Formal Methods: Model Checking and Other Applications

Orna Grumberg
Technion, Israel

Marktoberdorf 2017

Outline

- Model checking of finite-state systems
- Assisting in program development
  - Program repair
  - Program differencing

Modular Demand-Driven Analysis of Semantic Difference for Program Versions

Anna Trostanetski, Orna Grumberg,
Daniel Kroening

(SAS 2017)

Program versions

Programs often change and evolve, raising the following interesting questions:

- Did the new version introduced new bugs or security vulnerabilities?
- Did the new version remove bugs or security vulnerabilities?
- More generally, how does the behavior of the program change?

Differences between program versions can be exploited for:

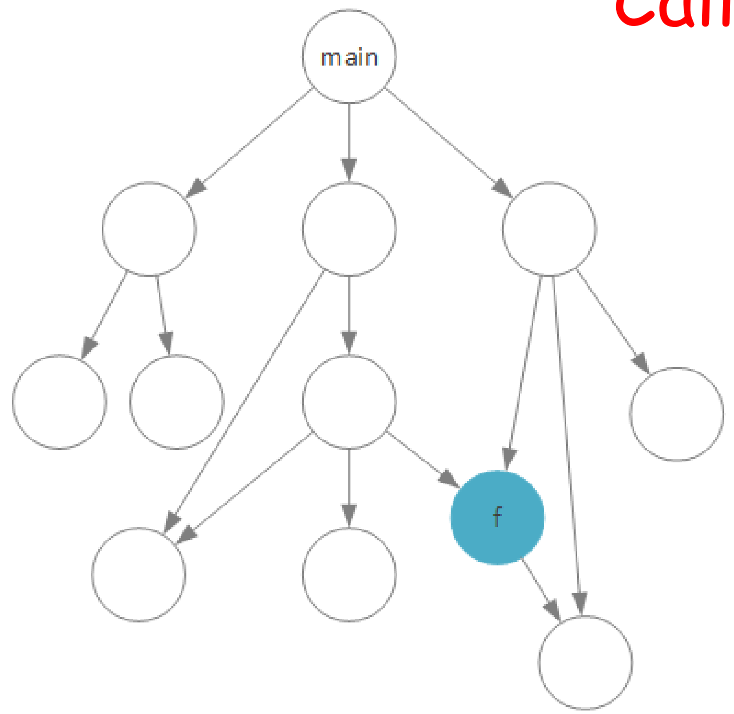
- Regression testing of new version w.r.t. old version, used as "golden model"
- Producing zero-day attacks on old version
- characterizing changes in the program's functionality

How Programs Change

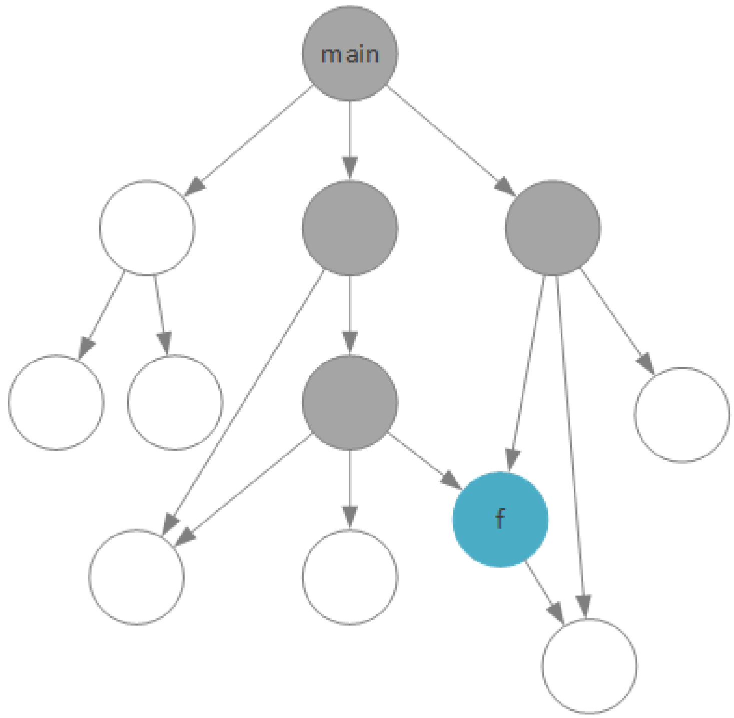
Changes are small, programs are large

Can our work be **O(change)** instead of **O(program)**?

