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

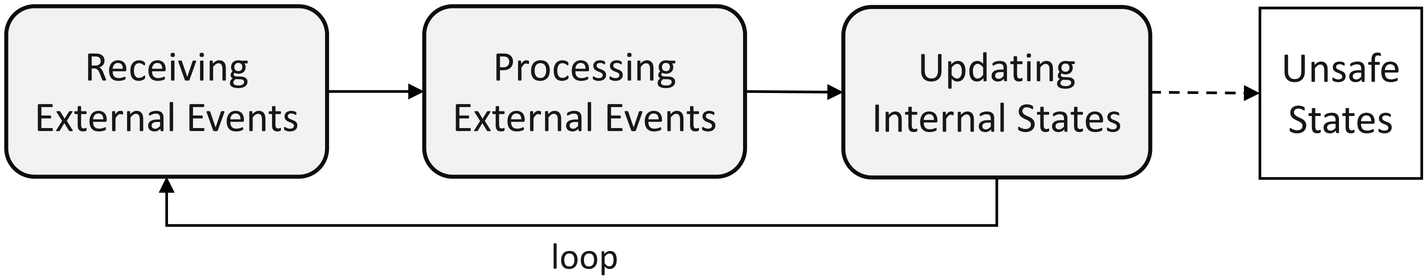
STAAR

SOFTWARE DISASTER RESEARCH CENTER

AGILE & AUTOMATED RESPONSE & REPAIR

Motivation

- Major characteristics of **embedded software**:
  - It constantly interacts with external events within an **infinite loop**
  - It maintains **internal states** whose updates must be traced
  - **Safety properties** may be violated only after many loop iterations



Motivation

- **Verification/Falsification** of embedded software:
  - e.g., (dis)proving "A car reduces its speed within a specified time"
  - We have seen decades of effort for this:
    - Most of them were **model-based** approaches applied to cyber-physical systems, automobiles, or medical devices, etc.
    - **Code-based** property verification/falsification is very **expensive**

Table 1: Code-based property falsification of an elevator controller (about 900 LoC)

Tools	# of refinements	Time usage	Memory usage
CBMC	N/A	8.3 hours	**> 80 GB (OOM)**
CPAchecker	**418**	**> 72 hours (T/O)**	7.1 GB

Motivating Example

Embedded Control Program

```c
int process(int *data) {
  int max_area = -1;
  int max_index = -1;
  for(int i = 0; i < data[0]; i++) {
    if(data[5 * i + 1] != 0) {
      int xul = data[i * 5 + 2];
      int xlr = data[i * 5 + 3];
      int yul = data[i * 5 + 4];
      int ylr = data[i * 5 + 5];
      int width = abs(xul - xlr);
      int height = abs(yul - ylr);
      int area = width * height;
      if(area >= max_area) {
        max_area = area;
        max_index = i;
      }
    }
  }
  return max_index; }
```

```c
int timer = 0;
int speed = 0;
#define MAX_SPEED 100

int main() {
  int data[41];
  int i;
  while(1) {
    data = receive();
    i = process(data);
    if(i >= 0) {
      update(i, data);
    }
    assert(is_safe());
  }
}
```

```c
void update(int i, int *data) {
  speed = speed + data[i * 5 + 1];
  if(speed > MAX_SPEED) {
    speed = MAX_SPEED; }}
```

```c
int is_safe() {
  if(speed >= MAX_SPEED){
    timer = timer + 1;
  }
  else timer = 0;
  return timer < 1000;}
```

- Checking this **concrete** program suffers from state space explosion (OOM)
- Abstracting this program too **coarsely** produces many false alarms

Motivating Example

Embedded Control Program

```
int process(int *data) {
  int max_area = -1;
  int max_index = -1;
  for(int i = 0; i < data[0]; i++) {
    if(data[5 * i + 1] != 0) {
      int
      int
      int
      int
      int
      int
      int
      if(area >= max_area) {
        max_area = area;
        max_index = i;
      }
    }
  }
  return max_index; }
```

```
int timer = 0;
int speed = 0;
#define MAX_SPEED 100

int main() {
    if(i >= 0) {
      update(i, data);
    }
    assert(is_safe());
  }
}
```

```
) {
 + 1];
```

```
int is_safe() {
  if(speed >= MAX_SPEED){
    timer = timer + 1;
  }
  else timer = 0;
  return timer < 1000;}
```

