MODULAR DEMAND-DRIVEN ANALYSIS OF SEMANTIC DIFFERENCE FOR PROGRAM VERSIONS

Orna Grumberg
Technion

Joint work with:
Anna Trostanetski, Daniel Kroening [SAS 2017]

Helmut Veith Memorial Workshop 2018

PROGRAM VERSIONS

Programs change and evolve, raising the following interesting questions:

- Did the new version introduce new bugs or security vulnerabilities?
- Did the new version introduce the desired feature?
- 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

WHAT IS A DIFFERENCE IN BEHAVIOR?

```
void p1(int& x) {
    if (x < 0)
        x = -1;
        return;
    x--;
    if (x >= 1)
        x=x+1;
        return;
    else
        while ( x ==1);
     x=0;
}
```



```
void p2(int& x) {
    if (x < 0)
        x = -1;
        return;
    x--;
    if (x > 2)
        x=x+1;
        return;
    else
        while ( x == 1);
     x=0;
}
```

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.**

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.**

violate **Partial Equivalence**

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**.

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**.

violate **Mutual Termination**

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.

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.

Full Equivalence

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.

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

EXAMPLE

The difference summary is:

$$changed := \{3\}$$

$$termination\_changed := \{2\}$$

$$unchanged := \{c : (c < 2) \vee (c > 3)\}$$

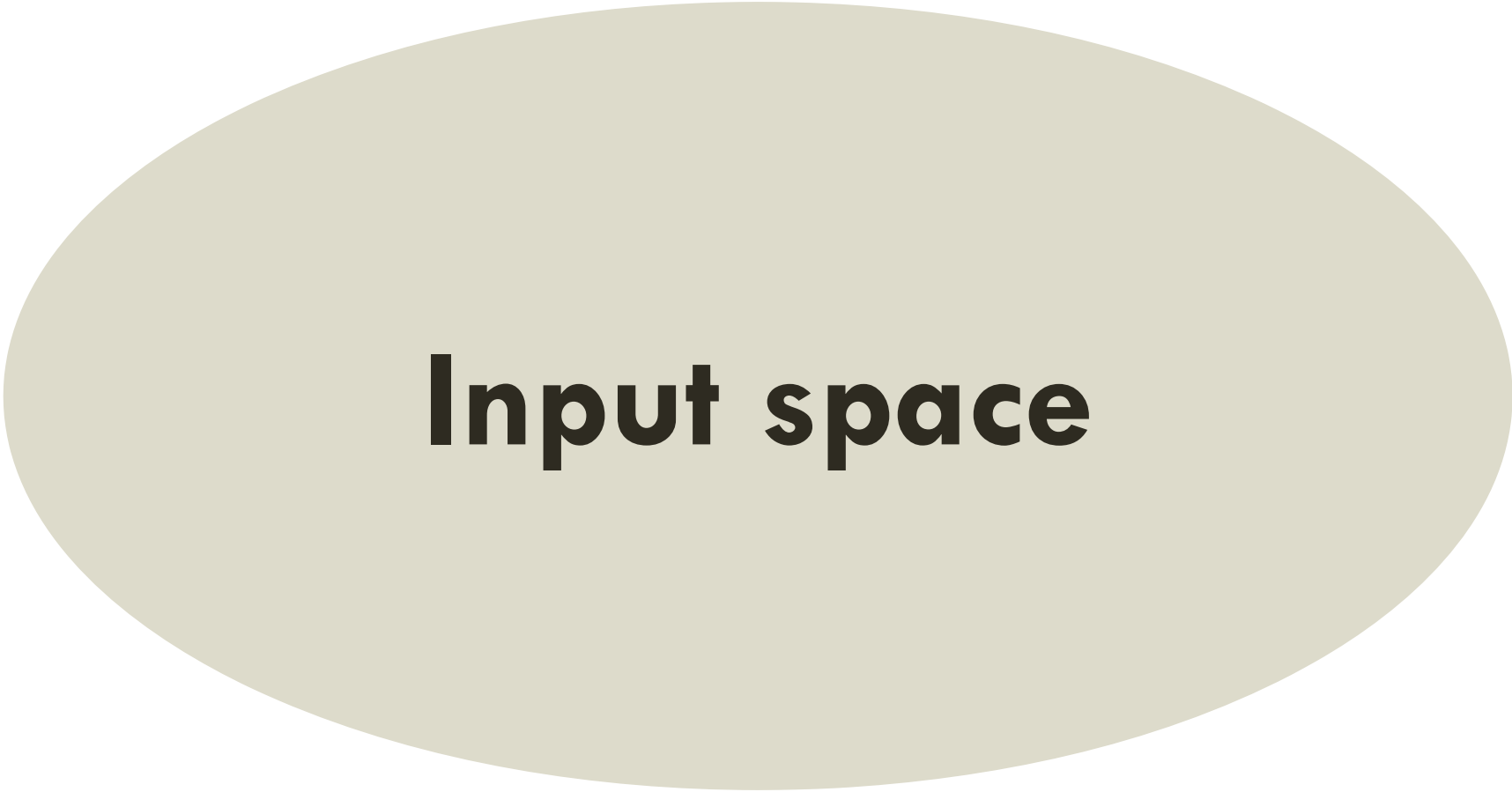
```
void p1(int& x) {
    if (x < 0)
        x = -1;
        return;
    x--;
    if (x >= 1)
        x=x+1;
        return;
    else
        while ( x ==1);
     x=0;
}
```

```
void p2(int& x) {
    if (x < 0)
        x = -1;
        return;
    x--;
    if (x > 2)
        x=x+1;
        return;
    else
        while ( x == 1);
     x=0;
}
```

