Verifying Recursive Programs using Verifiers for Non-recursive Programs

Project Progress Report

謝橋, GIEE, NTU

Jan. 17, 2014

Outline

- Introduction
- Overview
- Experiment Result
- Discussion

Introduction

From WAVAS

Background & Our Goal

- Established Tools
  - Predator, UFO, CPAChecker, Blast, etc.
  - Handle non-recursive programs efficiently
  - Limited support on recursion for most verifiers
- Light-weight extensions to handle recursion
  - Based on established verifiers
  - Minimize efforts on implementation

Overview

From WAVAS

Sample

```
/* Original */
int main()
{
 if(y<0)
    y = rec(x);
 assert(y >= 0);
 return 0;
}
```

How to verify?

Sample

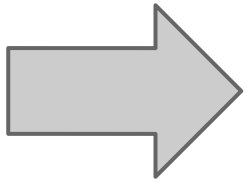
```
/* Original */
int main()
{
 if(y<0)
    y = rec(x);
 assert(y >= 0);
 return 0;
}
```

How to verify?
Find assertion violation that
rec() is not involved

Under-approximation

```
/* Original */
int main()
{
 if(y<0)
   y = rec(x);
 assert(y >= 0);
 return 0;
}
```

How to verify?
Find assertion violation that rec() is not involved



```
/* Under-approx. */
int main()
{
 if(y<0)
   /*y = rec(x);*/
   assume(false);
 assert(y >= 0);
 return 0;
}
```

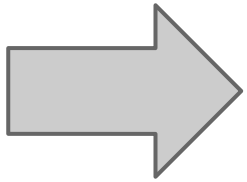
Consider executions without calling rec() only

Replace all function calls with assume(false)

Under-approximation~

```
/* Original */
int main()
{
 if(y<0)
   y = rec(x);
 assert(y >= 0);
 return 0;
}
```

How to verify?
Find assertion violation that rec() is not involved



```
/* Under-approx. */
int main()
{
 if(y<0)
   /*y = rec(x);*/
   assume(false);
 assert(y >= 0);
 return 0;
}
```

Consider executions without calling rec() only
What if all such executions are safe?

Sample ~ with assumption

```
/* Original */
int main()
{
 if(y<0)
    y = rec(x);
 assert(y >= 0);
 return 0;
}
```

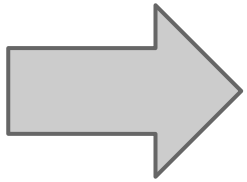
How to verify?

Assume we have

{true} y=rec(x) {y >= 0}

Over-approximation

```
/* Original */
int main()
{
 if(y<0)
   y = rec(x);
 assert(y >= 0);
 return 0;
}
```



```
/* Over-approx. */
int main()
{
 if(y<0)
   /*y = rec(x);*/
   assume(y>=0);
 assert(y >= 0);
 return 0;
} SAFE
```

How to verify?

Assume we have

{true} y=rec(x) {y >= 0}

Replace all function calls with assume(summary)

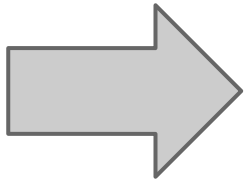
Over-approximation~

```
/* Original */
int main()
{
 if(y<0)
   y = rec(x);
 assert(y >= 0);
 return 0;
}
```

How to verify?

Assume we have

{true} y=rec(x) {y >= 0}



```
/* Over-approx. */
int main()
{
 if(y<0)
   /*y = rec(x);*/
   assume(y>=0);
 assert(y >= 0);
 return 0;
} SAFE
```

How to find reasonable function summaries?

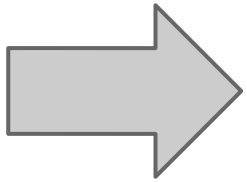
Over-approximation~~

```
/* Original */
int main()
{
 if(y<0)
   y = rec(x);
 assert(y >= 0);
 return 0;
}
```

How to verify?