We need to find a better abstraction approach for embedded software domain

- Checking this **concrete** program suffers from state space explosion (OOM)
- Abstracting this program too **coarsely** produces many false alarms

Main Idea 1: Selective Abstraction

Auxiliary Function

```c
int process(int *data) {
  int max_area = -1;
  int max_index = -1;
  for(int i = 0; i < data[0]; i++) {
    if(data[5 * i + 1] != 0) {
      int xul = data[i * 5 + 2];
      int xlr = data[i * 5 + 3];
      int yul = data[i * 5 + 4];
      int ylr = data[i * 5 + 5];
      int width = abs(xul - xlr);
      int height = abs(yul - ylr);
      int area = width * height;
      if(area >= max_area) {
        max_area = area;
        max_index = i;
      }
    }
  }
  return max_index; }
```

Main Control Logic

```c
int timer = 0;
int speed = 0;
#define MAX_SPEED 100

int main() {
  int data[41];
  int i;
  while(1) {
    data = receive();
    i = process(data);
    if(i >= 0) {
      update(i, data);
    }
    assert(is_safe());
  }
}
```

```c
void update(int i, int *data) {
  speed = speed + data[i * 5 + 1];
  if(speed > MAX_SPEED) {
    speed = MAX_SPEED; }}
```

```c
int is_safe() {
  if(speed >= MAX_SPEED){
    timer = timer + 1;
  }
  else timer = 0;
  return timer < 1000;}
```

- **Auxiliary functions:** functions that do *not* update any internal states (e.g., global variables)

Main Idea 1: Selective Abstraction

Auxiliary Function

```
int process(int *data) {
  int max_area = -1;
  int max_index = -1;
  for(int i = 0; i < data[0]; i++) {
    if(data[5 * i + 1] != 0) {
      int xul = data[i * 5 + 2];
```

Abstraction!

```
      int height = abs(yul - ylr);
      int area = width * height;
      if(area >= max_area) {
        max_area = area;
        max_index = i;
      }
    }
  }
  return max_index; }
```

Main Control Logic

```
int timer = 0;
int speed = 0;
#define MAX_SPEED 100

int main() {
  int data[41];
  int i;
  while(1) {
    data = receive();
    i = process(data);
    if(i >= 0) {
      update(i, data);
    }
    assert(is_safe());
  }
}
```

No Abstraction

```
i, int *data) {
+ data[i * 5 + 1];
X_SPEED) {
_SPEED; }}
```

```
int is_safe() {
  if(speed >= MAX_SPEED){
    timer = timer + 1;
  }
  else timer = 0;
  return timer < 1000;}
```

- **Main control logic** has a significant impact on the program & property
- Abstracting the **main control logic** is likely to produce many false alarms

Main Idea 2: PBE-Based Abstraction

(1) Initial Abstraction

```
int process(int *data) {
  return nondet();
}
```

240 seconds, but false alarm

(2) **Refined Abstraction**

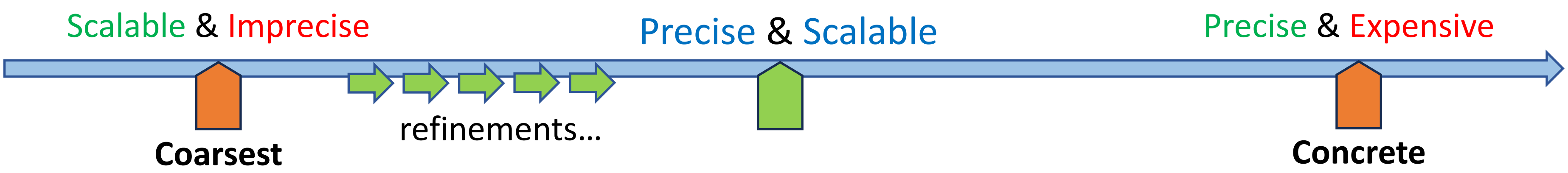
```
int process(int *data) {
  if(0 != data[36])
    if(data[0] >= data[15]) return 6;
    else return 7;
  else if(0 != data[1])
    if(data[12] < data[40]) return 5;
    else if(data[18] >= data[20]) return 1;
    else return 0;
  else if(0 != data[21]) return 4;
  else return -1; }
```

260 seconds, and **true alarm**!