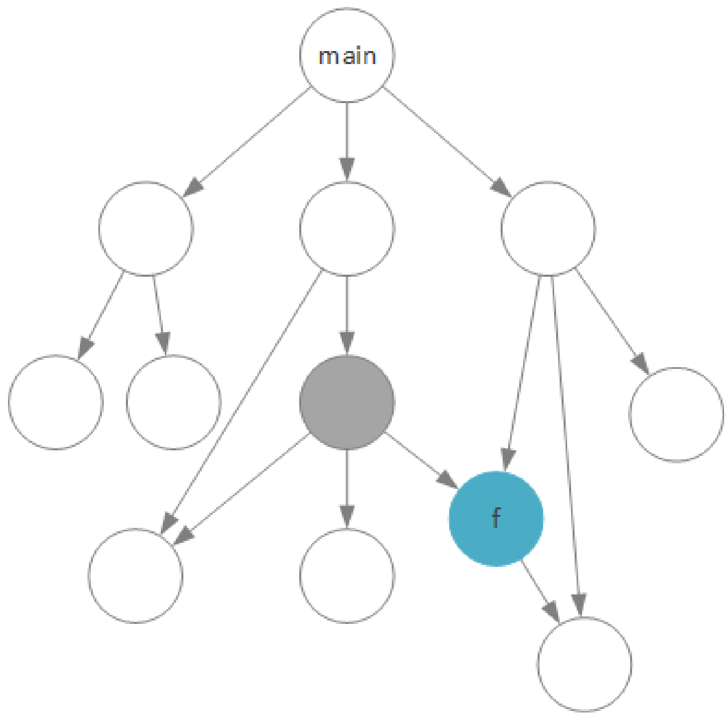
Call graph



Which procedures could be affected



Which procedures are affected



Main ideas (1)

- Modular analysis applied to one pair of procedures at a time
  - No inlining

- Only affected procedures are analyzed

- Over- and under-approximation of difference between procedures are computed

Main ideas (2)

- Procedures need not be fully analyzed:
  - Unanalyzed parts are abstraction using uninterpreted functions
  - Refinement is applied upon demand
- Anytime analysis:
  - Not necessarily terminates
  - Its partial results are meaningful
  - The longer it runs, the more precise its results are

Program representation

- Program is represented by a call graph
- Every procedure is represented by a Control Flow Graph (CFG)
- We are also given a matching function between procedures in the old and new versions

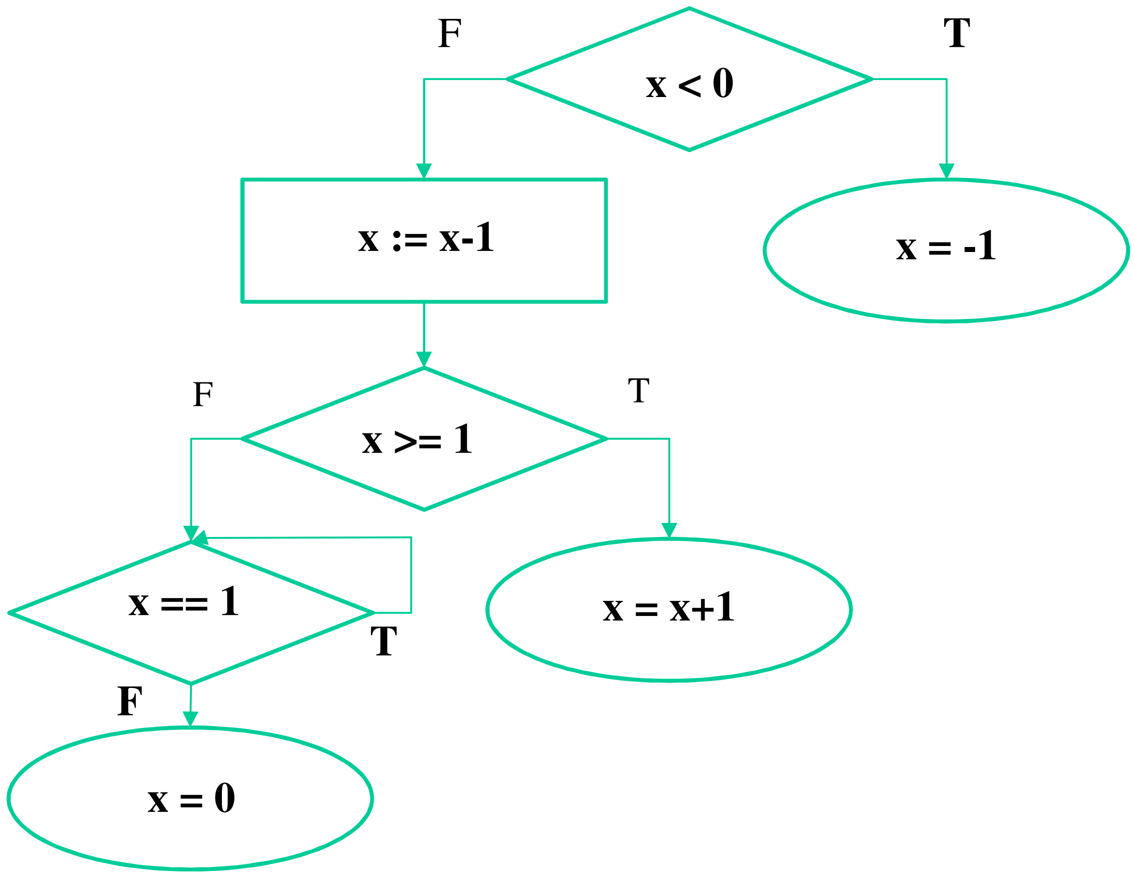
- A call graph is a directed graph:
  - Nodes represent procedures
  - It contains edge $p \rightarrow q$ if procedure p includes a call for procedure q

