# PBE-Based Selective Abstraction and Refinement
for Efficient Property Falsification of Embedded Software


**Yoel Kim** and Yunja Choi

Software Disaster Research Center, KNU


KYUNGPOOK NATIONAL UNIVERSITY

STAAR
Software disasTer research center
Agile & Automated Response & repair

# Motivation

- **Verification** of safety properties in embedded control software
  - e.g., proving “the car reduces its speed within a specified time”
  - Often impractical; high verification complexity
- **Falsification** as an alternative way
  - Proving that a system does *not* satisfy a property
  - Should be rigorous

# Motivation

- **Verification** of safety properties in embedded control software
  - e.g., proving "the car reduces its speed within a specified time"
  - Often impractical; high verification complexity

- **Falsification** as an alternative way
  - Proving that a system does *not* satisfy a property
  - Should be rigorous

Goal: **Rigorous falsification of safety properties in embedded control software**

# Rigorous Falsification is Also **Expensive**

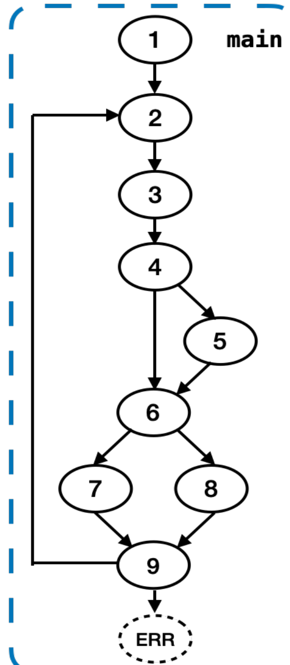
Embedded Control Program

```
 int speed = 0;
 void main() {
   int in, timer;
1: timer = 0;
2: while (true) {
3:   in = read_sensor();
4:   if(check(speed,in)){
5:     update(in);
     }
6:   if(speed >= 50) {
7:     timer = timer + 1;
     }
     else {
8:     timer = 0;
     }
9:   assert(timer < 1000);}}
```

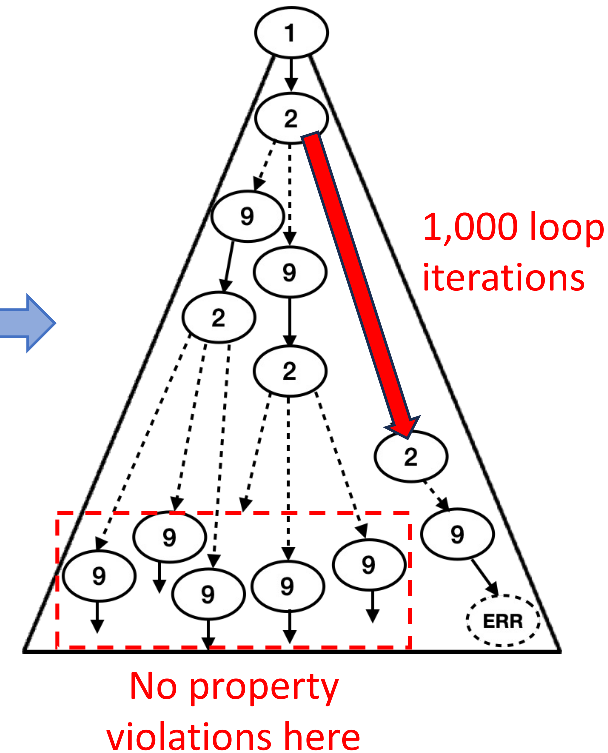
Reactive

Timed property

Model (CFG)