(3) No Abstraction

```
int process(int *data) {
  int max_area = -1;
  int max_index = -1;
  for(int i = 0; i < data[0]; i++) {
    if(data[5 * i + 1] != 0) {
      int xul = data[i * 5 + 2];
      int xlr = data[i * 5 + 3];
      int yul = data[i * 5 + 4];
      int ylr = data[i * 5 + 5];
      int width = abs(xul - xlr);
      int height = abs(yul - ylr);
      int area = width * height;
      if(area >= max_area) {
        max_area = area;
        max_index = i;
      }
    }
  }
  return max_index; }
```

Out-Of-Memory



Main Idea 2: PBE-Based Abstraction

(3) No Abstraction

```c
int process(int *data) {
  int max_area = -1;
  int max_index = -1;
  for(int i = 0; i < data[0]; i++) {
    if(data[5 * i + 1] != 0) {
      int xul = data[i * 5 + 2];
      int xlr = data[i * 5 + 3];
      int yul = data[i * 5 + 4];
      int ylr = data[i * 5 + 5];
      int width = abs(xul - xlr);
      int height = abs(yul - ylr);
      int area = width * height;
      if(area >= max_area) {
        max_area = area;
        max_index = i;
      }
urn max_index; }
```

Out-Of-Memory

(2) **Programming-By-Example (PBE) Based Abstraction**

```c
int process(int *data) {
  if(0 != data[36])
    if(data[0] >= data[15]) return 6;
    else return 7;
  else if(0 != data[1])
    if(data[12] < data[40]) return 5;
    else if(data[18] >= data[20]) return 1;
    else return 0;
  else if(0 != data[21]) return 4;
  else return -1; }
```

260 seconds, and **true alarm**!

(1) Initial Abstraction

```c
int process(int *data) {
  return nondet();
}
```

240 seconds, but false alarm

We could synthesize this abstraction using only 33 input-output examples!