- A control flow graph (CFG) is a directed graph:
  - Nodes represent program instructions (assignments, conditions and procedure calls)
  - Edges represent possible flow of control

Example

```
void p(int& x) {

    if (x < 0) {
        x= -1;
        return;
    }
    x--;
    if (x >= 1) {
        x= x+1;
        return;
    } else
        while ( x == 1);
    x=0;
}
```



Path characterization

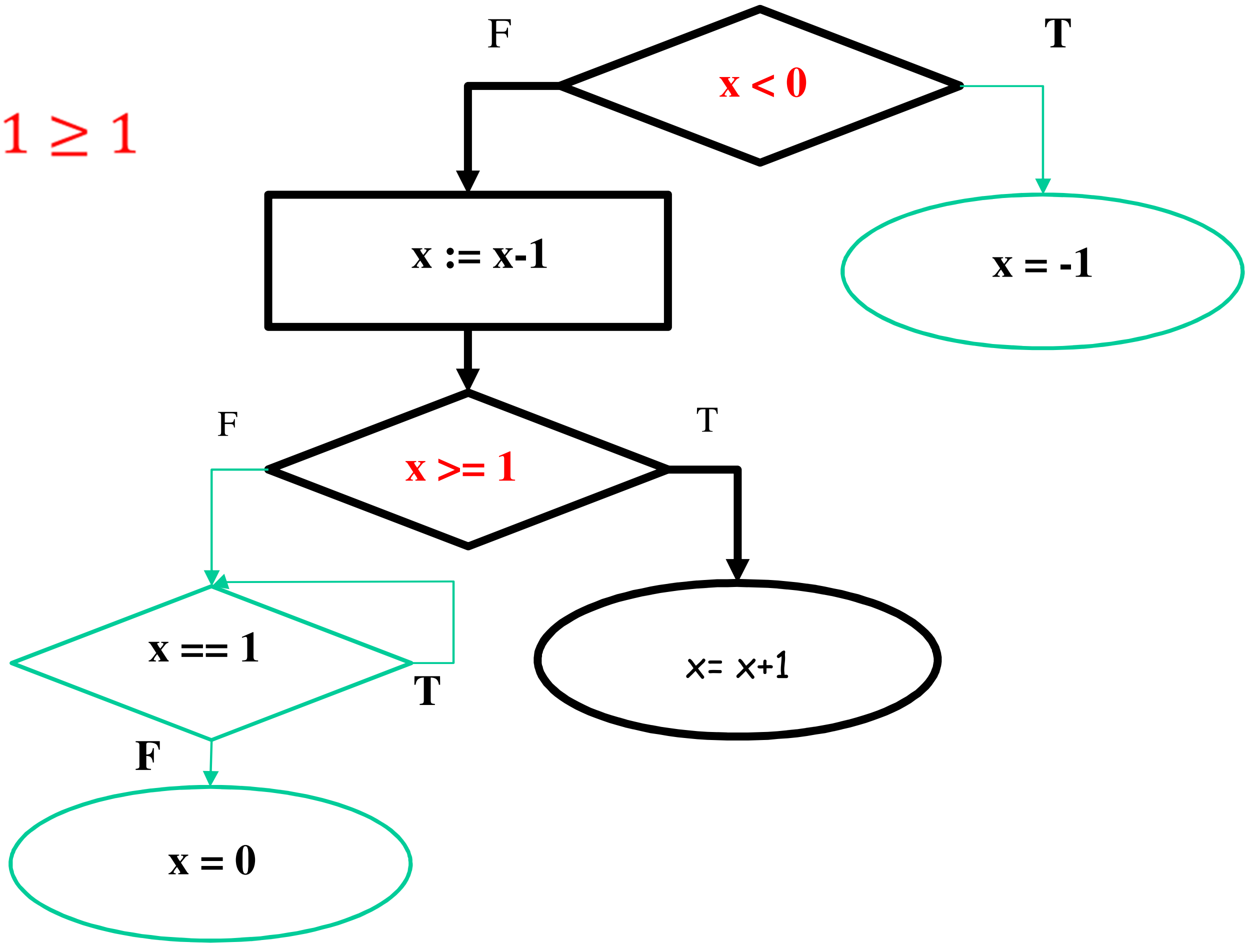
- For a finite path π in CFG from entry node to exit node:
  - The **reachability condition** R_π is a First Order Logic Formula, which guarantees that control traverses π
  - The **state transformation** T_π is an n-tuple of expressions over program variables, describing the transformation on the variables' values along π