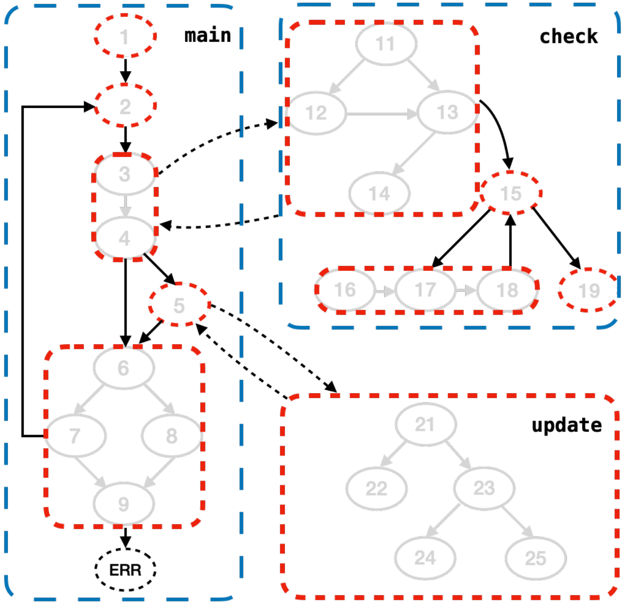



Search Space



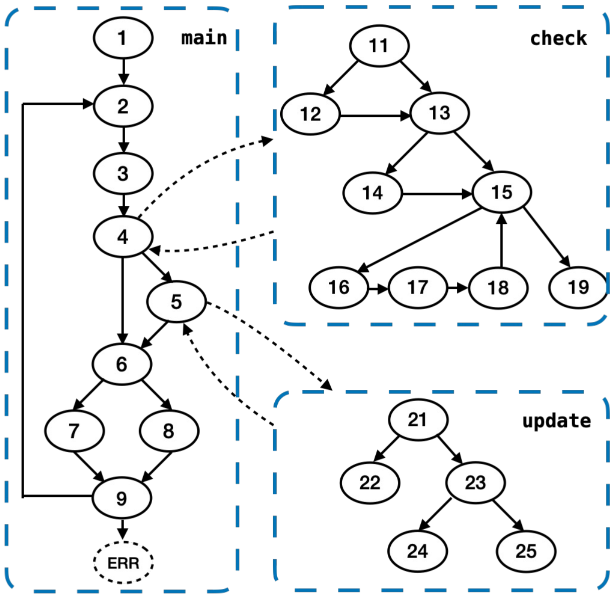
# Abstraction Makes it **Scalable**

Abstract Model



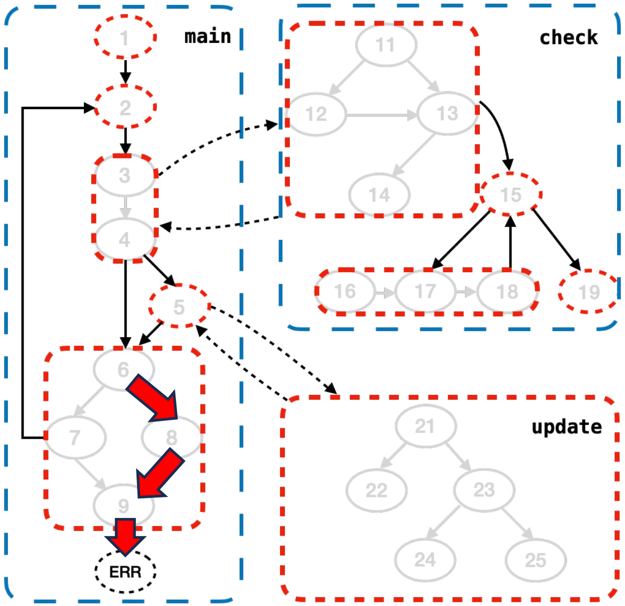

vs.

Concrete Model



# Abstraction Makes it Scalable but Imprecise

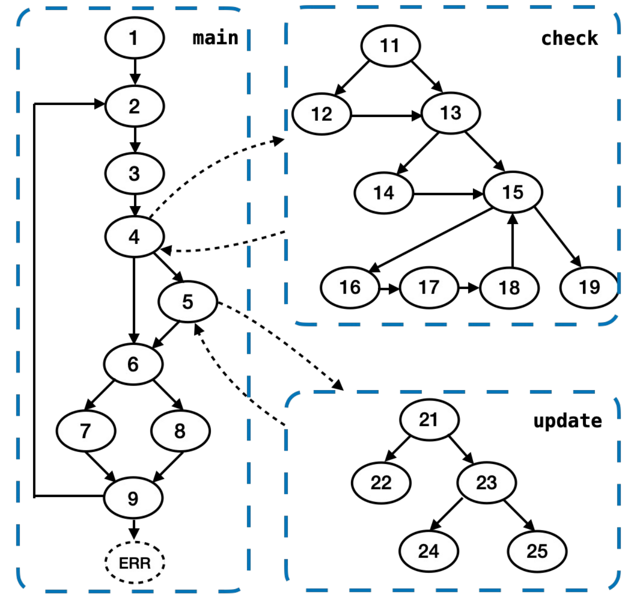
Abstract Model



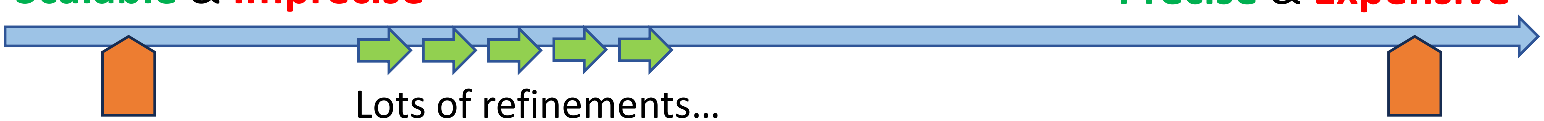

Infeasible

vs.

