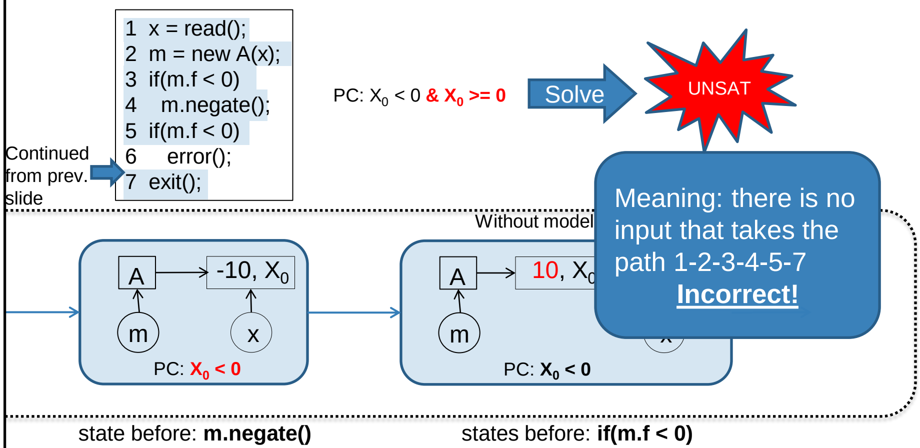


state before: **m.negate()**

states before: **if(m.f < 0)**

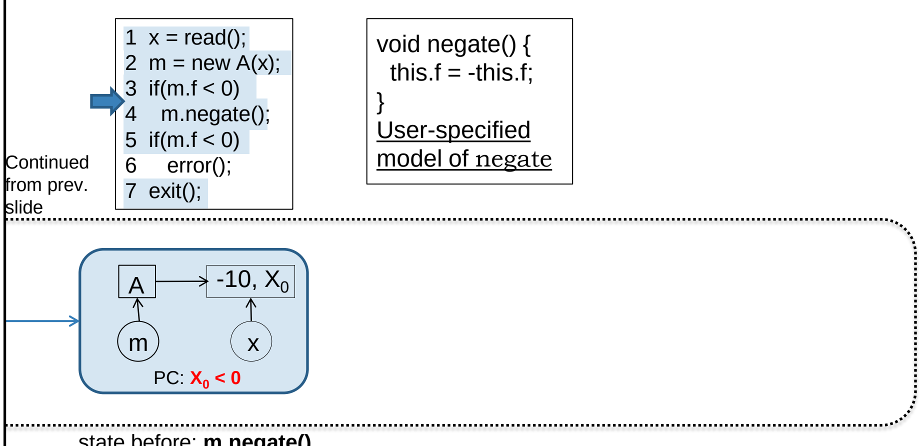
Differences between two successive states are shown in **Red**.

Dynamic Symbolic Execution



Differences between two successive states are shown in **Red**.

Dynamic Symbolic Execution with Models



Differences between two successive states are shown in **Red**.

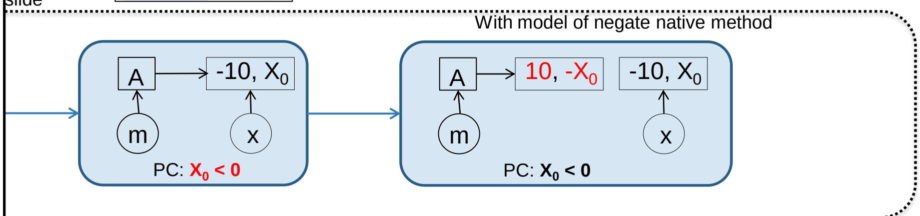
Dynamic Symbolic Execution with Models

```
x = read();
m = new A(x);
if(m.f < 0)
  m.negate();
if(m.f < 0)
  error();
exit();
```

```
void negate() {
  this.f = -this.f;
}
```

<u>User-specified model of negate</u>

Continued from prev. slide



state before: **m.negate()**

states before: **if(m.f < 0)**

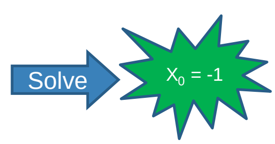
Differences between two successive states are shown in **Red**.