Assume we have

{true} y=rec(x) {y >= 0}



```
/* Over-approx. */
int main()
{
 if(y<0)
   /*y = rec(x);*/
   assume(y>=0);
 assert(y >= 0);
 return 0;
} SAFE
```

How to find reasonable function summaries?

We guess

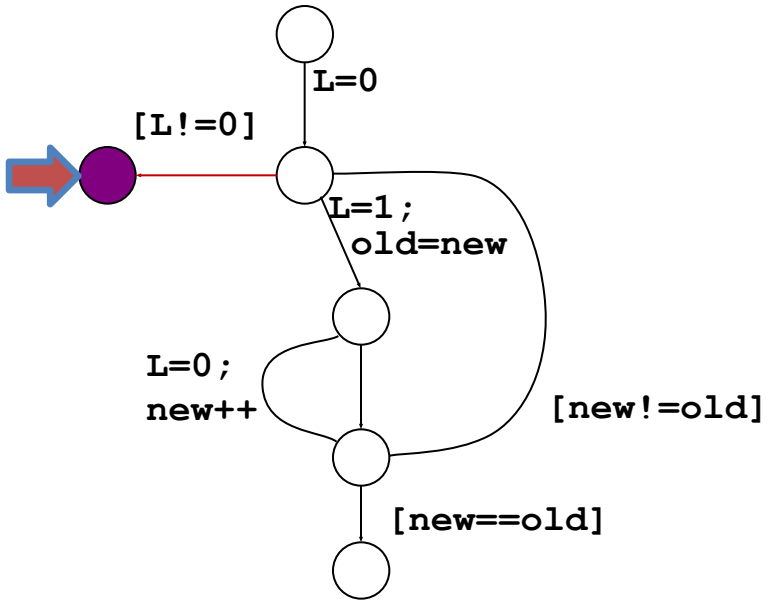
Guess Summary

Preliminary

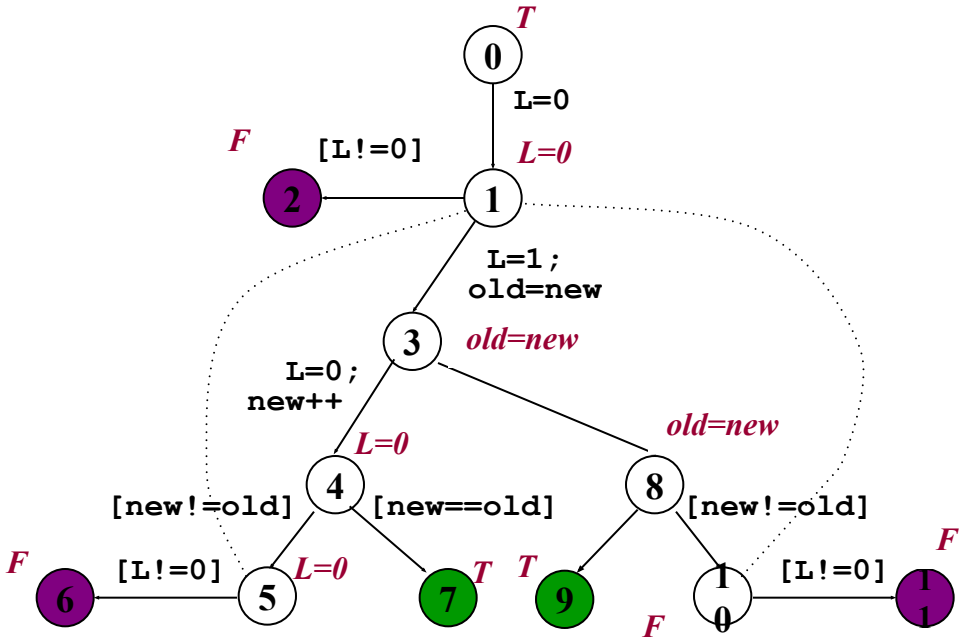
- Lazy Abstraction
  - Abstract Reachable Tree/Graph