Both given in terms of variables at the entry node of π

- End of lecture 3

Example

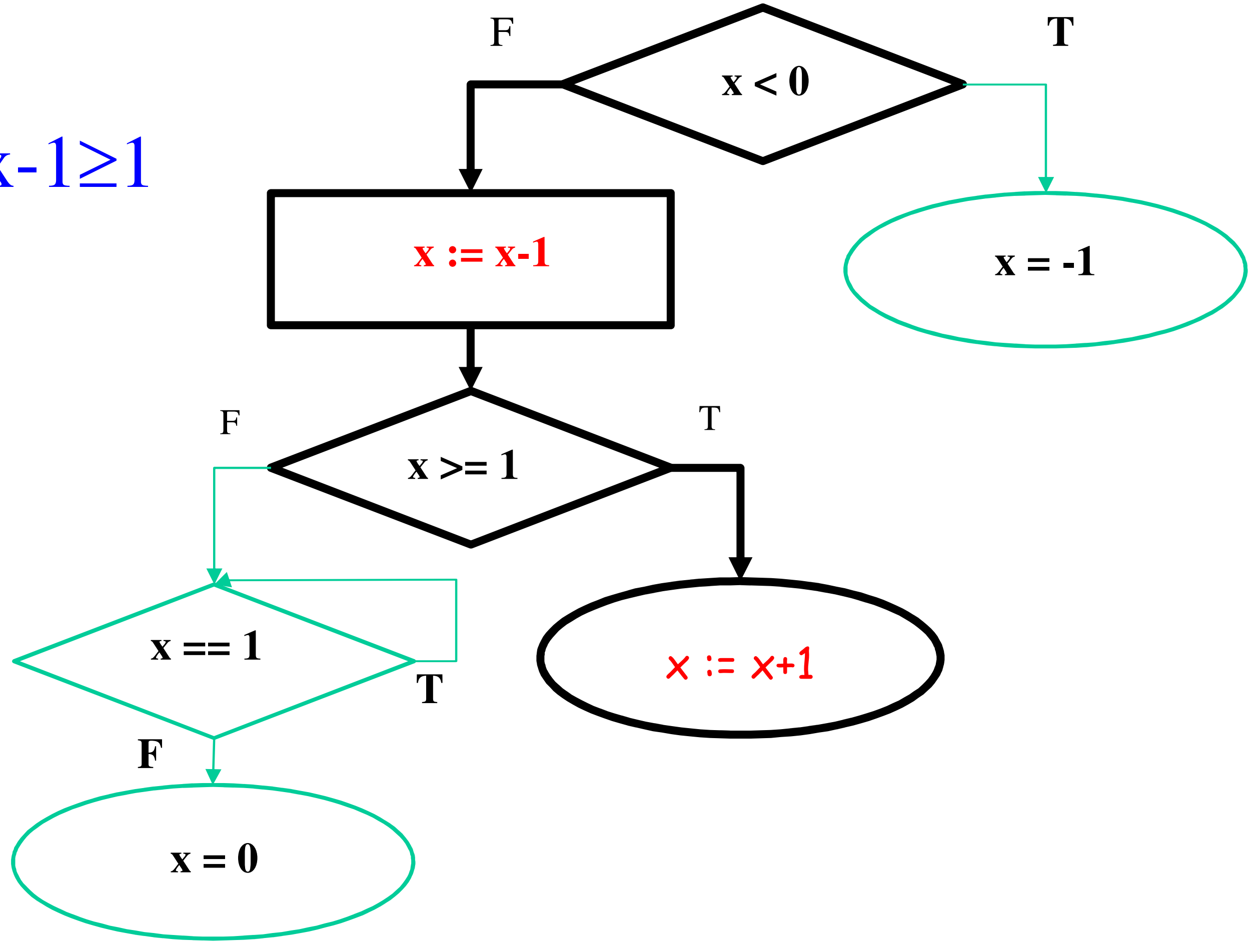
$$R_\pi(x) = x \geq 0 \wedge x - 1 \geq 1$$
$$\equiv x \geq 2$$