Concrete Model



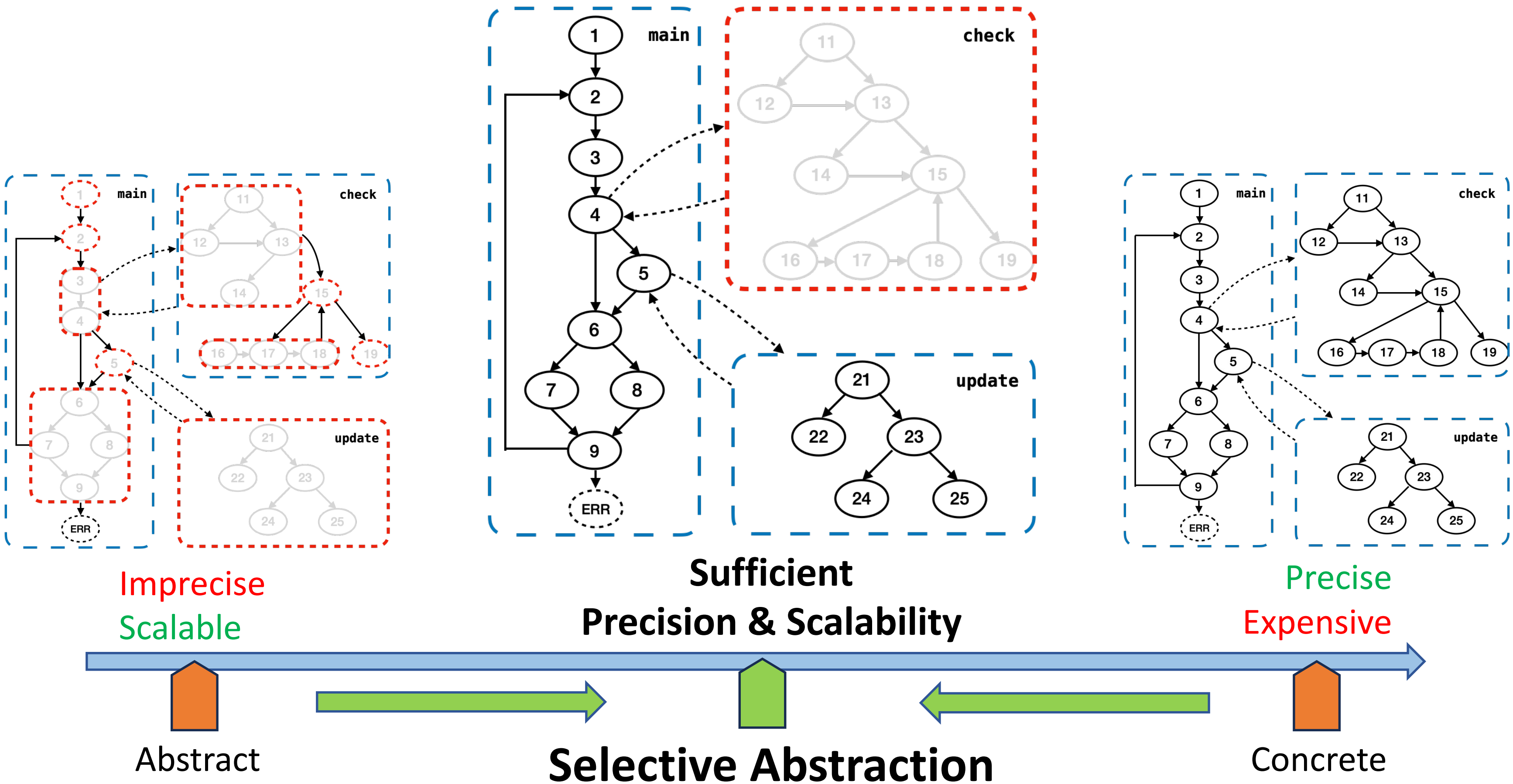

**Scalable & Imprecise** **Precise & Expensive**



Lots of refinements…

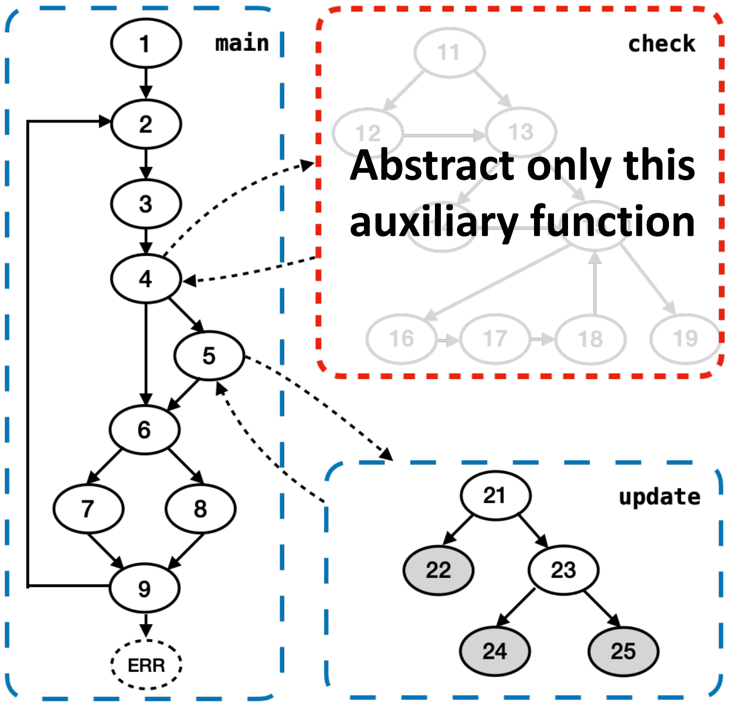
Abstract Concrete

# Our Approach: **Selective** Abstraction



# Key Observation

- Can be partitioned into **auxiliary functions** and **main control logic**
- **Auxiliary functions**: functions that do not update any global variables