Scalable & Imprecise

Precise & Scalable

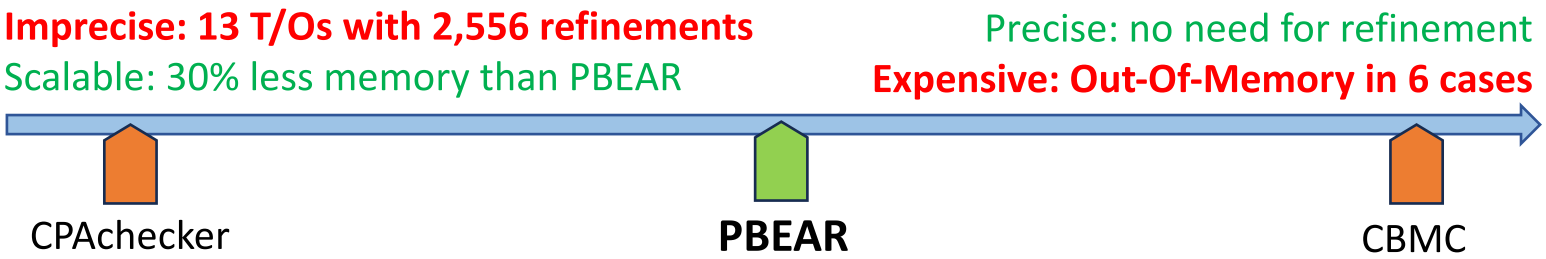
Precise & Expensive

Coarsest

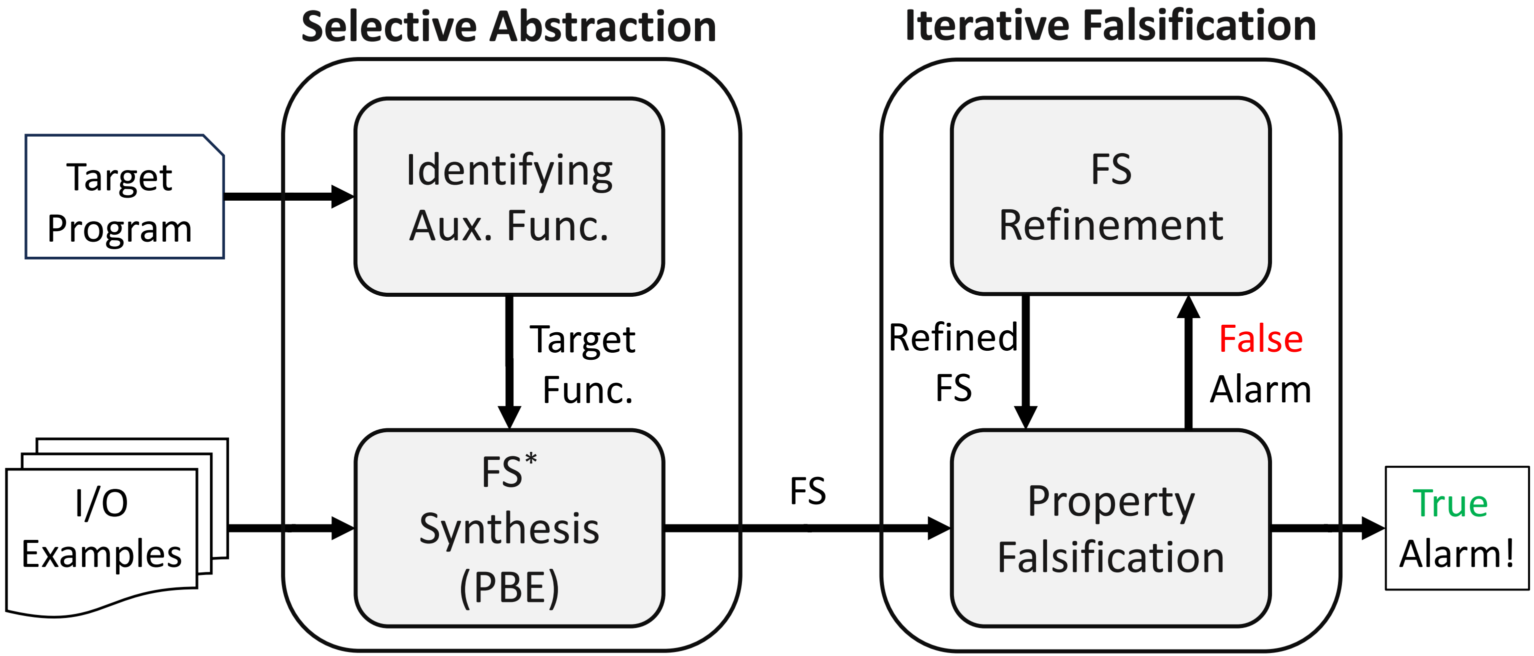
Concrete

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

- ✓ **PBEAR** suggests **selective abstraction** for efficient property falsification of embedded software
- ✓ **PBEAR** is **the first approach** utilizing **PBE** for more precise & scalable abstraction
- ✓ Out of 15 safety properties from three embedded software, **PBEAR** finds **5 to 12 more true alarms** than CBMC and CPAchecker



Overall Process of **PBEAR**



*FS: Function Summary

Problems & Solutions for Utilizing PBE

- **PBE** takes input/output (I/O) examples and produces a program that satisfies the given examples