Example

$R_\pi(x) = x \geq 0 \wedge x\text{-}1 \geq 1$

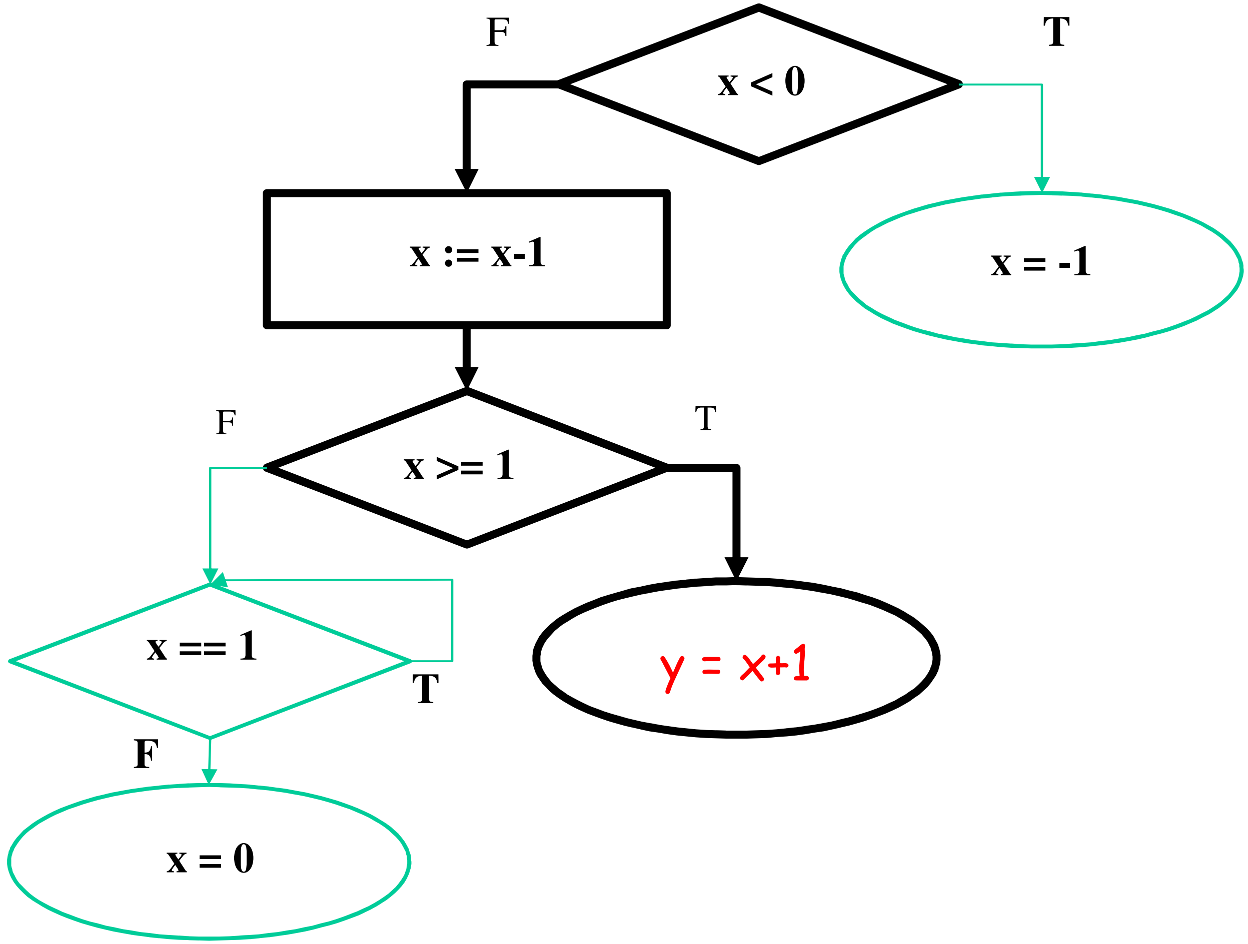
$T_\pi(x) = (x)$



Example

$R_\pi(x,y) =$
$x \geq 0 \wedge x\text{-}1 \geq 1$

$T_\pi(x,y) = (x\text{-}1,x)$



Symbolic execution

- Input variables are given symbolic values
- Every execution path is explored individually (in some heuristic order)
- On every branch, a feasibility check is performed with a constraint solver

Symbolic execution

The symbolic execution we use consists of:

For path π in procedure p

- $\mathbf{R_{\pi}(V_p)}$
- $\mathbf{T_{\pi}(V_p)}$

where $\mathbf{V_p}$ denotes the input variables (the parameters) for procedure **p**

Computing symbolic execution

Given a finite path $\pi = l_1, \dots, l_n$,

R_π^i and T_π^i are the path condition and state transformation for path $l_1, \dots, l_{i-1}$, respectively.

$$R_\pi = R_\pi^{n+1}$$
$$T_\pi = T_\pi^{n+1}$$

Computing symbolic execution

Iterative computation:

- Initialization:
  - For every $x \in V_p$, $T_\pi^1[x] = x$
  - $R_\pi^1 = true$
- Assume R_π^i, T_π^i are already defined. R_π^{i+1}, T_π^{i+1} are defined according to the instruction at node i:

Computing symbolic execution

Instruction	R	T
Assignment $x := e$	$R_\pi^{i+1} = R_\pi^i$	$\forall y \neq x \;\; T_\pi^{i+1}[y] := T_\pi^i[y]$ $T_\pi^{i+1}[x] := e[V_p \leftarrow T_\pi^i]$
Test B	$R_\pi^{i+1} = R_\pi^i \wedge \tilde{B}$	$\forall x \; T_\pi^{i+1}[x] := T_\pi^i[x]$
Procedure call $g(Y)$	Inlined	

Our goal:

- Compute procedure summary for individual procedures
  - using path summaries (R_π, T_π)

- Compute difference summary for matching pairs of procedures

Procedure summary

- Procedure summary of procedure p is

$$Sum_p \subseteq \{ (R_\pi, T_\pi) \mid \pi \text{ is a finite path in } p\}$$

- The full set of path summaries often cannot be computed
  - And might not be needed