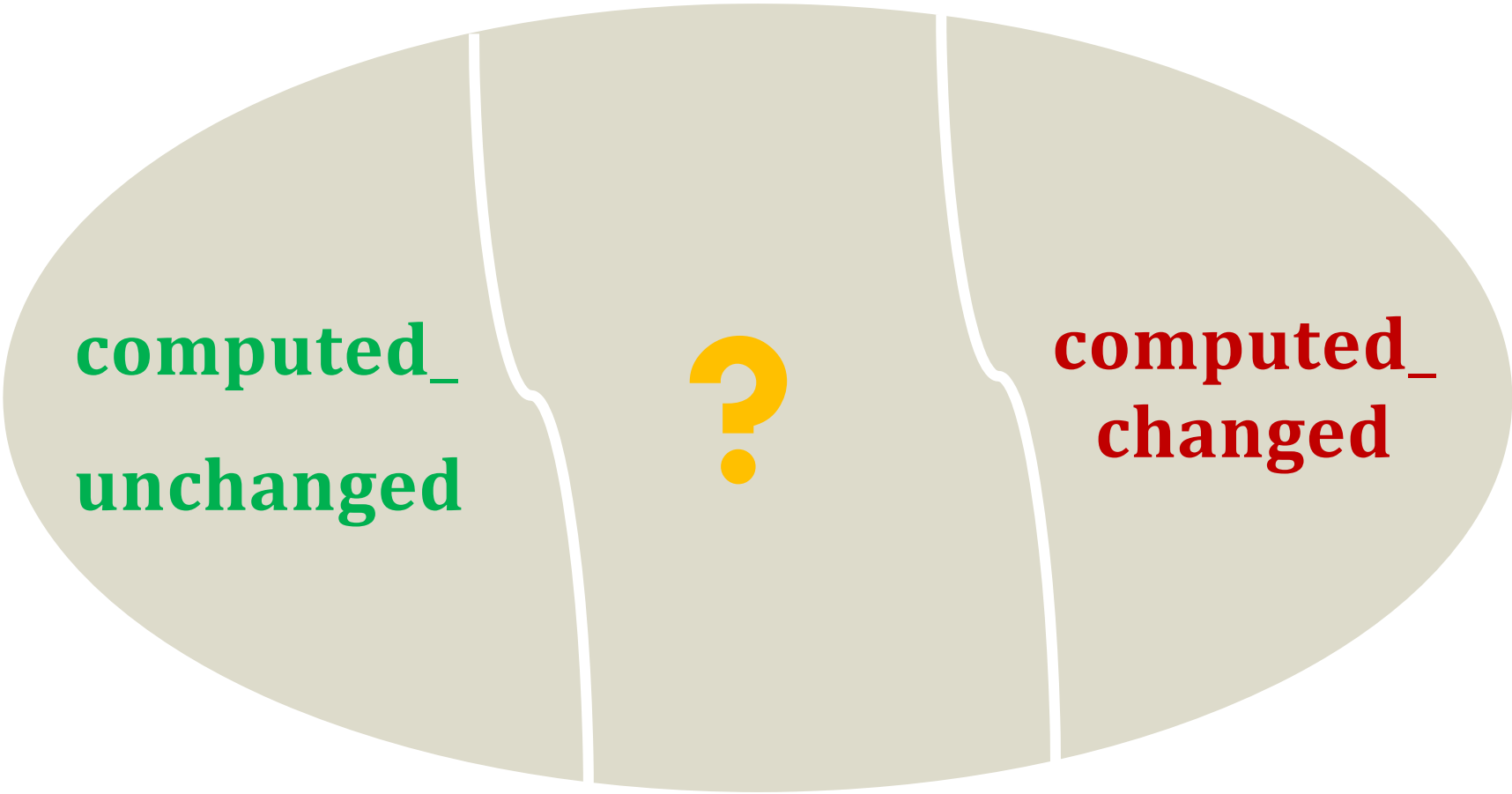
DIFFERENCE SUMMARY - COMPUTATION

- The full difference summary is incomputable
- We compute under-approximations of changed and unchanged, ignoring termination_changed:
  - A set $\boldsymbol{computed\_changed} \subseteq changed$
  - A set $\boldsymbol{computed\_unchanged} \subseteq unchanged$