Auxiliary Function

```
int gcd(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

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

PBE

Function Summary (FS)

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

1. Overfitting problems in PBE:

✓ **Iterative function summary synthesis**

2. PBE-based refinement may reproduce infeasible paths:

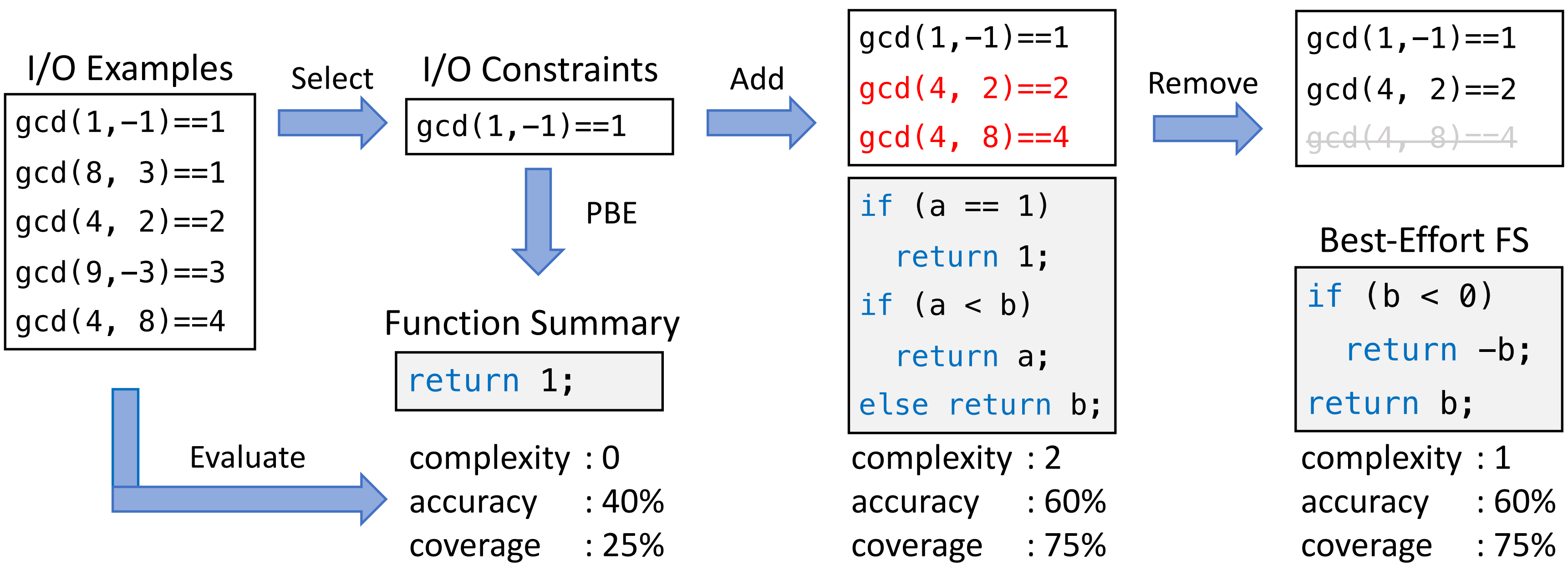
✓ **A novel PBE-based refinement with symbolic alarm filtering**

1. Overfitting ➔ Iterative FS Synthesis

Measurement:

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

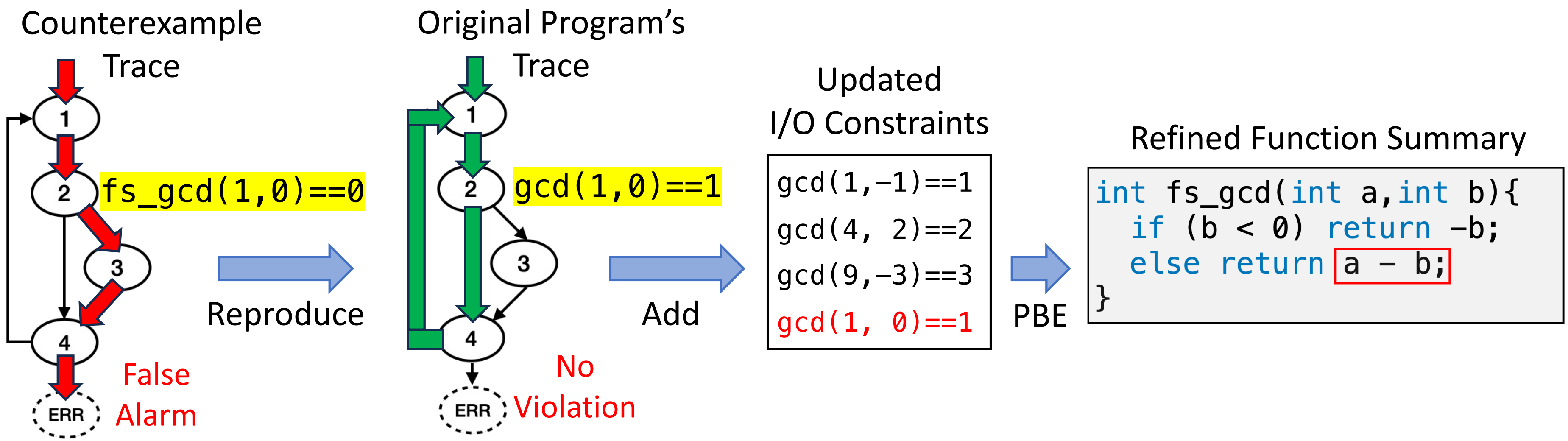
Synthesize until a given time budget (1 hour)



2. PBE Produces Infeasible Paths

Baseline of refinement approach:

1. Add new I/O constraints from the false alarm