- Handle Function Call
  - Unwind all function calls
  - Not work for recursive function
    - Recursive call remains
- Handle Recursive Function Call
  - Use aforementioned Under-approximations of recursive functions

Unwinding the CFG



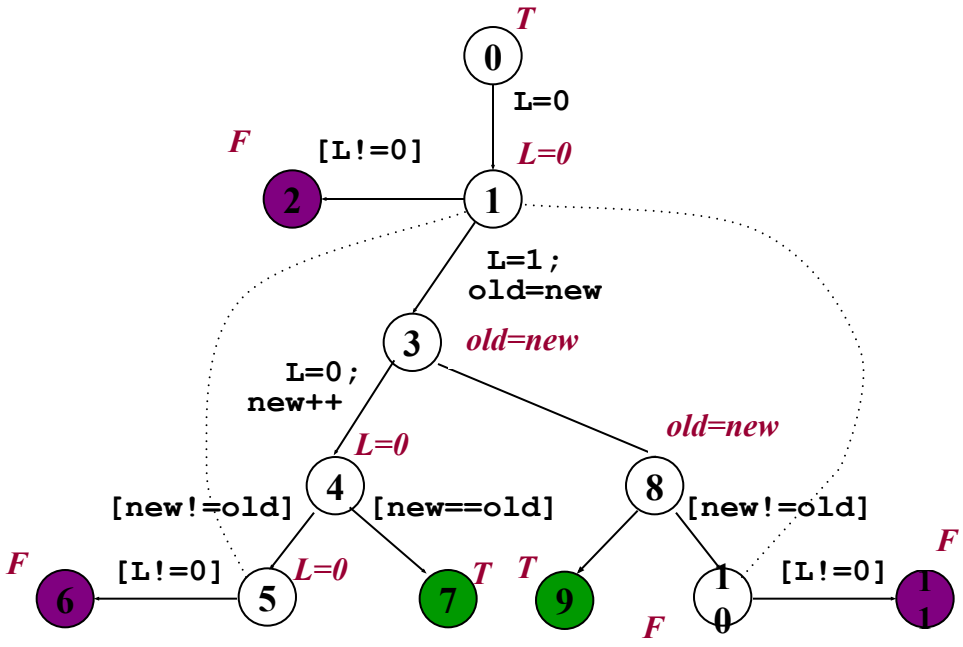
control-flow graph



Error Node Unreachable
SAFE and complete ARG

Unwinding the CFG

- Label on Node
  - Approximation of reachable states at that location
- Approxiamte a Function Call
  - Select labels at begin and end of a function call