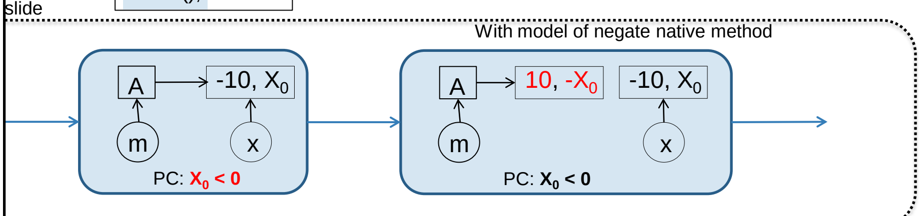
Dynamic Symbolic Execution with Models

```
x = read();
m = new A(x);
if(m.f < 0)
  m.negate();
if(m.f < 0)
  error();
exit();
```

Continued from prev. slide

PC: $X_0 < 0$ **& $-X_0$ >= 0**





state before: **m.negate()**

states before: **if(m.f < 0)**

Differences between two successive states are shown in **Red**.

But, Models Not Always Needed for Soundness!

```
void negate() {
  this.f = -this.f;
}
```

Original example

```
1   x = read();
2   m = new A(x);
3   if(m.f < 0)
4     m.negate();
5   if(m.f < 0)
6      error();
7   exit();
```

Modified example

```
1   x = read();
2   m = new A(x);
3   if(m.f < 0)
4     m.negate();
5   if(x < 0)
6      error();
7   exit();
```

But, Models Not Always Needed for Soundness!

Stmt at line 5 uses the value defined inside negate method.

Thus, negate introduces imprecision.

Thus, model for negate is needed.

```
void negate() {
  this.f = -this.f;
}
```

Stmt at line 5 does not use the value defined inside negate method.

Thus, negate does not introduce imprecision.

Thus, no need for model for negate.

```
4     m.negate();
5   if(m.f < 0)
6      error();
7   exit();
```

```
1
2
3
4     m.negate();
5   if(x < 0)
6      error();
7   exit();
```

Goal and Approach

Goal:
Identify where imprecision is introduced and report to user. User then provides models for only a subset methods in P' to eliminate imprecision.

Approach:
1. Transform the program using Heap Cloning.
2. Perform dynamic symbolic execution of the transformed program without any models
3. Detect and report where imprecision might be introduced.

Outline

- Problem
- Heap Cloning
- Empirical evaluation
- Conclusion

Heap Cloning

Original Program P $\xrightarrow{\text{Heap Cloning Transformation}}$ Transformed Program P_T

1. Symbolic execution of P_T generates same results as symbolic execution of P.
2. However, if imprecision is introduced during symbolic execution of P_T, it can be detected and reported to user.

Heap Cloning

For every class A, Heap Cloning generates HC.A.

```
class HC.java.lang.Object {
   …
}

class HC.A extends
                                 : {





}
```

Heap Cloning

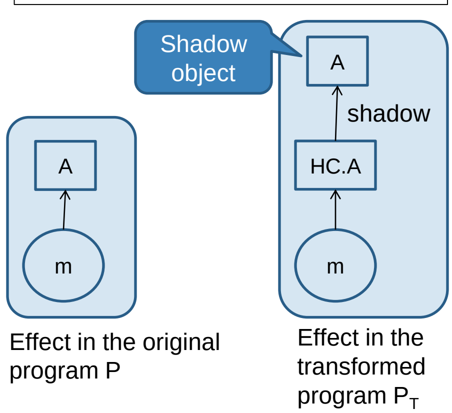
For every class A, Heap Cloning generates HC.A.