2. Re-synthesize FS using the updated I/O constraints



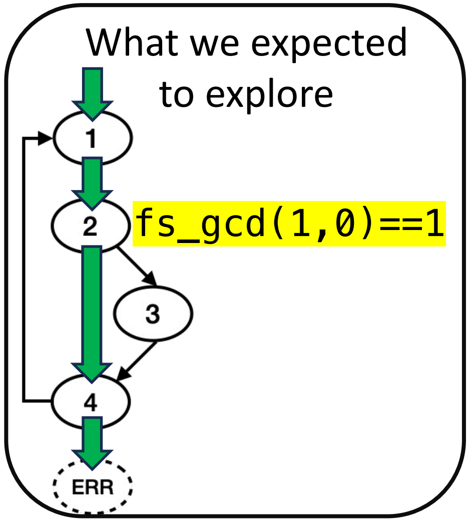
2. PBE **Re**produces Infeasible Paths

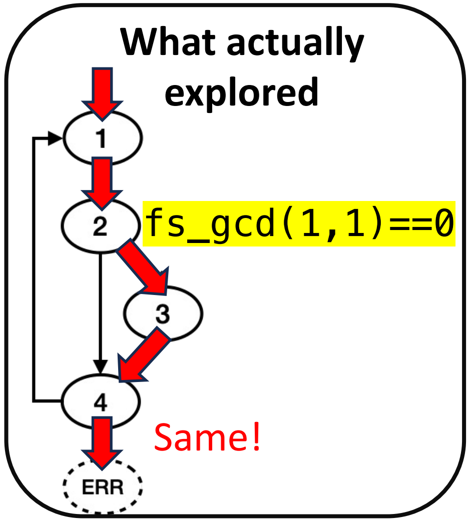
Problem: No guarantee for avoiding re-exploration of **previously identified infeasible paths**

Refined Function Summary

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

Falsify

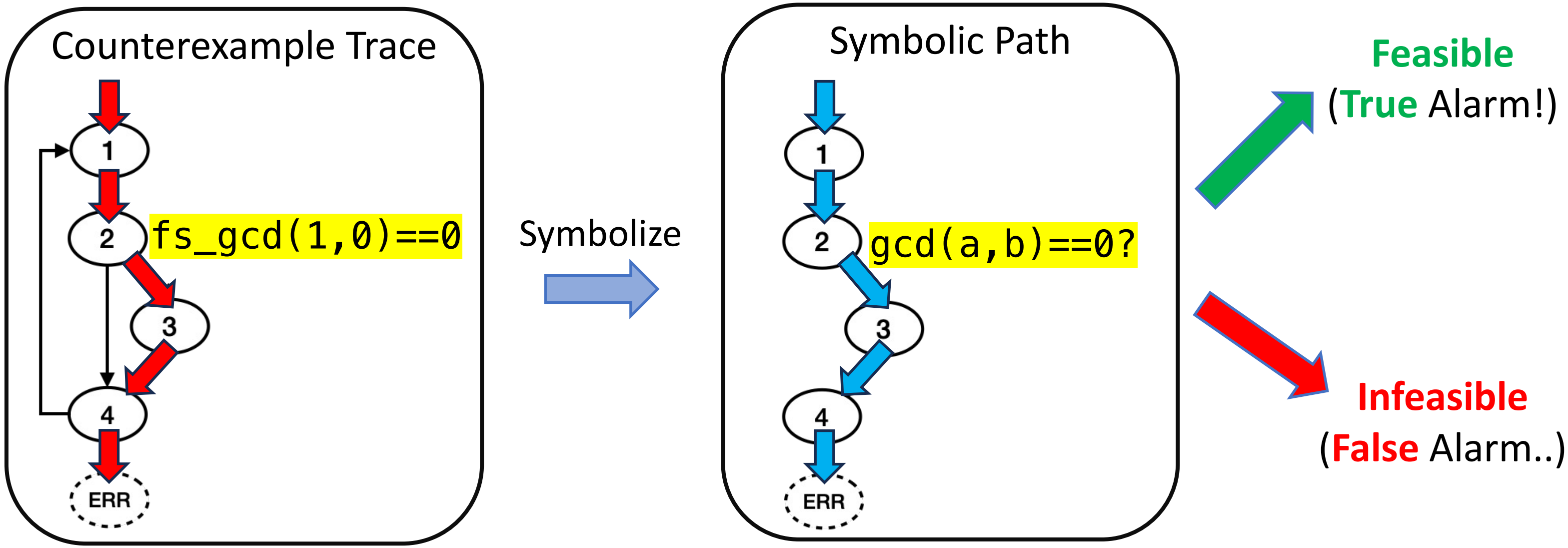




There is **another I/O example** leading to the same infeasible path

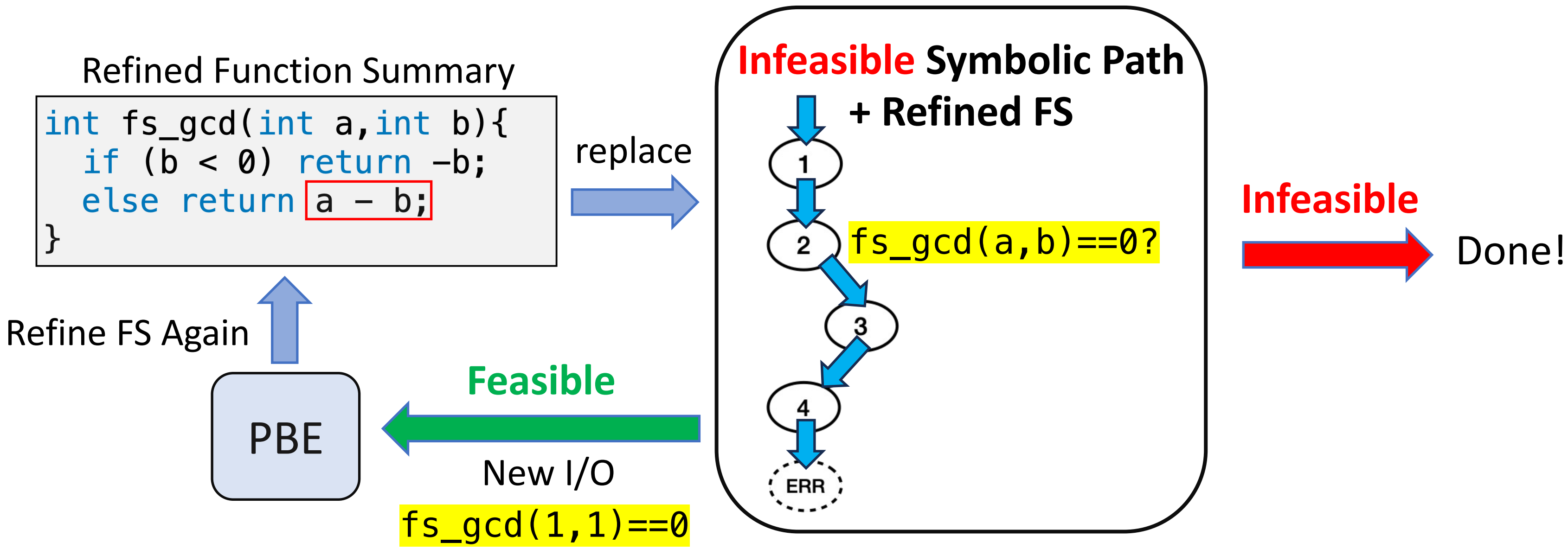
2. Solution: PBE-Based Refinement with Symbolic Alarm Filtering

1. Convert the counterexample trace into a **symbolic path**
2. Check the feasibility of the symbolic path using SAT solving

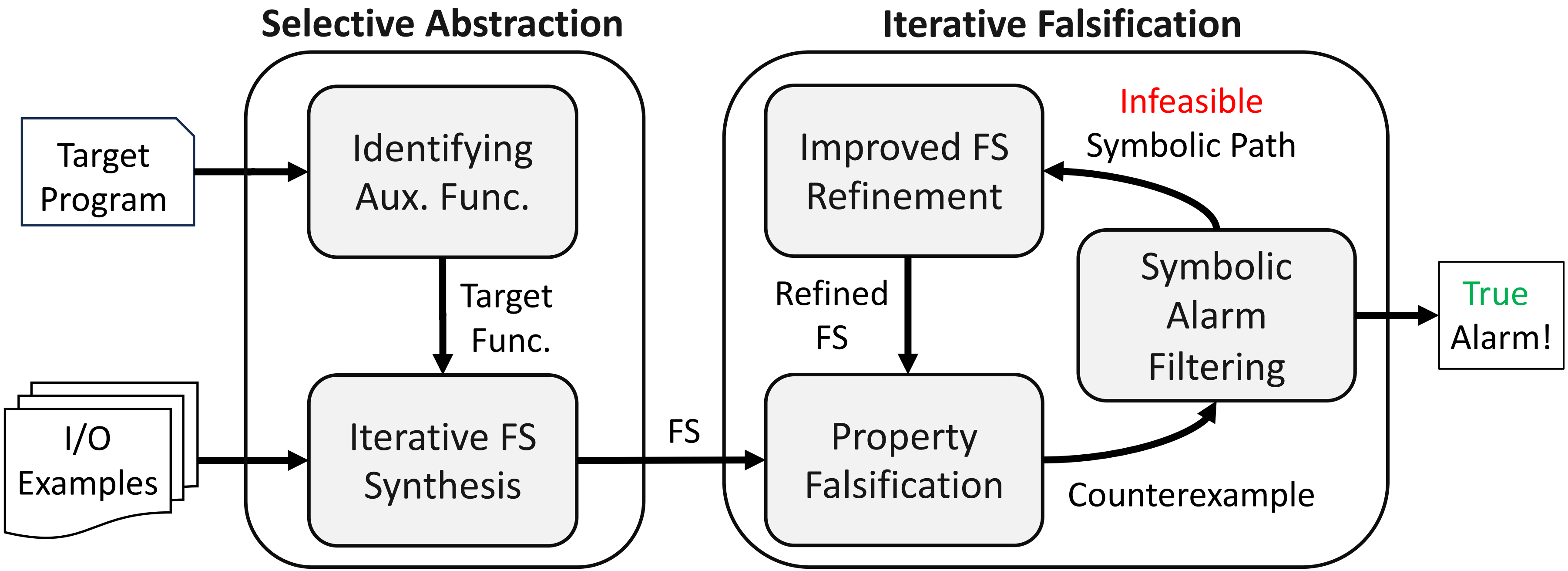