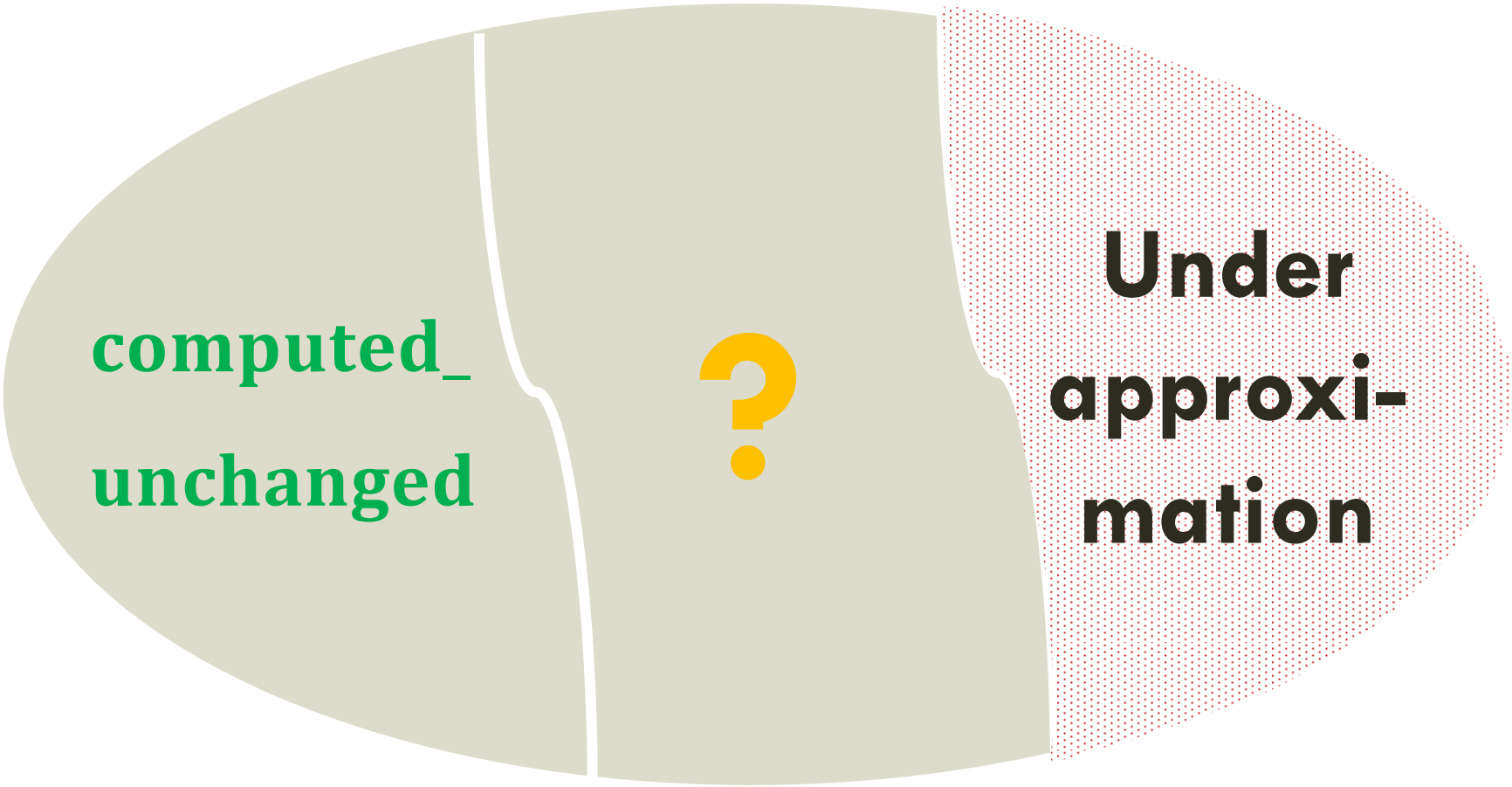
DIFFERENCE SUMMARY - COMPUTATION



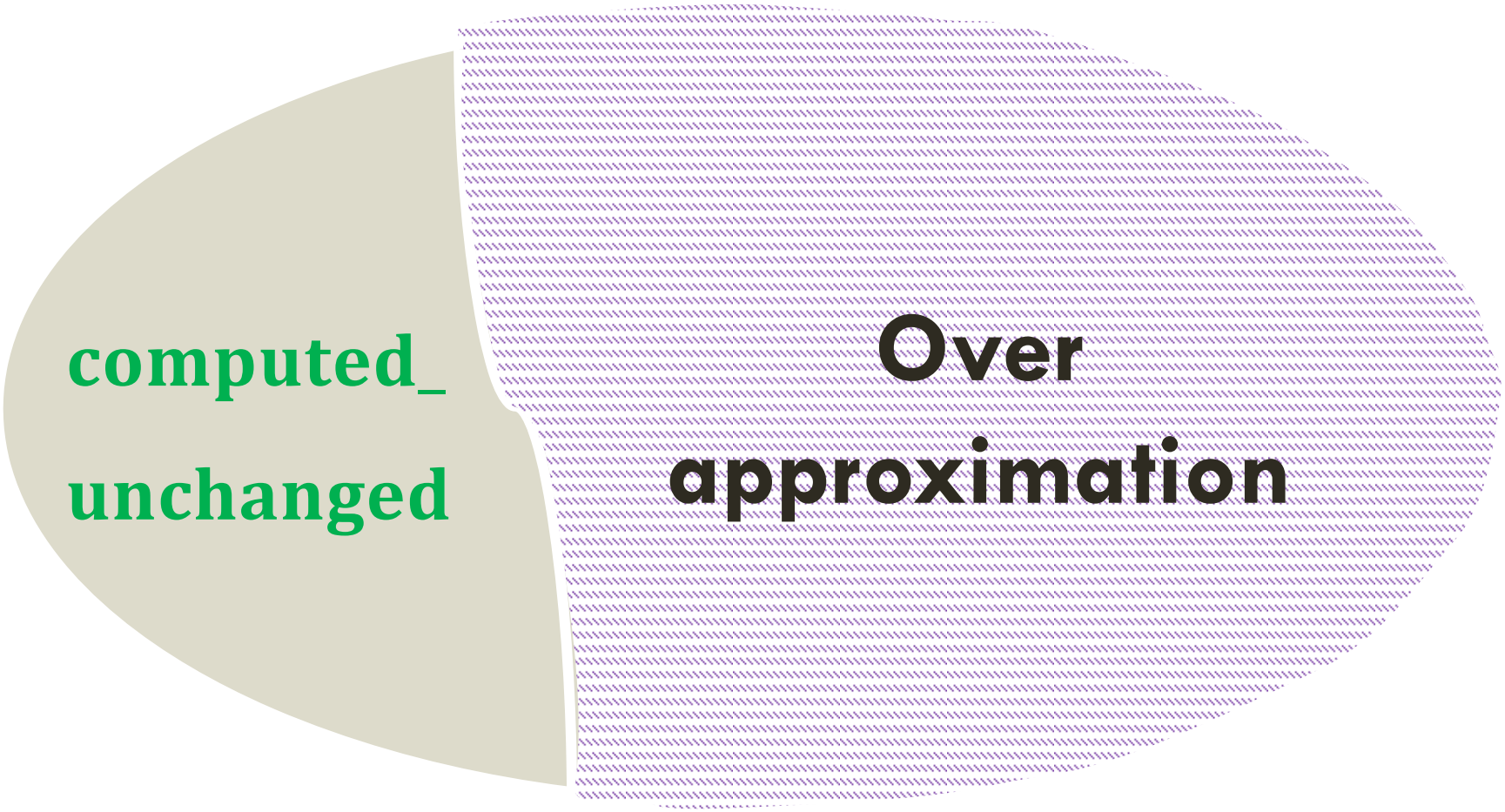
DIFFERENCE SUMMARY - COMPUTATION



DIFFERENCE SUMMARY - COMPUTATION



DIFFERENCE SUMMARY - COMPUTATION



PROGRAM REPRESENTATION

- Deterministic imperative programs
  - input-output behavior
- A 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

MAIN GOAL

Given a **matching function** between procedures in the old and new program versions,

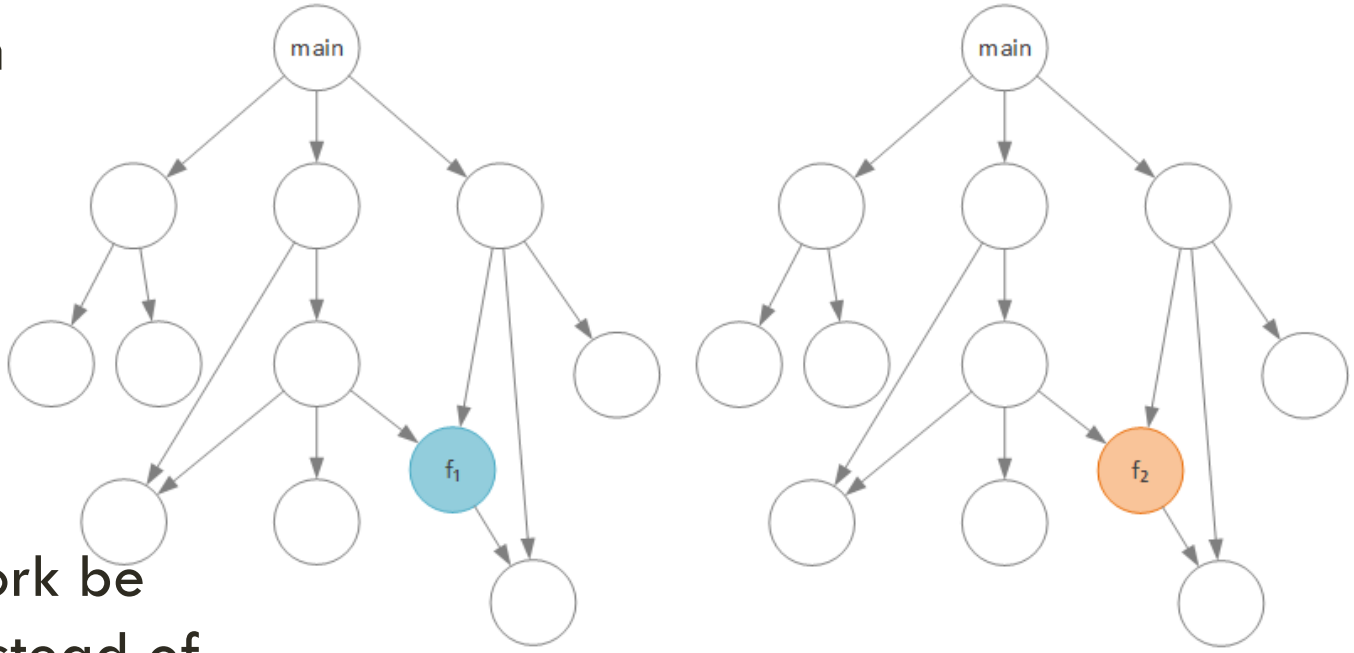
- compute
  $< computed\_changed, computed\_unchanged >$

  for every pair of matched procedures, and
- for inputs in $computed\_changed$, return the **different outputs** in both versions