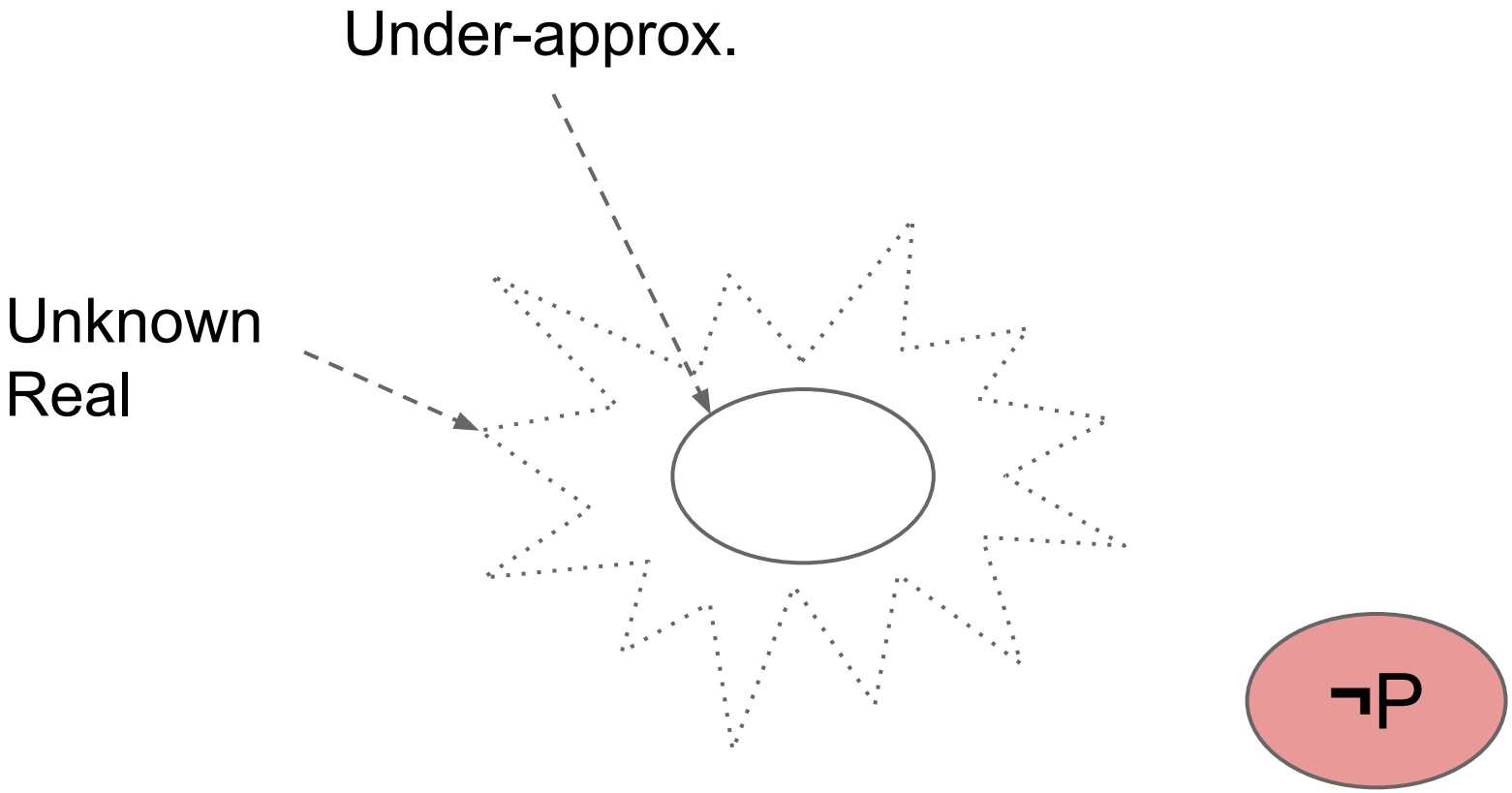


Error Node Unreachable
SAFE and complete ARG

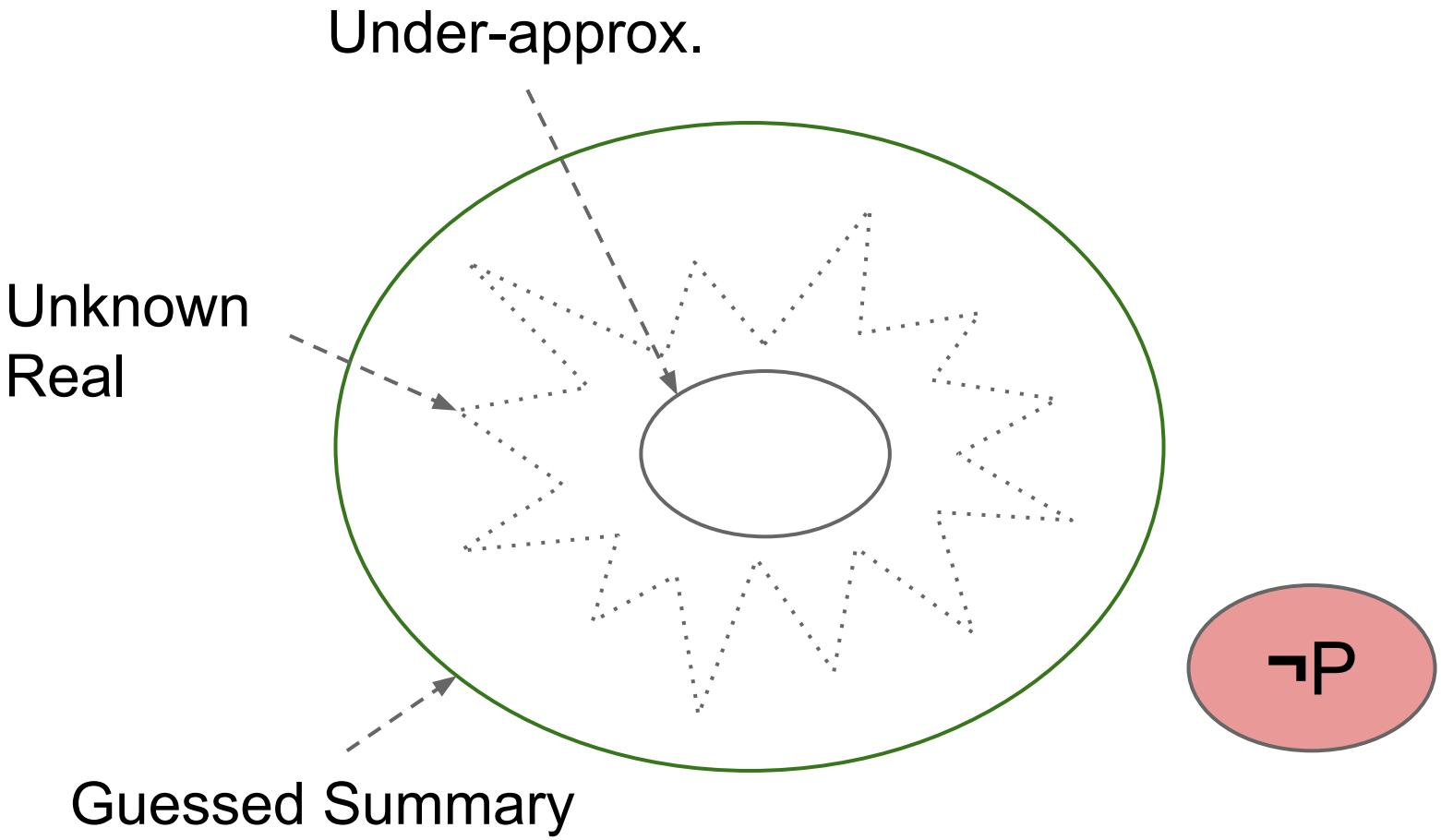
Generate Summary for function

- Pre-Condition *Pre-CON*
  - Given some labels mapped to begin of an unwinded function call, pre_1, …, pre_k
  - *Pre-CON* = $pre_1 \vee \ldots \vee pre_k$
- Post-Condition *Post-CON*
  - Given some labels mapped to end of the function call
    , $post_1$, …, $post_n$
  - *Post-CON* = $post_1 \vee \ldots \vee post_n$
- Summary *SUM*
  - *SUM* = (*Pre-CON* => *Post-CON*)
    = (¬*Pre-CON* ∨ *Post-CON*)

A SAFE design



Valid Guess



Question

- Summary is an over-approximation of ?
  - Overapproximate the unwinded function call
  - But we use under-approximation of recursive call.
  - What does it over-approximate?
    - Under-approximation
- Generated summary is a guess
  - How to validate?