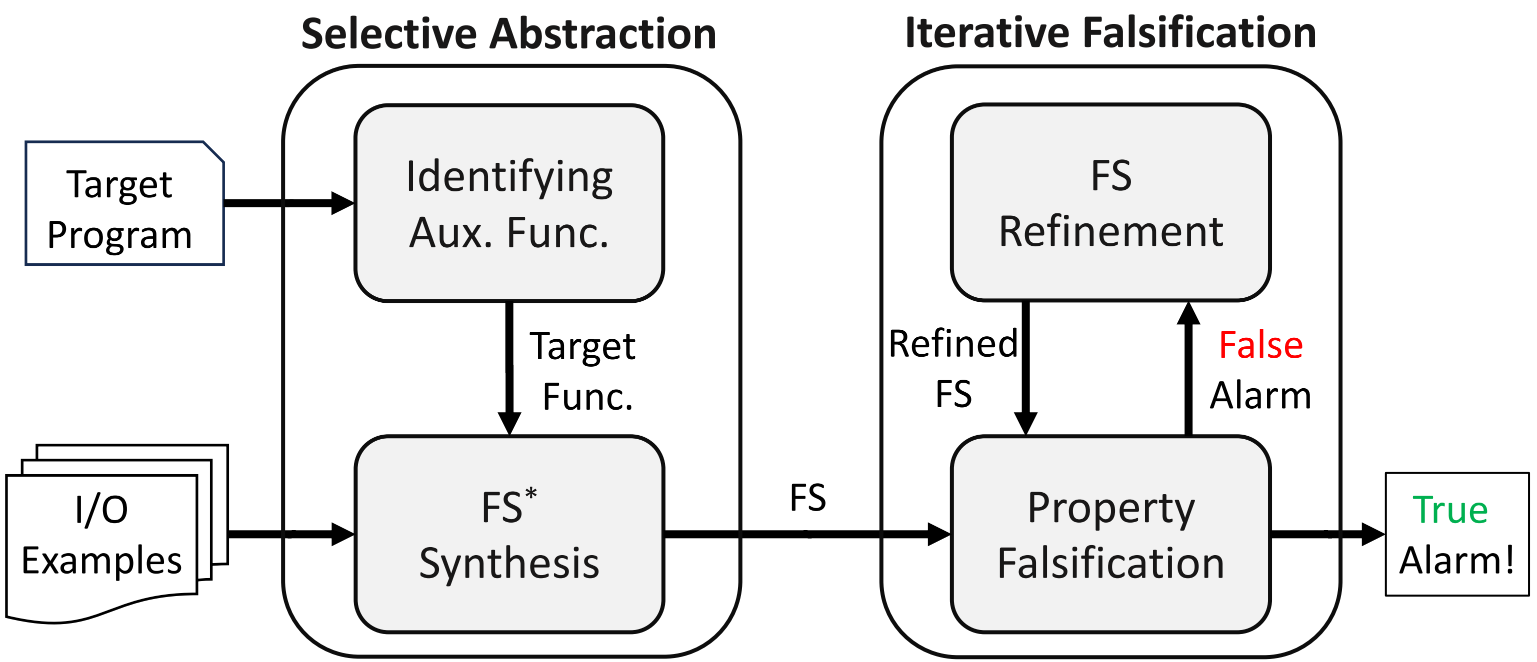



- **Main control logic** has a significant impact on the entire system & property
- Abstracting **main control logic** is highly likely to add refinement overhead

: Nodes that update global variables

# **PBEAR**: **PBE**-Based Selective **A**bstraction and **R**efinement

**Idea**: Synthesizing **function summary** using **P**rogramming-**B**y-**E**xample (**PBE**)




*FS: Function Summary

# Programming-By-Example (PBE)

- **PBE** takes input/output (I/O) examples and produces a program that satisfies the given examples
  - ✓ **Simpler** than the auxiliary function
  - ✓ **More precise** than a stub function returning a nondeterministic value

Auxiliary Function

```
int check(int a,int b){
  if (a < 0) a = -a;
  if (b < 0) b = -b;
  while (b != 0) {
    int t = b;
    b = a % b;
    a = t;
  }
  return a;}
```

I/O Examples

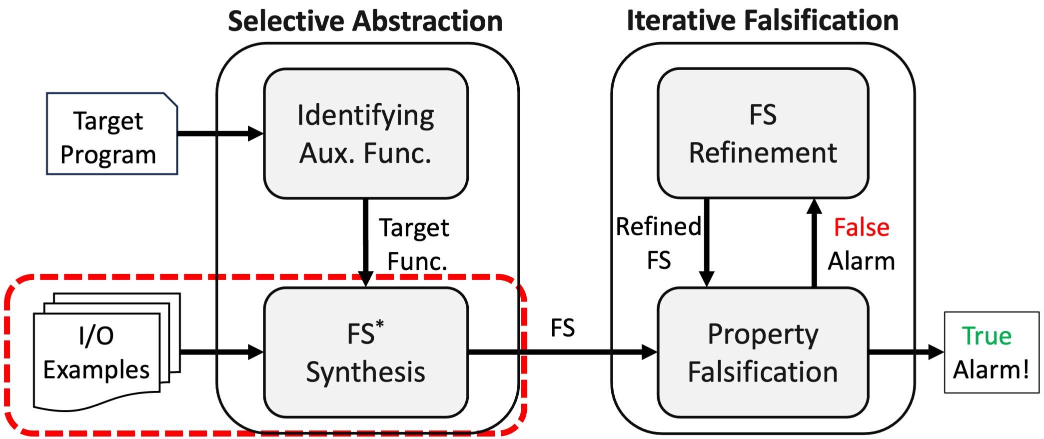
```
check(1,-1)==1
check(4, 2)==2
check(9,-3)==3
```

PBE

Function Summary (FS)

```
int fs_check(int a,int b){
  if (b < 0) return -b;
  else return b;
}
```

# Overfitting Problems in PBE



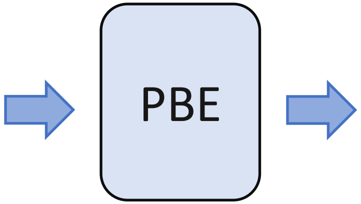

If we give I/O examples **arbitrarily**…

→ No generalization

→ Code size becomes large

I/O Examples

```
check(1,-1)==1
check(8, 3)==1
check(4, 2)==2
check(9,-3)==3
check(4, 8)==4
...
```