HOW PROGRAMS CHANGE

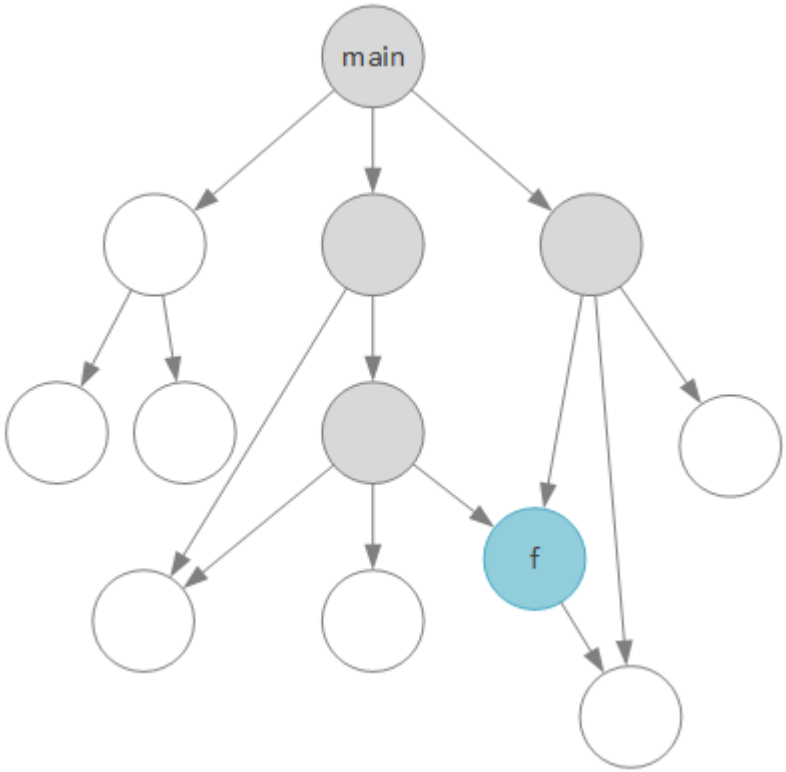
Changes are small, program are big

call graphs



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

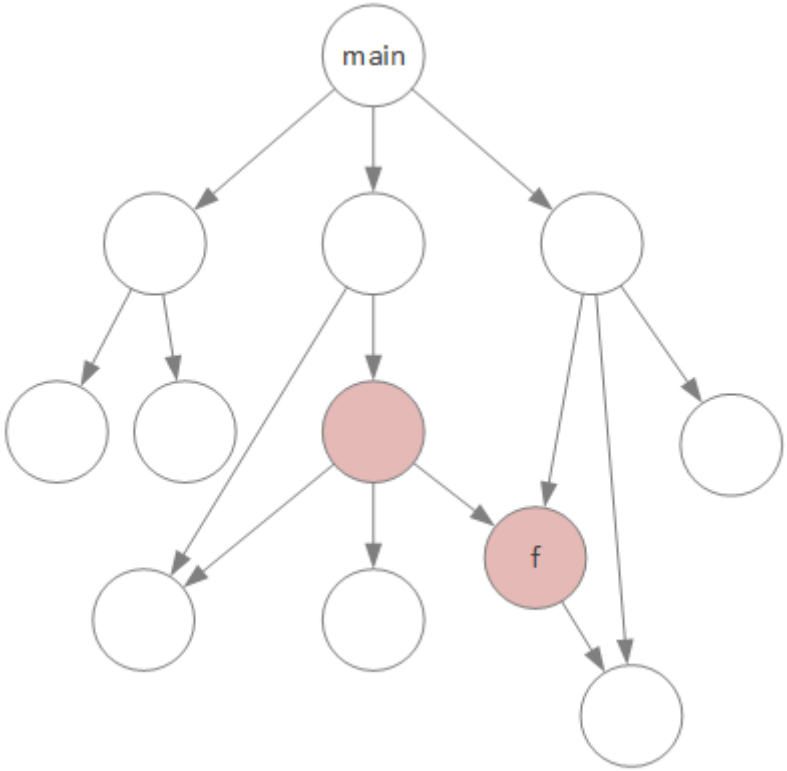
WHICH PROCEDURES COULD BE AFFECTED



potential semantical change

syntactical change

WHICH PROCEDURES ARE AFFECTED



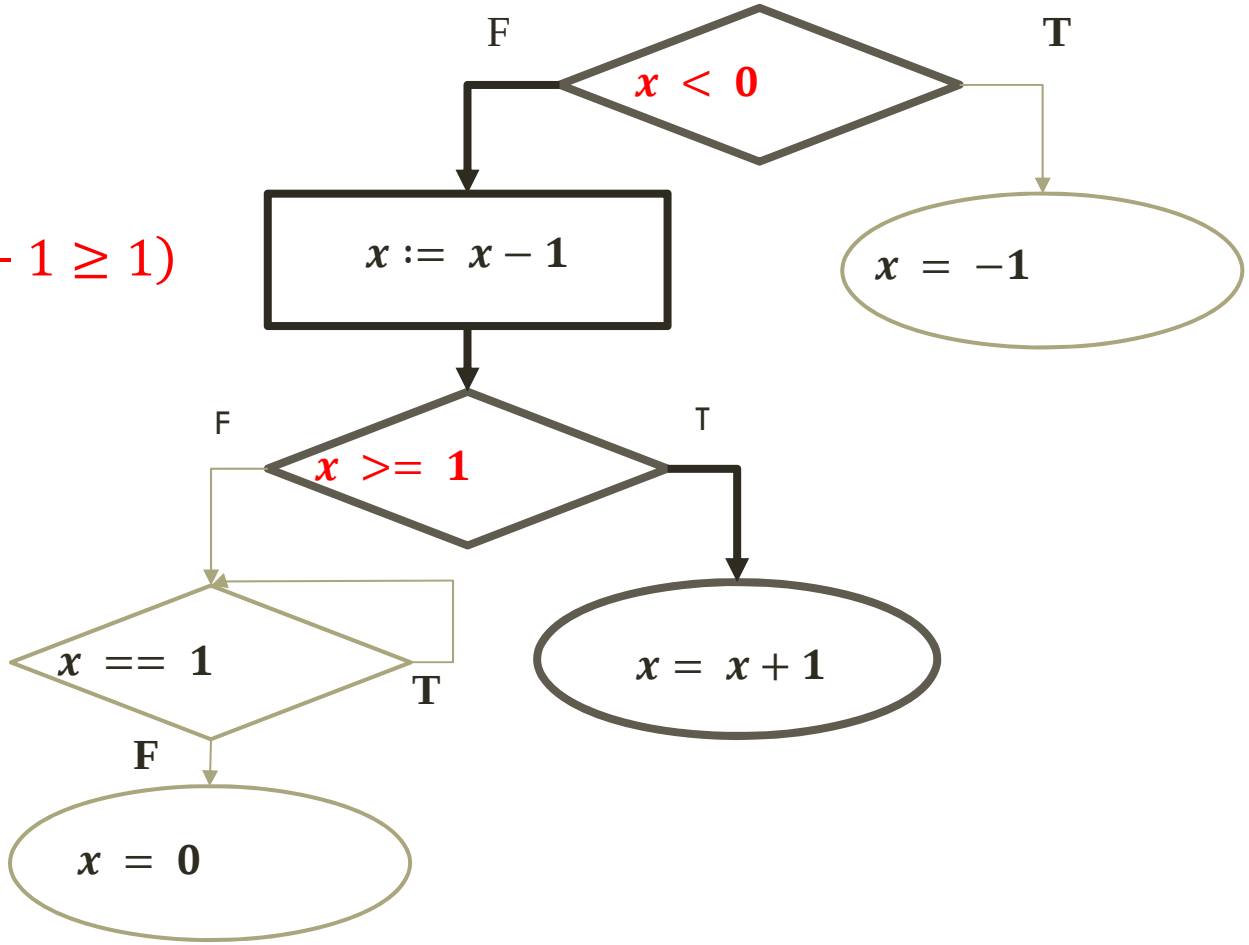
semantical change

MAIN IDEAS

- Modular analysis applied to one pair of procedures at a time
  - No inlining
- Only affected procedures are analyzed
- Procedures need not be fully analyzed:
  - Unanalyzed parts are abstracted using **uninterpreted functions**
  - Refinement is applied upon demand
- Anytime analysis:
  - Does not necessarily terminate
  - Its partial results are meaningful
  - The longer it runs, the more precise its results are