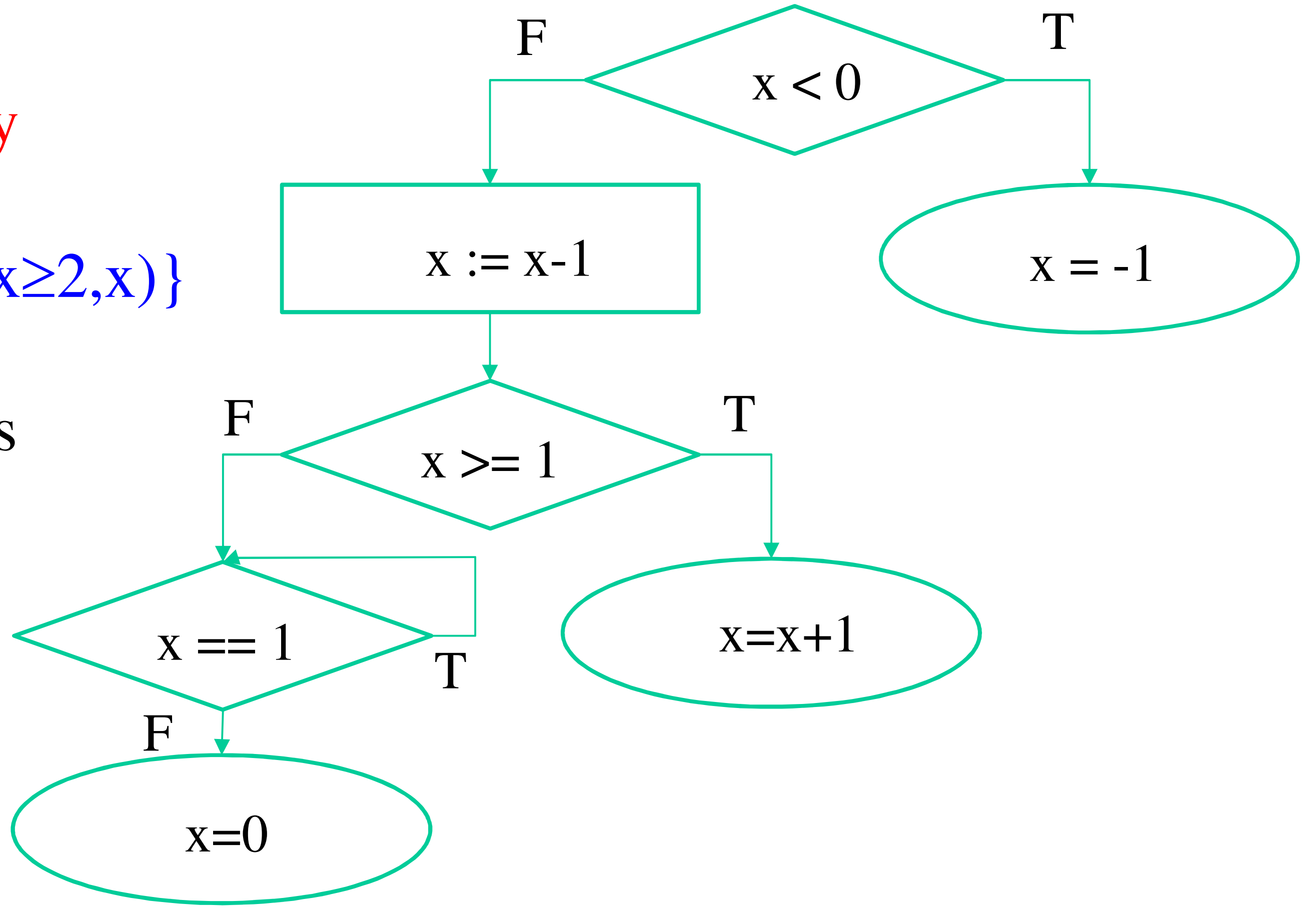
Example

A possible summary for procedure p:

$sum_p = \{(x<0,-1), (x\geq 2,x)\}$

Its uncovered part is

$x\geq 0 \wedge x<2$



Another goal:

- To compute path summaries without in-lining called procedures

- We suggest modular symbolic execution

Modular symbolic execution

- Path π of procedure p includes call $g(Y)$ at location l_i

- $sum_g = \{ (r_1,t_1),\ldots, (r_1,t_1)\}$ previously computed

- Instead of in-lining g we compute:

Modular symbolic execution

- $R_\pi^{i+1} = R_\pi^{i} \wedge \bigvee_{j=1,..n} r_j$

- $T_\pi^{i+1} = \mathbf{ITE}(r_1, t_1, \ldots, \mathbf{ITE}(r_n, t_n, error)..)$

Modular Symbolic Execution

$$\boldsymbol{R_{\pi}^{i+1}} = R_{\pi}^{i} \wedge \bigvee_{j=1}^{n} r_j \left[V_g^v \leftarrow T_{\pi}^{i}(Y)\right]$$

$$\boldsymbol{T_{\pi}^{i+1}} = \ \boldsymbol{ITE}(\ \boldsymbol{r_1}\left[V_g^v \leftarrow T_{\pi}^{i}(Y)\right], \boldsymbol{t_1}\left[V_g^v \leftarrow T_{\pi}^{i}(Y)\right], \ldots,$$

$$\boldsymbol{ITE}(\ \boldsymbol{r_n}\left[V_g^v \leftarrow T_{\pi}^{i}(Y)\right], \boldsymbol{t_n}\left[V_g^v \leftarrow T_{\pi}^{i}(Y)\right], \text{error})$$

Can we do better?