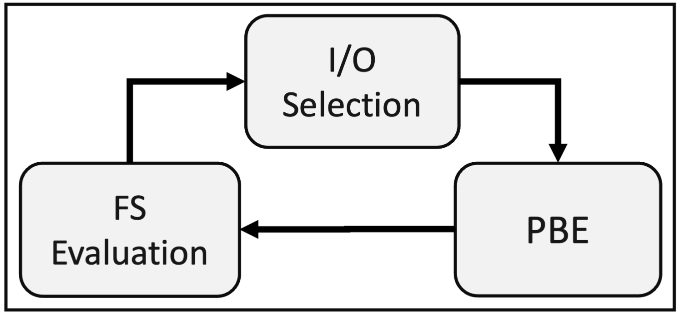

Overfitted Function Summary

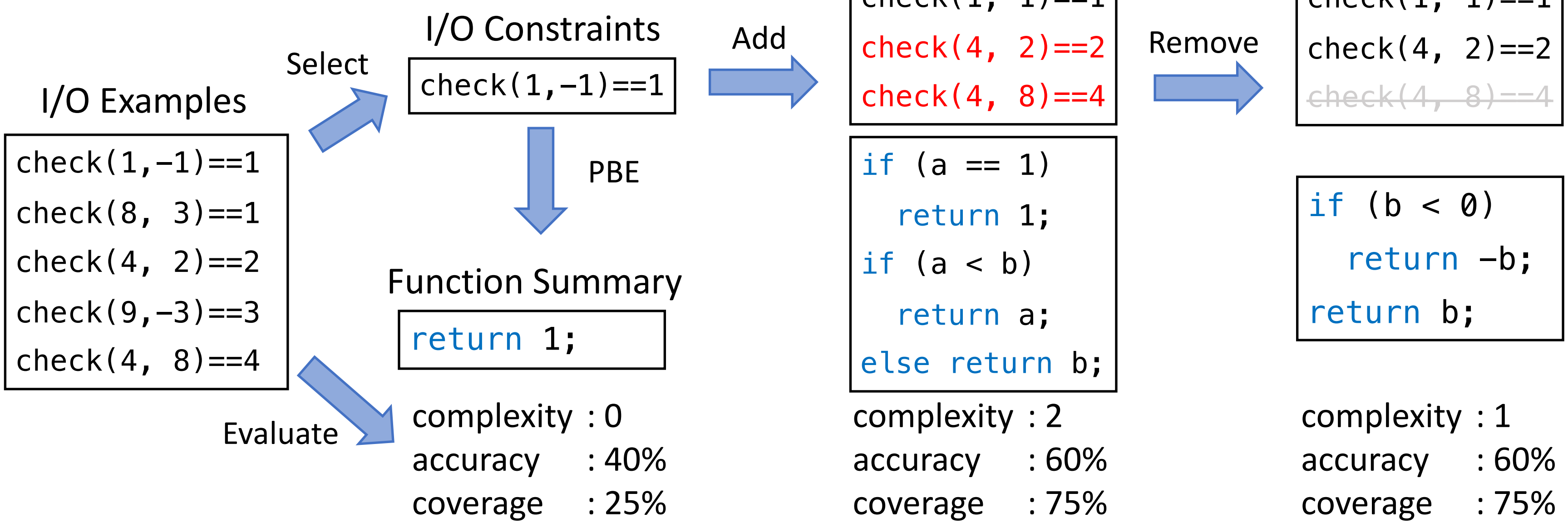
```
if (a == 1 || a == 8) return 1;
if (a == 4)
  if (b == 2) return b;
  else return a;
if (b < 0) return 3;
...
```

# Iterative Function Summary Synthesis

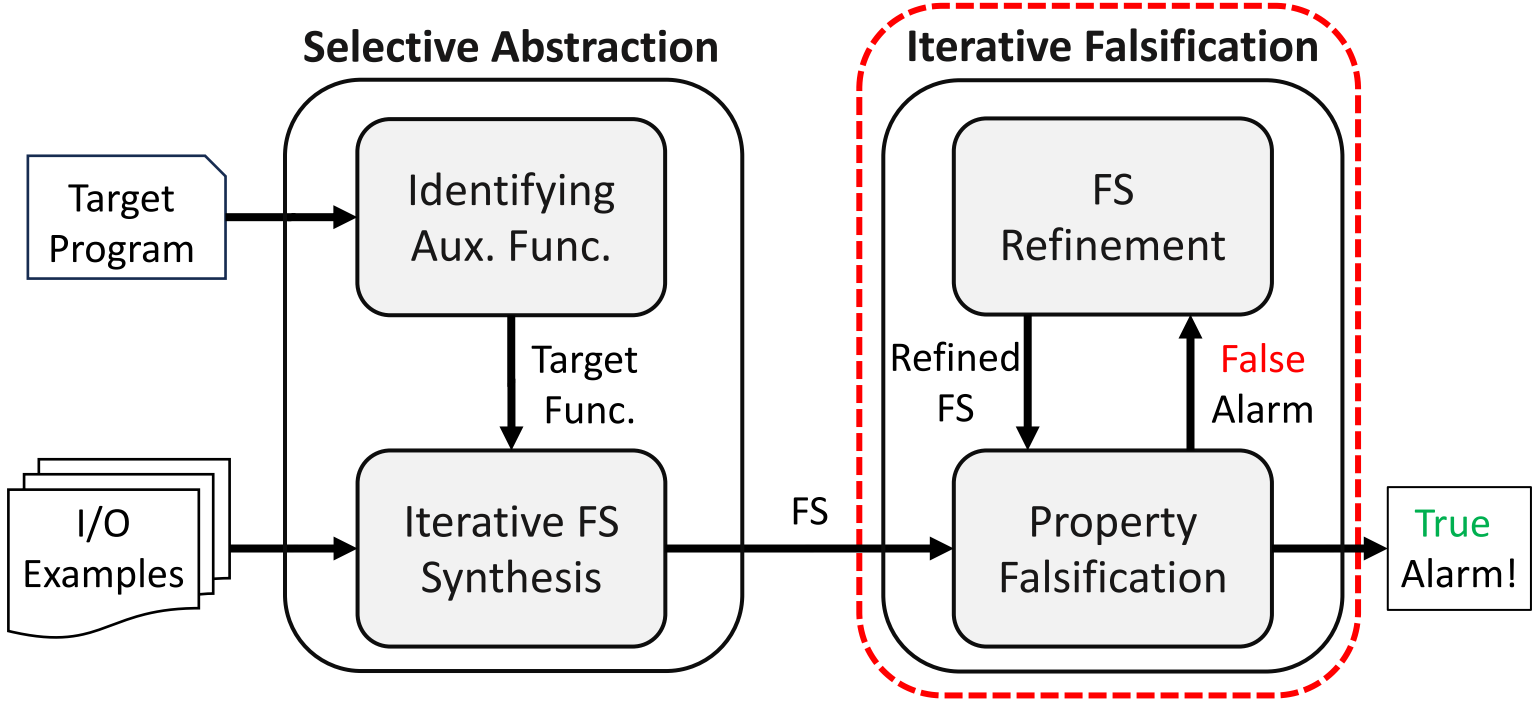
Measurement:

- **Complexity**: # of branch statements
- **Similarity**: accuracy and output coverage



# Iterative Falsification & Refinement



# Property Falsification

Replacing check with its FS

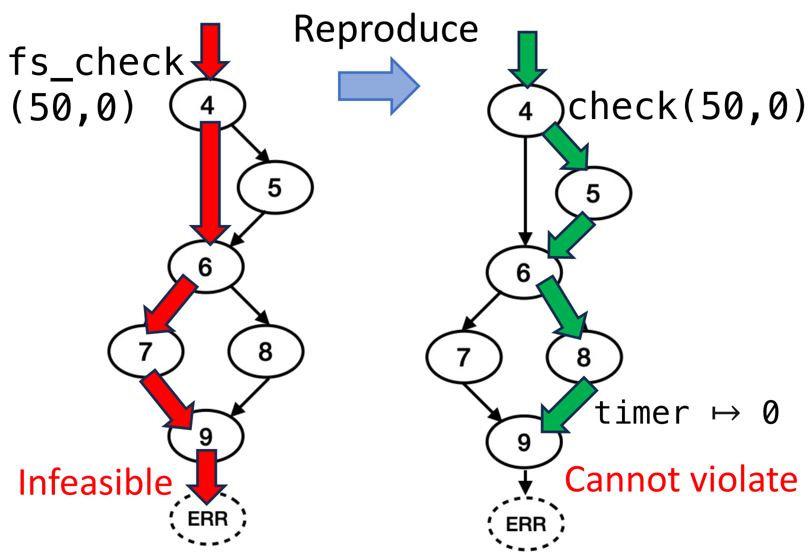
```
 int speed = 0;
 void main() {
   int in, timer;
1: timer = 0;
2: while (true) {
3:   in = read_sensor();
4:   if(fs_check(speed,in)){
5:     update(in);
     }
6:   if(speed >= 50) {
7:     timer = timer + 1;
     }
     else {
8:     timer = 0;
     }
9:   assert(timer < 1000);}}
```

```
int fs_check(int a,int b){
  if (b < 0) return -b;
  else return b;
}
```

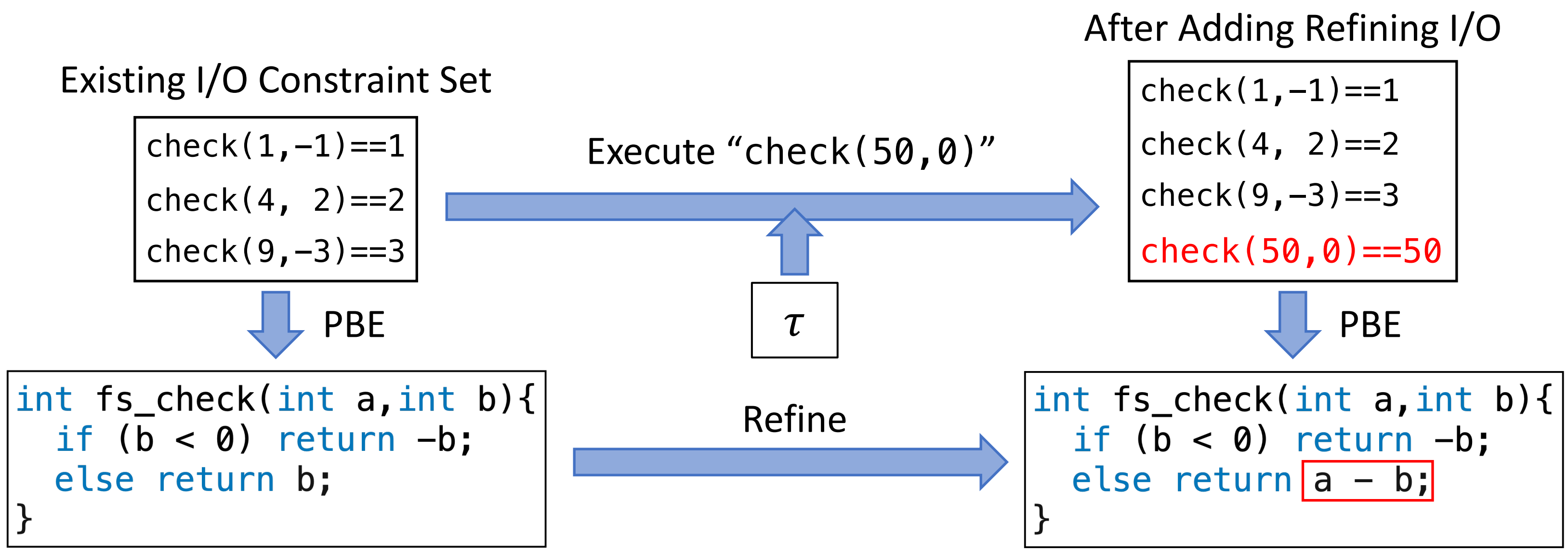
Efficient Falsification

Counterexample Trace

```
        ...
3: in ↦ 0
4: // fs_check(50,0) == 0
6: // if (speed >= 50)
7: timer ↦ 1000
9: // assert(timer < 1000)
```