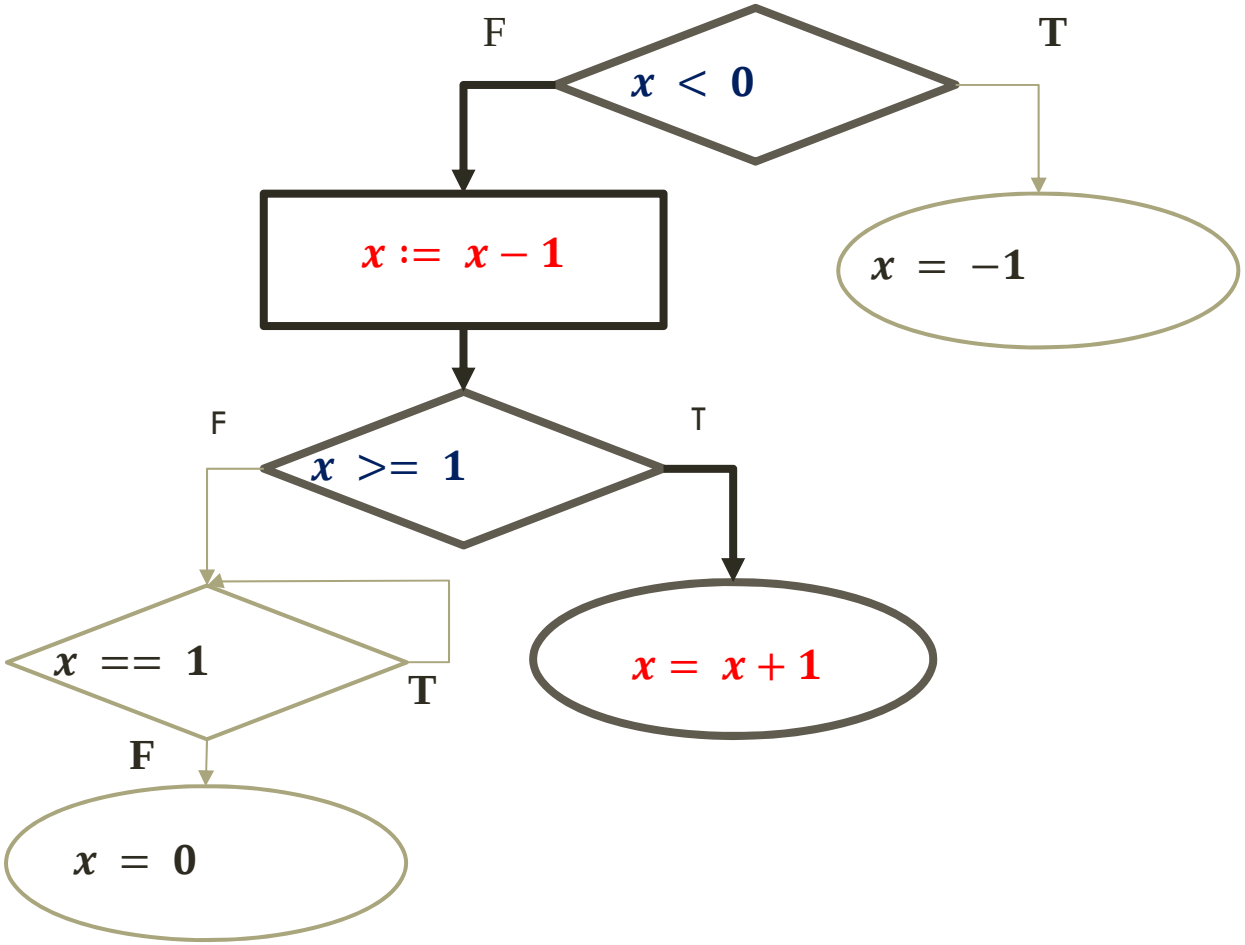
EXAMPLE - PATH CHARACTERIZATION

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



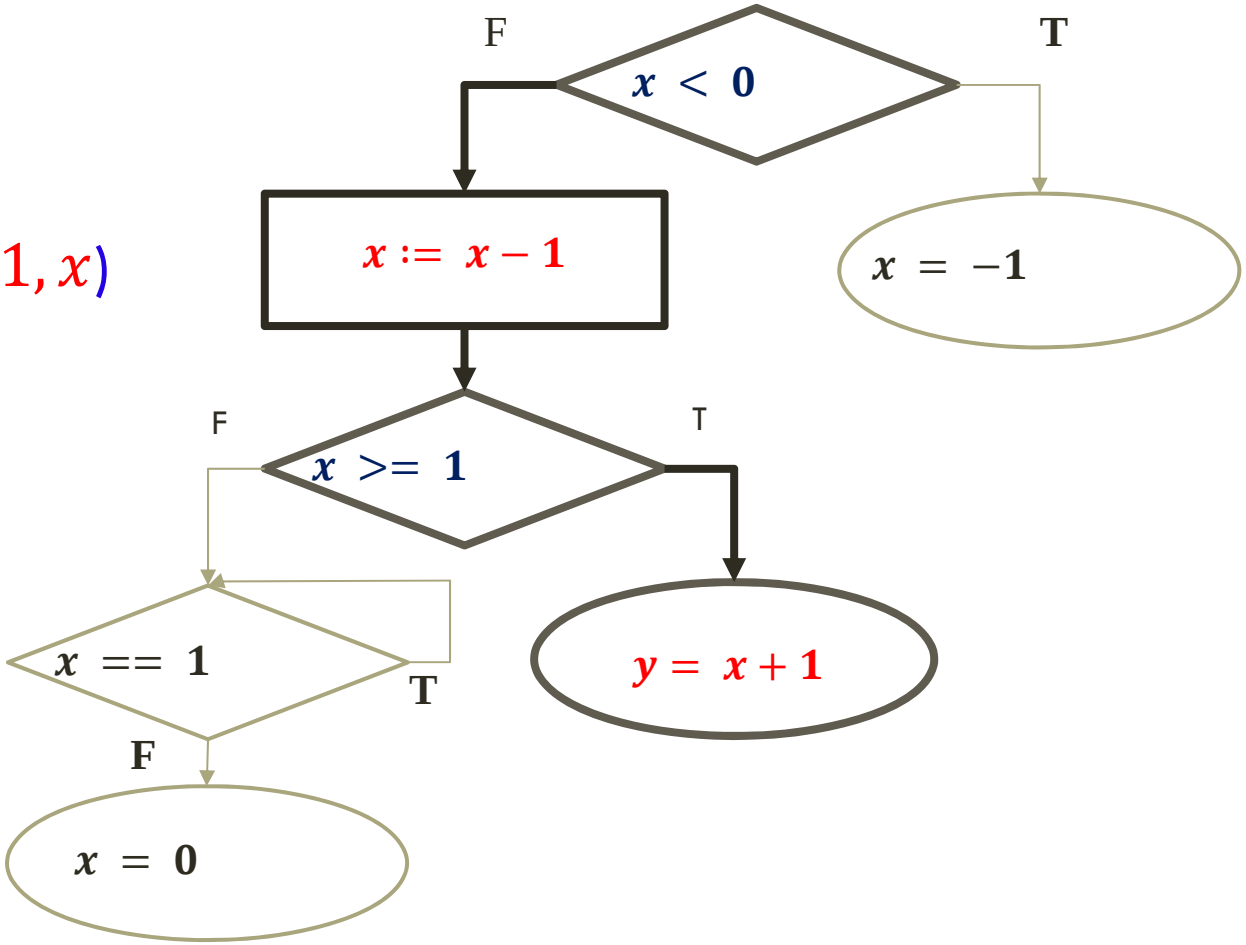
EXAMPLE - PATH CHARACTERIZATION

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



EXAMPLE - PATH CHARACTERIZATION