- Use abstraction for the un-analyzed (uncovered) parts
- Later check if these parts are needed at all for the analysis of the full program (procedure main)
  - If needed - refine

Abstraction

- Unanalyzed parts of a procedure is replaced by uninterpreted functions

- For matched procedures g_1,g_2 we have
  - A common uninterpreted function $UF_{g1,g2}$
  - Individual uninterpreted functions UF_{g1} and UF_{g2}

Abstract modular symbolic execution

For call $g_1(Y)$ with

$sum_{g1} = \{ (r_1,t_1),\ldots, (r_n,t_n)\}$:

$R_\pi^{i+1} = R_\pi^{i}$

$T_\pi^{i+1} = ITE(r_1, t_1,\ldots ITE(r_n, t_n,$

$ITE(computed\_unchanged, UF_{g1,g2}, UF_{g1}))$

- For $g_2(Y)$ we use sum_{g2} and UF_{g2}

Full Difference Summary

Difference for a pair of procedures p_1, p_2 is a triplet:

– **changed:** is the set of initial states for which both procedures terminate with different final states.

– **termination_changed**: is the set of initial states for which exactly one procedure terminates.

– **unchanged**: is the set of initial states for which both procedures either terminate with the same final states, or both do not terminate.

$$\boldsymbol{changed} \cup \boldsymbol{temination\_changed} \cup \boldsymbol{unchanged}$$
$$= \boldsymbol{input\ space}$$

Example

```
void p1(int& x) {

    if (x < 0)
        x = -1;
    x--;
    if (x >= 1)
        x=x+1;
        return;
    else
        while ( x ==1);
     x=0;
}
```

```
void p2(int& x) {

    if (x < 0)
        x = -1;
    x--;
    if (x > 2)
        x=x+1;
        return;
    else
        while ( x == 1);
     x=0;
}
```

Example

The full difference summary is:

$$\boldsymbol{changed} \coloneqq \{3\}$$

$$\textbf{terminate\_change}\boldsymbol{d} \coloneqq \{2\}$$

$$\textbf{unchange}\boldsymbol{d} \coloneqq \{c \mid (c < 2) \vee (c > 3)\}$$

Difference Summary - computation

Full difference summary is incomputable!

Compute under-approximations of changed and unchanged, ignoring terminate_change:

- $\boldsymbol{computed\_changed} \subseteq changed$
- $\boldsymbol{computed\_unchanged} \subseteq unchanged$

Difference Summary - computation

Difference Summary gives us:

- An under-approximation of the difference:

  computed_changed

- An over-approximation of the difference:

  $$\boldsymbol{may\_change} = \neg computed\_unchanged$$

Computing difference summary

For each (r_1,t_1) in p_1, (r_2,t_2) in p_2

- diffCond := $r_1 \wedge r_2 \wedge t_1 \neq t_2$
- If diffCond is SAT, add it to computed_changed

- eqCond := $r_1 \wedge r_2 \wedge t_1 = t_2$
- If eqCond is SAT, add it to computed_unchanged

Refinement

- Since we are using uninterpreted functions, the discovered difference may not be feasible:

```
void p1(int& x) {
   if (x == 5) {
        abs1(x);
        if (x==0)
            x = 1;
   }
}
```

abs1=abs2=abs →

```
void p2(int& x) {
   if (x == 5) {
        abs2(x);
        if (x==0)
            x = -1;
   }
}
```

- The following formula will be added to $computed\_changed_{p1,p2}$ (if SAT)

$$x=5 \wedge x' = UF_{abs1,abs2}(x) \wedge x'=0 \wedge 1\neq-1$$

- In order to check satisfiability, symbolic execution is applied to abs
  - Not necessarily on all paths