# Baseline of Function Summary Refinement



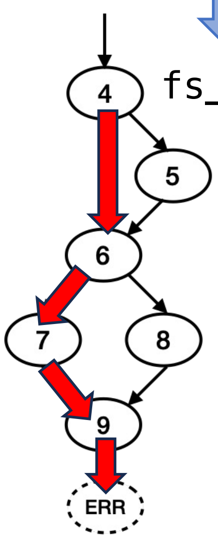
# Baseline of Function Summary Refinement

- **Problem**: no guarantee for avoiding re-exploration of identified infeasible paths

```
int fs_check(int a,int b){
  if (b < 0) return -b;
  else return b;
}
```

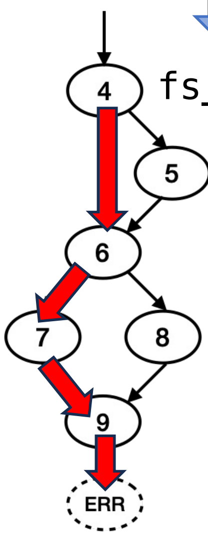
Refine

```
int fs_check(int a,int b){
  if (b < 0) return -b;
  else return a - b;
}
```




fs_check(50,0)

Different inputs,
same infeasible path!

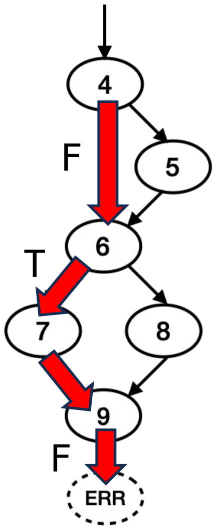



fs_check(50,50)

# Solution: Symbolic Alarm Filtering

Counterexample Trace

```
              ...
3: in ↦ 0
4: // fs_check(50,0) == 0
6: // if (speed >= 50)
7: timer ↦ 1000
9: // assert(timer < 1000)
```




Symbolize

**Path Program**

```
int main() {
              ...
3: in = read_sensor();
4: ret = check(speed,in);
   assume(ret == 0);
6: assume(speed >= 50);
7: timer = timer + 1;
9: assume(timer >= 1000);
     /* Feasible!! */     }
```

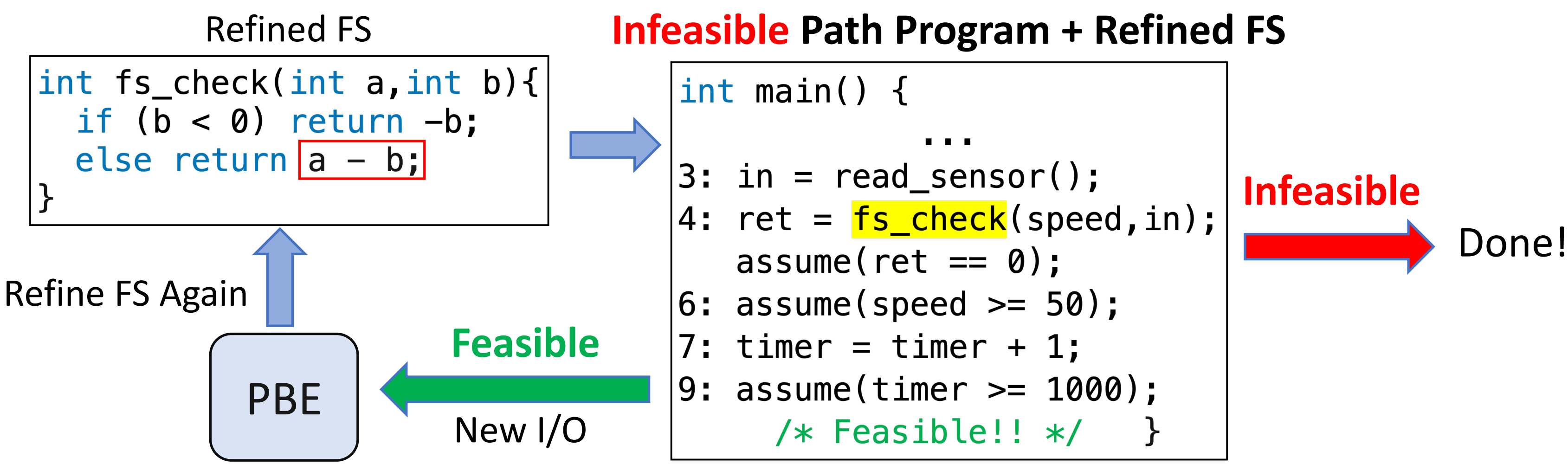
**Feasible**
(**True** Alarm)

**Infeasible**
(**False** Alarm)

- **Path program**: a symbolic path extracted from the counterexample
  - `assume` statement: the evaluation of a branch condition
- Utilize **infeasible** **path program** to improve FS refinement