For each field (method) of A, HC.A has a corresponding field (method).

```
class HC.java.lang.Object {
   …
}

class HC.A extends
                                    : {
   int f;
   …

  void main() {

    …
  }
}
```

Heap Cloning

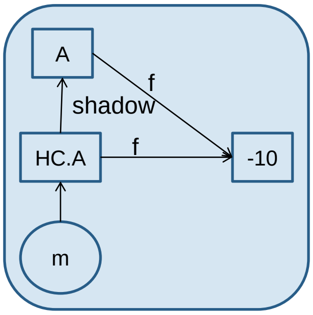
Effects of the statement **m = new A** on program state



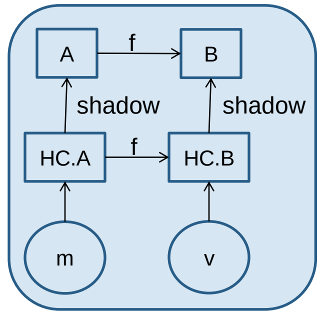
For every object of class A in P, P_T allocates two objects: one of class A, other of class HC.A

Heap Cloning

Effects of the statement **m.f = -10** in P_T

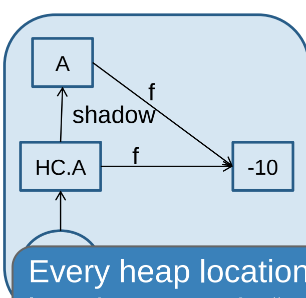


Effects of the statement **m.f = v** (v is a reference type variable) in P_T

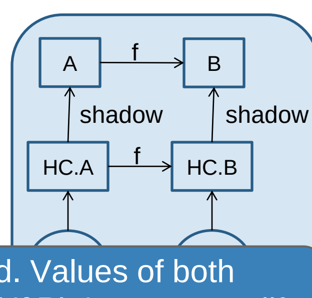


Heap Cloning

Effects of the statement **m.f = -10** in P_T



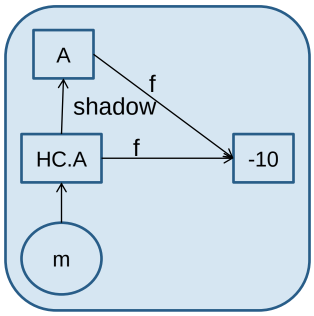
Effects of the statement **m.f = v** (v is a reference type variable) in P_T



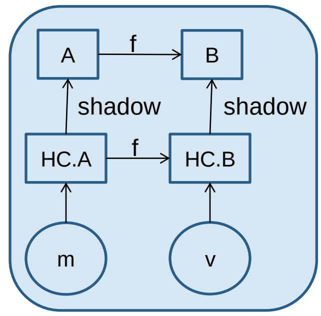
Every heap location is cloned. Values of both locations remain "consistent" if P' does not modify value of shadow location

Heap Cloning

Effects of the statement **m.f = -10** in P_T

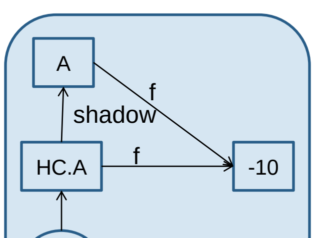


Effects of the statement **m.f = v** (v is a reference type variable) in P_T

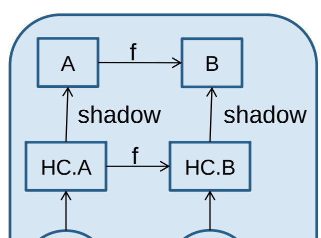


Heap Cloning

Effects of the statement **m.f = -10** in P_T



Effects of the statement **m.f = v** (v is a reference type variable) in P_T

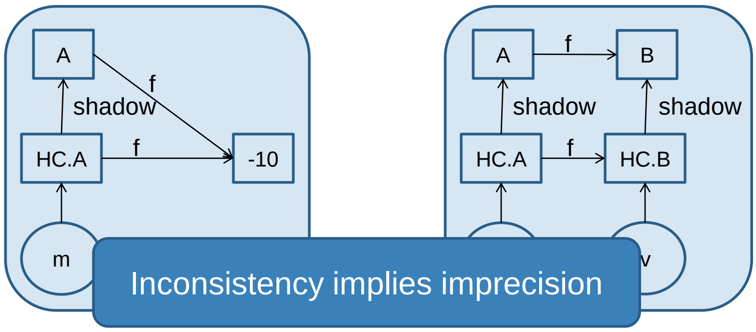


Every heap location is cloned. Values of both locations remain “consistent” if P’ does not modify value of shadow location