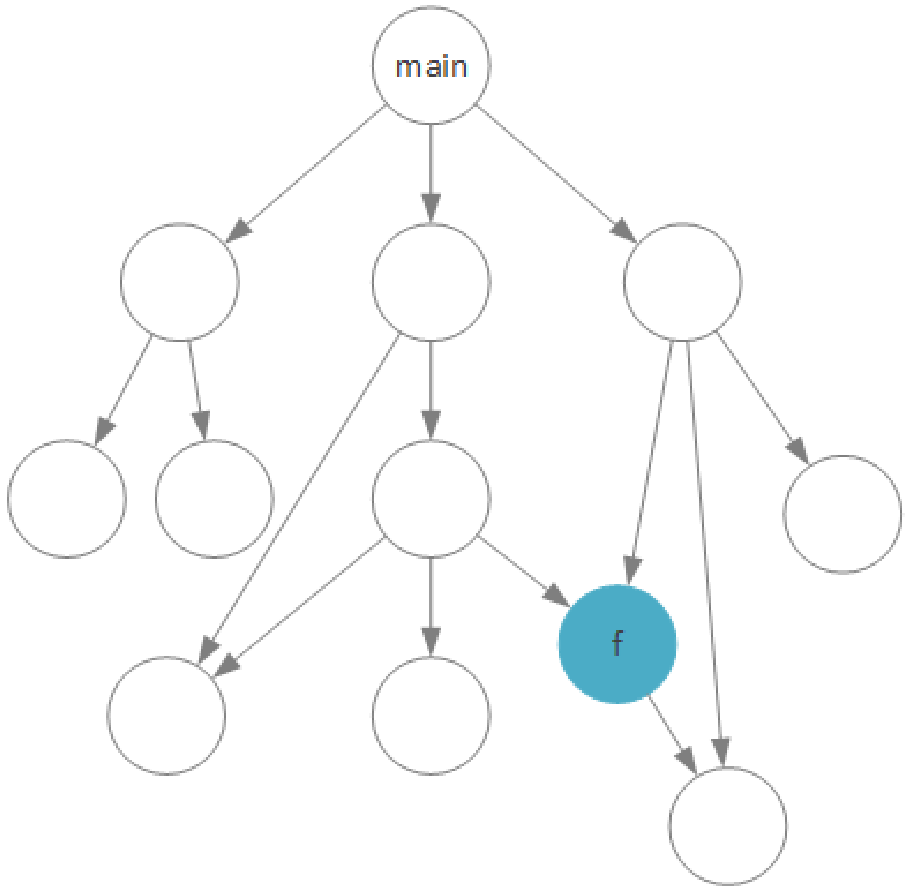
Refinement

- We run symbolic execution on abs on the path traversed by input 5.
- Now the difference summary is refined and we can check satisfiability again of

$$x = 5 \wedge x' = \left(\boldsymbol{x > 0}?\ \boldsymbol{x}: \boldsymbol{UF_{abs_1,abs_2}(x)}\right) \wedge x' = 0,$$

which is now unsatisfiable meaning there is no difference

Overall Algorithm



Experimental Results - Equivalent Benchmarks

Benchmark	MDDiff	MDDiffRef	RVT	SymDiff
Const	0.545s	0.541s	4.06s	14.562s
Add	0.213s	0.2s	3.85s	14.549s
Sub	0.258s	0.308s	5.01s	F
Comp	0.841s	0.539s	5.19s	F
LoopSub	0.847s	1.179s	F	F
UnchLoop	F	2.838s	F	F
LoopMult2	1.666s	1.689s	F	F
LoopMult5	F	3.88s	F	F
LoopMult10	F	9.543s	F	F
LoopMult15	F	21.55s	F	F
LoopMult20	F	49.031s	F	F
LoopUnrch2	0.9s	0.941s	F	F
LoopUnrch5	1.131s	1.126s	F	F
LoopUnrch10	1.147s	1.168s	F	F
LoopUnrch15	1.132s	1.191s	F	F
LoopUnrch20	1.157s	1.215s	F	F

LoopMult Benchmark

```
void foo1(int a, int b) {
   int c=0;
   for (int i=1; i <= b;
i++)
      c+=a;

   return c;
}
```

```
void foo2(int a, int b) {
   int c=0;
   for (int i=1; i <= a;
i++)
      c+=b;

   return c;
}
```

LoopMult Benchmark

LoopMult2

```
int main(int x) {
    return
foo(2,2);
}
```

LoopMult5

```
int main(int x) {
    if (x>=5 &&
x<7) {
        return
foo(x,5);
    }
}
```

LoopUnrch Benchmark

```
void foo1(int a, int b)
{
   int c=0;
   if (a<0) {
      for (int i=1; i <=
b; i++)
         c+=a;
   }
   return c;
}
```

```
void foo1(int a, int b)
{
   int c=0;
   if (a<0) {
      for (int i=1; i <=
a; i++)
         c+=b;
   }
   return c;
}
```

Experimental Results - Non Equivalent Benchmarks

Benchmark	MDDiff	MDDiffRef
LoopSub	1.187s	2.426s
UnchLoop	F	8.053s
LoopMult2	3.01s	3.451s
LoopMult5	F	5.914s
LoopMult10	F	10.614s
LoopMult15	F	14.024s
LoopMult20	F	25.795s
LoopUnrch2	2.157s	2.338s
LoopUnrch5	2.609s	3.216s
LoopUnrch10	2.658s	3.481s
LoopUnrch15	2.835s	3.446s
LoopUnrch20	3.185s	3.342s

Summary

We present a differential analysis method that is:

- Modular (analyzes each procedure independently of its current use)
- Incremental
- Computes over- and under-approximation of inputs that produce different behavior
- Introduces abstraction in the form of uninterpreted functions, and allows refinement upon demand

Thank you