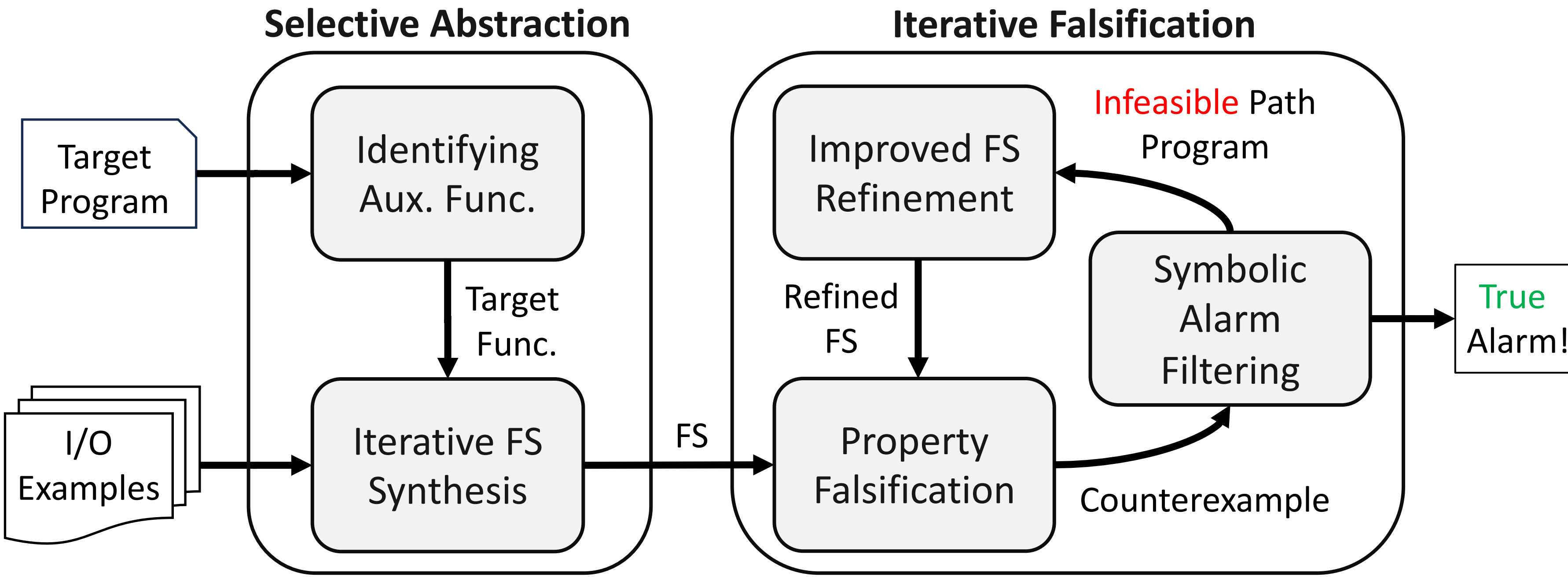
# Our Improved Function Summary Refinement

- **Idea**: Refining FS to *not* make the **infeasible** path program **feasible**




- Can guide PBEAR to explore another path
- Checking **infeasible path program** vs. **entire program** with refined FS

# Overall Process of **PBEAR**



# Experiments

- Target
  - 16 benchmark programs from SV-COMP
  - 15 safety properties from 3 embedded software (about 800~1,000 LoC)
  - object following car, elevator controller, clean-up robot

- RQ1: Performance of PBEAR
  - CBMC: bounded model checker
  - CPAchecker: predicate abstraction

- RQ2: Effectiveness of our improved FS refinement
  - $\text{PBEAR}^{\text{base}}$: no symbolic alarm filtering

# RQ1: Performance of PBEAR (on SV-COMP)

| | # of refinements | total time | total memory | # of detected true alarms |
|---|---|---|---|---|
| PBEAR | 9 | 485 s | 3.1 GB | 15/16 |
| CBMC | N/A | 177 s | 3.9 GB | 16/16 |
| CPAchecker | 57 | 4,334 s | 10.3 GB | 13/16 |

* T/O: 900 s

- **CBMC was the best:** the overhead of refinement did not pay off on simple programs
- **PBEAR** showed competitive performance on **general programs**

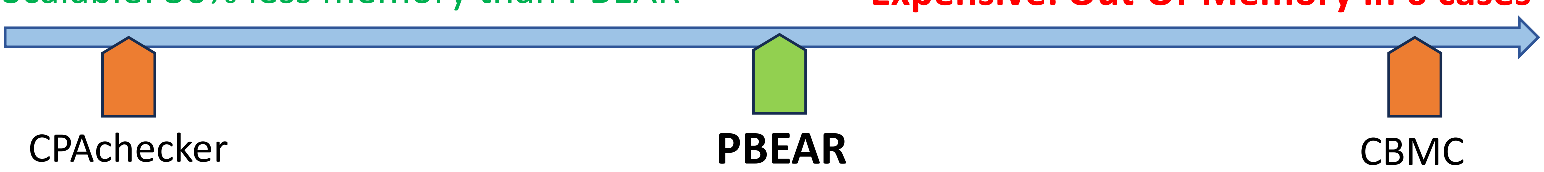
# RQ1: Performance of PBEAR (on Embedded Software)

| | # of refinements | total time | total memory | # of detected true alarms |
|---|---|---|---|---|
| **PBEAR** | **77** | **135.4 h** | **423.5 GB** | **14/15** |
| CBMC | N/A | 96.9 h | 748.9 GB | 9/15 |
| CPAchecker | 2,556 | 936.3 h | 336.8 GB | 2/15 |

* T/O: 3 days; Out-Of-Memory: 80 GB

**Imprecise: 13 T/Os with 2,556 refinements**
Scalable: 30% less memory than PBEAR

Precise: no need for refinement
**Expensive: Out-Of-Memory in 6 cases**



# RQ2: Effectiveness of Our Improved FS Refinement

| | # of refinements | total time | # of detected true alarms |
|---|---|---|---|
| **PBEAR** | **86** | **133.6 h** | **13/15** |
| PBEAR$^{base}$ (without symbolic alarm filtering) | 210 | 217.4 h | 12/15 |

* We omitted benchmarks where any refinements were not conducted by PBEAR$^{b}$

PBEAR$^{base}$ took **additional 124 refinements** and **83.6 more hours**

# Discussion

- Contributions
  - **PBEAR** is **the first work** utilizing **PBE** for **selective abstraction**
  - Provide a **novel refinement approach** based on **symbolic alarm filtering**
  - Show **promising results** on three **embedded software** and competitive performance on general programs

- Limitations
  - PBEAR is neither sound nor complete abstraction
  - PBEAR relies on the performance of PBE
  - PBEAR cannot solve some limitations of PBE (e.g., dealing with complex data types)