Heap Cloning

Effects of the statement **m.f = -10** in P_T

Effects of the statement **m.f = v** (v is a reference type variable) in P_T

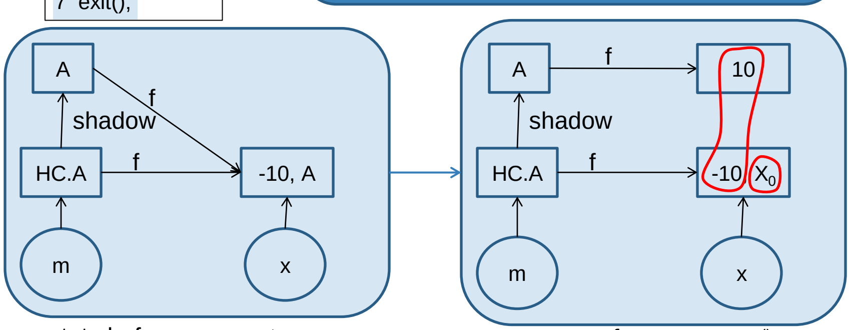


Heap Cloning

```
1  x = read();
2  m = new A(x);
3  if(m.f < 0)
4    m.negate();
5  if(m.f < 0)
6    error();
7  exit();
```

Two conditions are satisfied and indicate potential imprecision.

1. Concrete values are different.
2. m.f has a symbolic value



state before m.negate

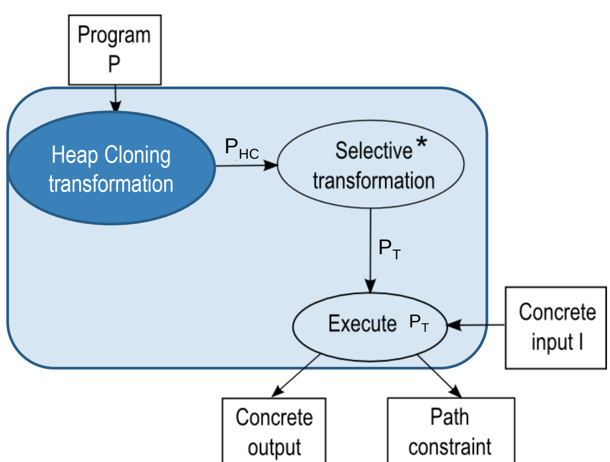
state after m.negate()

Outline

- Problem
- Heap Cloning
- Empirical evaluation
- Conclusion

Empirical Study

Implementation: Cinger



* our prior work from TACAS'07

Empirical Study

Subjects

Subject	Covered lines of code		Avg. no. of conjuncts in path constraints
	Application	Library	
NanoXML	1,230	14,604	19,582
JLex	6,566	13,702	65,068
Sat4J-Dimacs	3,908	17,195	60,351
Sat4J-CSP	4,125	39,617	629,078
BCEL	2,321	12,659	34,161
Lucene	20,821	56,622	47,248

Empirical Study

Goal

Evaluate the reduction in number of methods for which user must write models, when P' is the set of all native methods that the program uses

Method

1. Estimate the number of native methods that may cause side effects
2. Count the number of native methods that introduce imprecision when symbolic execution is performed on Heap-Cloning transformed program.
3. Compare the two

Empirical Study

Results

Subject	Number of unique native methods that were executed and that took reference-type parameters
NanoXML	4
JLex	6
Sat4J-Dimacs	0
Sat4J-CSP	9
BCEL	
Lucene	

Over all subjects, **only one** native method System.arraycopy introduced imprecision, and thus needed model.

Summary and Future Work

1. Heap Cloning worked well for selected subjects. In future, use more and diverse set of subjects.
2. Heap Cloning reduced manual effort when P' consisted of all native methods. In future, treat framework (e.g., Google's Android framework) code as part of P'.
3. Heap Cloning could identify where models are necessary. In future, check for correctness of user-specified models

Contributions

1. Heap Cloning technique that
   enables correct dynamic symbolic execution, but requires minimal manual effort to specify models
2. Implementation of Cinger system that
   is capable of correct dynamic symbolic execution of real-world Java programs
3. Empirical studies that
   shows Heap Cloning identifies a small number of methods that cause imprecision and need to be manually modeled