2. Solution: PBE-Based Refinement with Symbolic Alarm Filtering

3. **Iteratively refine FS using PBE until it does not make infeasible symbolic path feasible**



(Refined) Overall Process of **PBEAR**



Experiments

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

- RQ1: Performance of PBEAR vs. two state-of-the-art model checkers,
  - CBMC: bounded model checker
  - CPAchecker: predicate abstraction

- RQ2: Effectiveness of our improved FS refinement vs. PBEAR^{base}
  - PBEAR^{base}: baseline of refinement approach; no symbolic alarm filtering

* PBEAR uses DUET for function summary synthesis and CBMC for property falsification

RQ1: Performance of PBEAR (on SV-COMP)

Tools	# 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 these programs
- **PBEAR** showed competitive performance on **general-purpose programs**

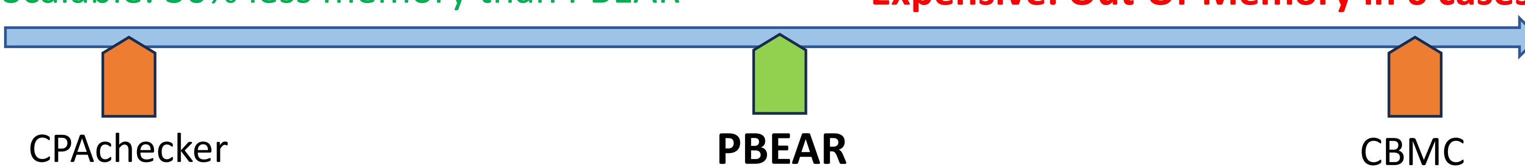
RQ1: Performance of PBEAR (on Embedded Software)

Tools	# 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: 72 hours; 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

Tools	# 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^{base}

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

Conclusion

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

- Future Work
  - PBEAR is highly dependent on the performance & limitation of PBE
    - We may need to combine PBEAR with other abstraction techniques
  - We plan to further investigate the applicability of PBEAR to:
    - Programs in other domains with control-oriented structures