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



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

PROCEDURE SUMMARY

A 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

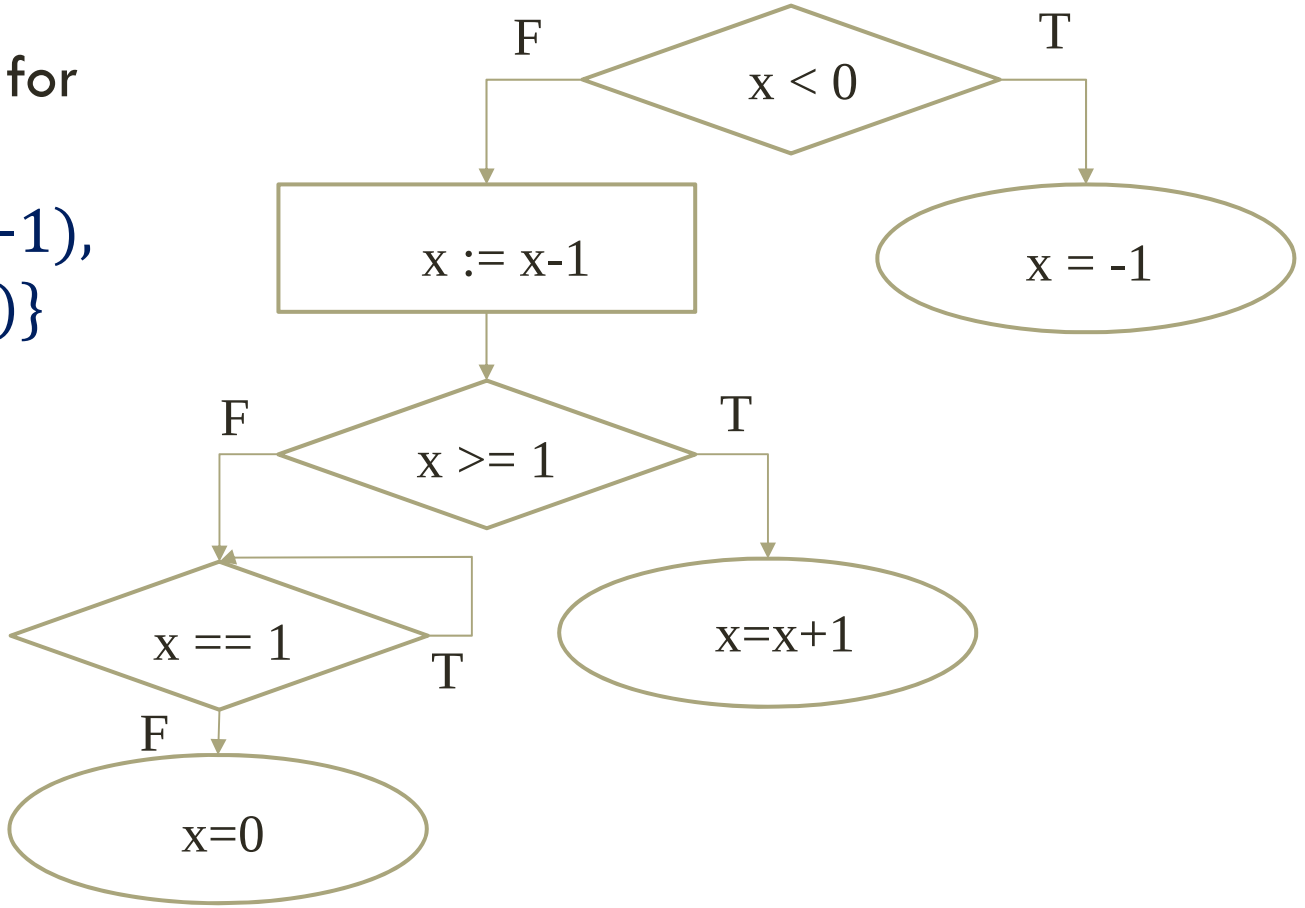
EXAMPLE

A possible summary for procedure p is:

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

Its uncovered part is

$x \geq 0 \wedge x < 2$



FROM SUMMARY TO DIFFERENCE SUMMARY

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

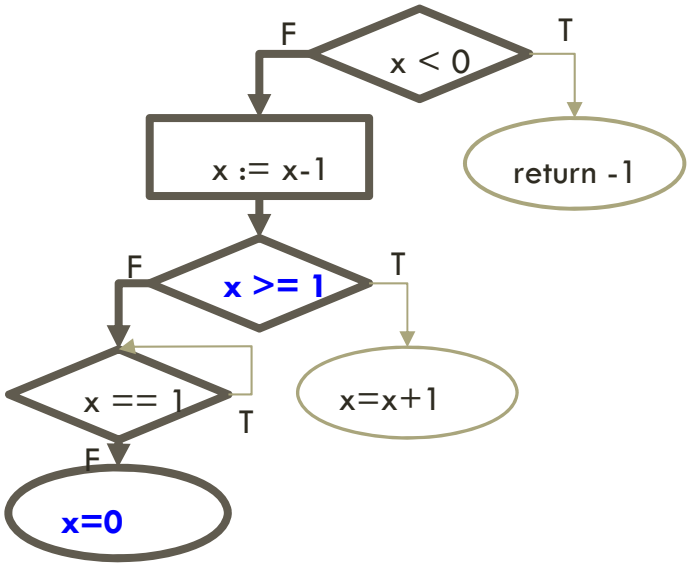
- diffCond $:= r_1 \wedge r_2 \wedge (t_1 \neq t_2)$

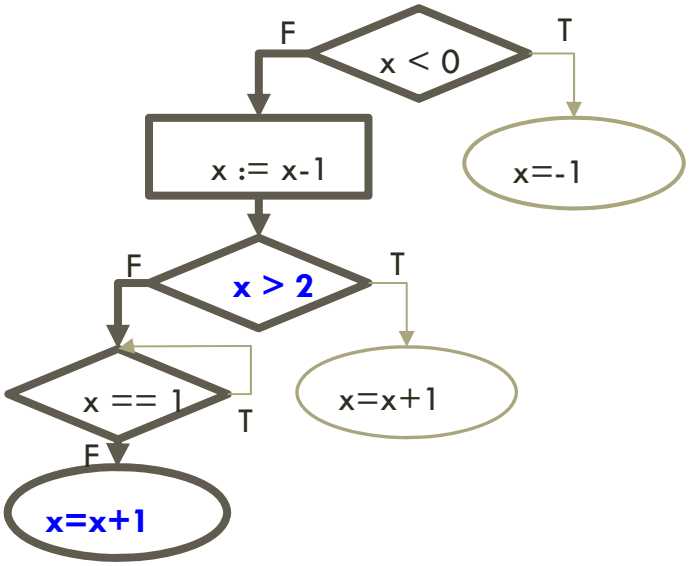
If diffCond is SAT, add it to computed_changed

- eqCond $:= r_1 \wedge r_2 \wedge (t_1 = t_2)$

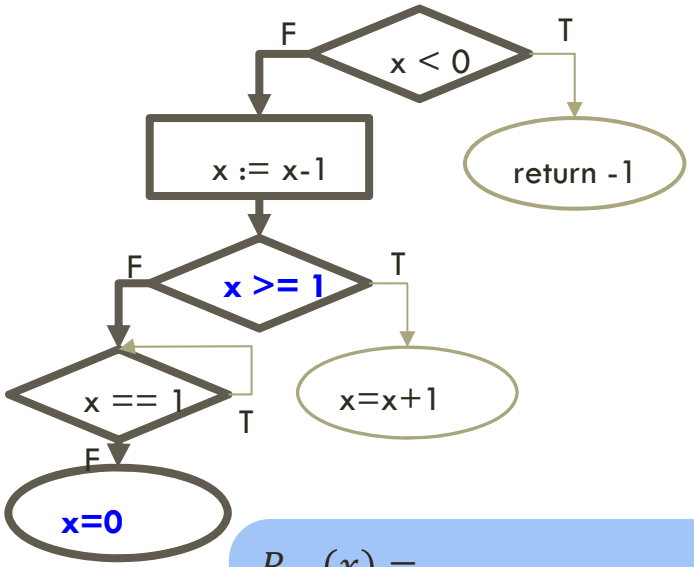
If eqCond is SAT, add it to computed_unchanged

EXAMPLE

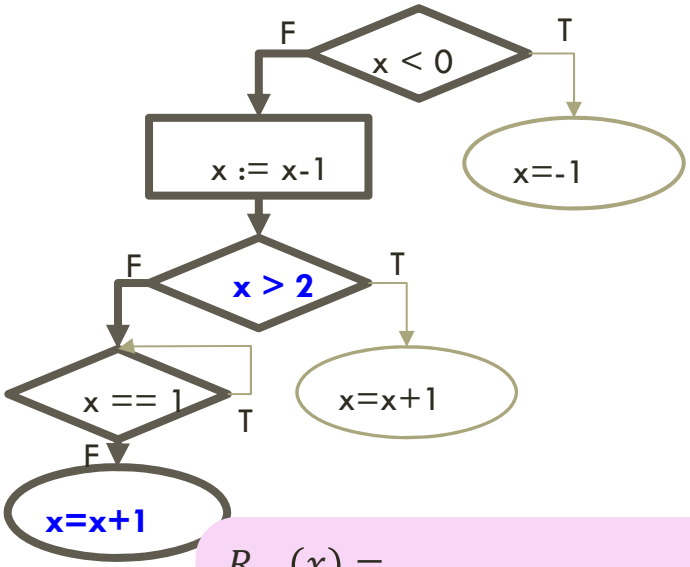

F
x < 0
T
x := x-1
return -1
F
x >= 1
T
x == 1
T
x=x+1
F
x=0


F
x < 0
T
x := x-1
x=-1
F
x > 2
T
x == 1
T
x=x+1
F
x=x+1

EXAMPLE

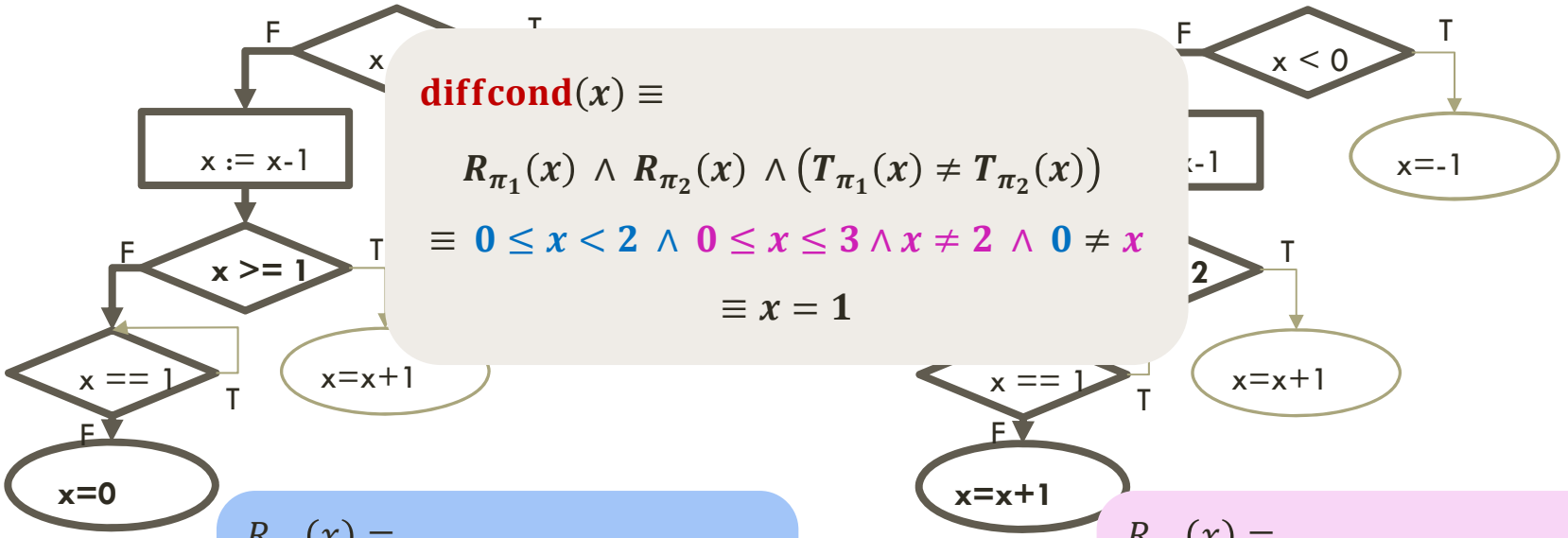


$R_{\pi_1}(x) =$
$x \geq 0 \wedge x - 1 < 1 \wedge x - 1 \neq 1$
$\equiv 0 \leq x < 2$
$T_{\pi_1}(x) = 0$



$R_{\pi_2}(x) =$
$x \geq 0 \wedge x - 1 \leq 2 \wedge x - 1 \neq 1$
$\equiv 0 \leq x \leq 3 \wedge x \neq 2$
$T_{\pi_2}(x) = x$

EXAMPLE



diffcond$(x) \equiv$

$$R_{\pi_1}(x) \wedge R_{\pi_2}(x) \wedge \left(T_{\pi_1}(x) \neq T_{\pi_2}(x)\right)$$

$$\equiv 0 \leq x < 2 \wedge 0 \leq x \leq 3 \wedge x \neq 2 \wedge 0 \neq x$$

$$\equiv x = 1$$

$R_{\pi_1}(x) =$
$x \geq 0 \wedge x - 1 < 1 \wedge x - 1 \neq 1$
$\equiv 0 \leq x < 2$
$T_{\pi_1}(x) = 0$

$R_{\pi_2}(x) =$
$x \geq 0 \wedge x - 1 \leq 2 \wedge x - 1 \neq 1$
$\equiv 0 \leq x \leq 3 \wedge x \neq 2$
$T_{\pi_2}(x) = x$

SYMBOLIC EXECUTION
FOR COMPUTING (R_π, T_π)

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	

COMPUTING THE SUMMARIES

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

$\boldsymbol{sum_g} = \{(\boldsymbol{r_1}, \boldsymbol{t_1}), \ldots, (\boldsymbol{r_n}, \boldsymbol{t_n})\}$ previously computed

Instead of in-lining g we compute:

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

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

MODULAR SYMBOLIC EXECUTION

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

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

Instead of in-lining g we compute:

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

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

MODULAR SYMBOLIC EXECUTION

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

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

Instead of in-lining g we compute:

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

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

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 calling procedure
- If needed - refine

ABSTRACTION

Unanalyzed parts of a procedure are replaced by uninterpreted functions

For matched procedures g_1,g_2 we have

- A common uninterpreted function $\boldsymbol{UF_{g_1,g_2}}$
- Individual uninterpreted functions $\boldsymbol{UF_{g_1}}$ and $\boldsymbol{UF_{g_2}}$

ABSTRACT MODULAR SYMBOLIC EXECUTION

For call $g_1(Y)$ with

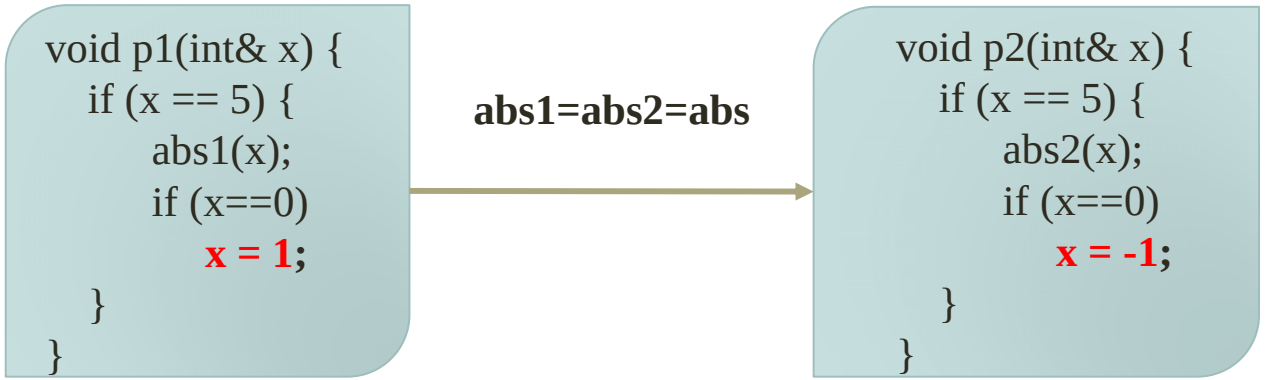
$$sum_{g_1} = \{(r_1, t_1), \ldots, (r_n, t_n)\}:$$

$$R_\pi^{i+1} = R_\pi^i$$
$$\mathrm{T}_\pi^{i+1} = ITE(r_1, t_1, \ldots ITE(r_n, t_n,$$
$$ITE(computed\_unchanged, UF_{g_1,g_2}, UF_{g_1}))$$

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

REFINEMENT

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



REFINEMENT

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

$$x = 5 \wedge \left(x' = UF_{abs_1,abs_2}(x)\right) \wedge x' = 0 \wedge 1 \neq -1$$

In order to check satisfiability, symbolic execution is applied to abs

REFINEMENT

- We run symbolic execution with the calling context as a precondition up to a certain bound
- In abs, we run the single corresponding path to the condition $x = 5$, add this path to the summary, and now we get:

$$x = 5 \wedge \left(x' = \left(\boldsymbol{x > 0?\, x}, UF_{abs_1, abs_2}(x)\right)\right) \wedge x' = 0 \wedge 1 \neq -1$$

which is now unsatisfiable

OVERALL ALGORITHM

- Start the analysis from the syntactically changed procedures. Analyze them with modular symbolic execution up to a certain bound
- Compute difference summaries for those procedures. If you can prove equivalence for all inputs – stop.
- Use refinement when needed
- Repeat for calling procedures
- Can be guided towards interesting procedures by the user

EXPERIMENTAL RESULTS – EQUIVALENT BENCHMARKS

We compared to two tools that prove equivalence between procedures:

- Regression Verification Tool (RVT)

Godlin, Benny, and Ofer Strichman. "**Regression verification.**" *Proceedings of the 46th Annual Design Automation Conference*. ACM, 2009.

- SymDiff

Lahiri, Shuvendu K., et al. "**SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs.**" *CAV*. Vol. 12. 2012.

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

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

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

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 have presented a differential analysis method that is:

- Modular (analyzes each procedure independently of its current use)
- 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

questions?