Check
Guessed Summary

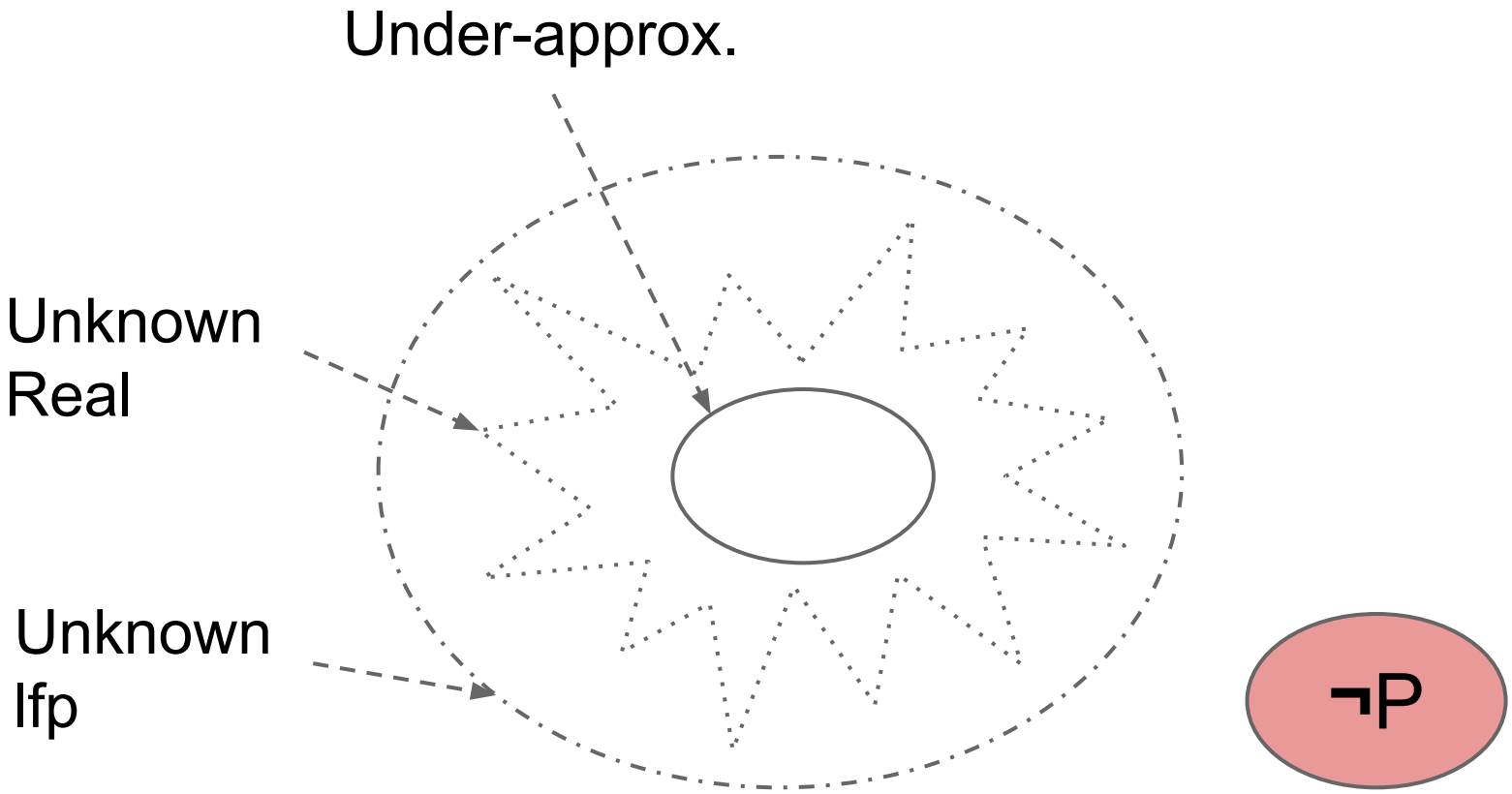
Preliminary

- Least Fixedpoint (lfp)
  - Tarski's fixed point theorem

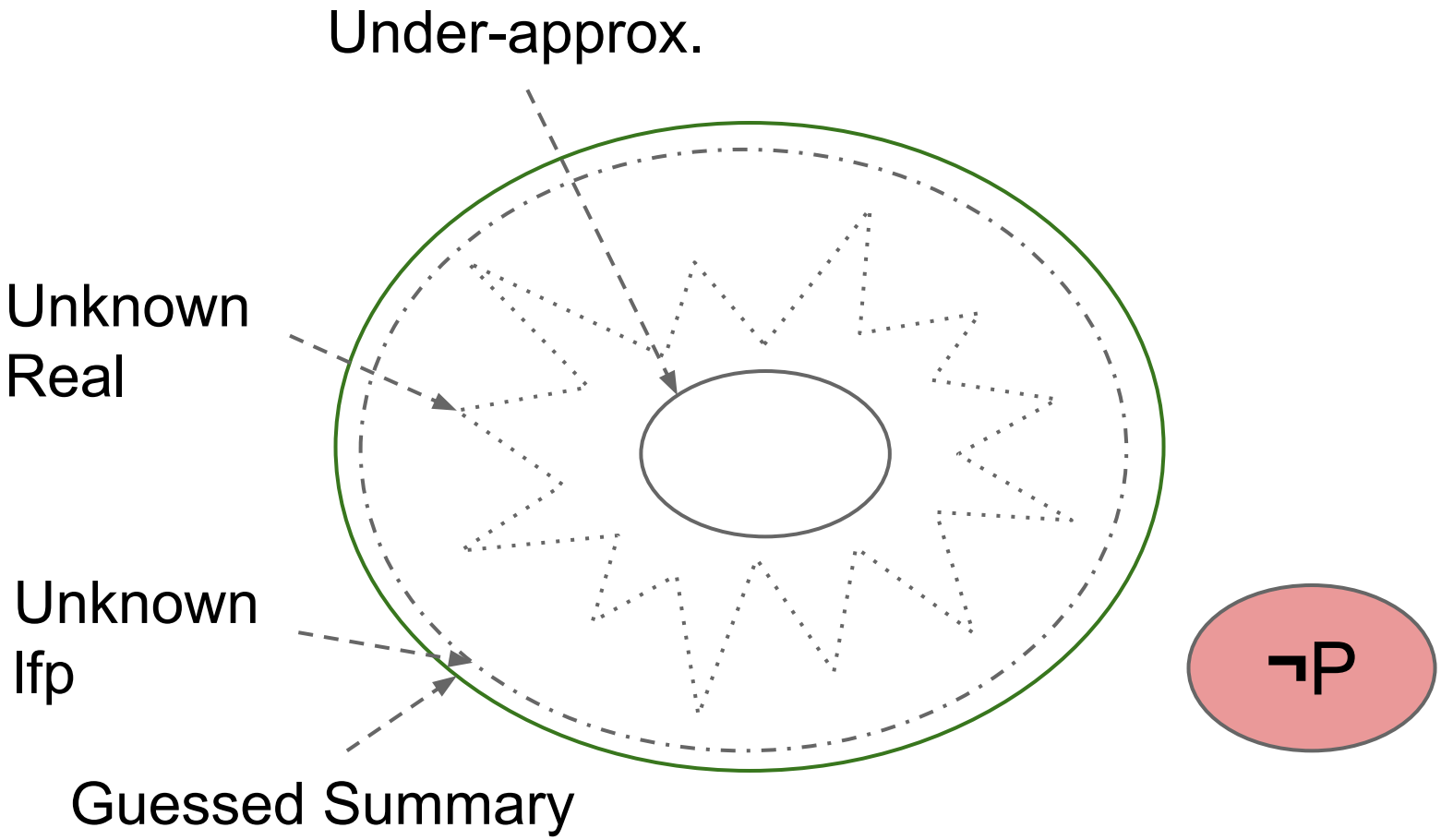
$$\bigwedge P = \bigwedge \{x \in L \mid x \geq f(x)\}$$

- Result
  - f as recursive function, x as guessed summary
  - If x contains f(x)
    - We develop a method to check this
  - x must contain least fixedpoint

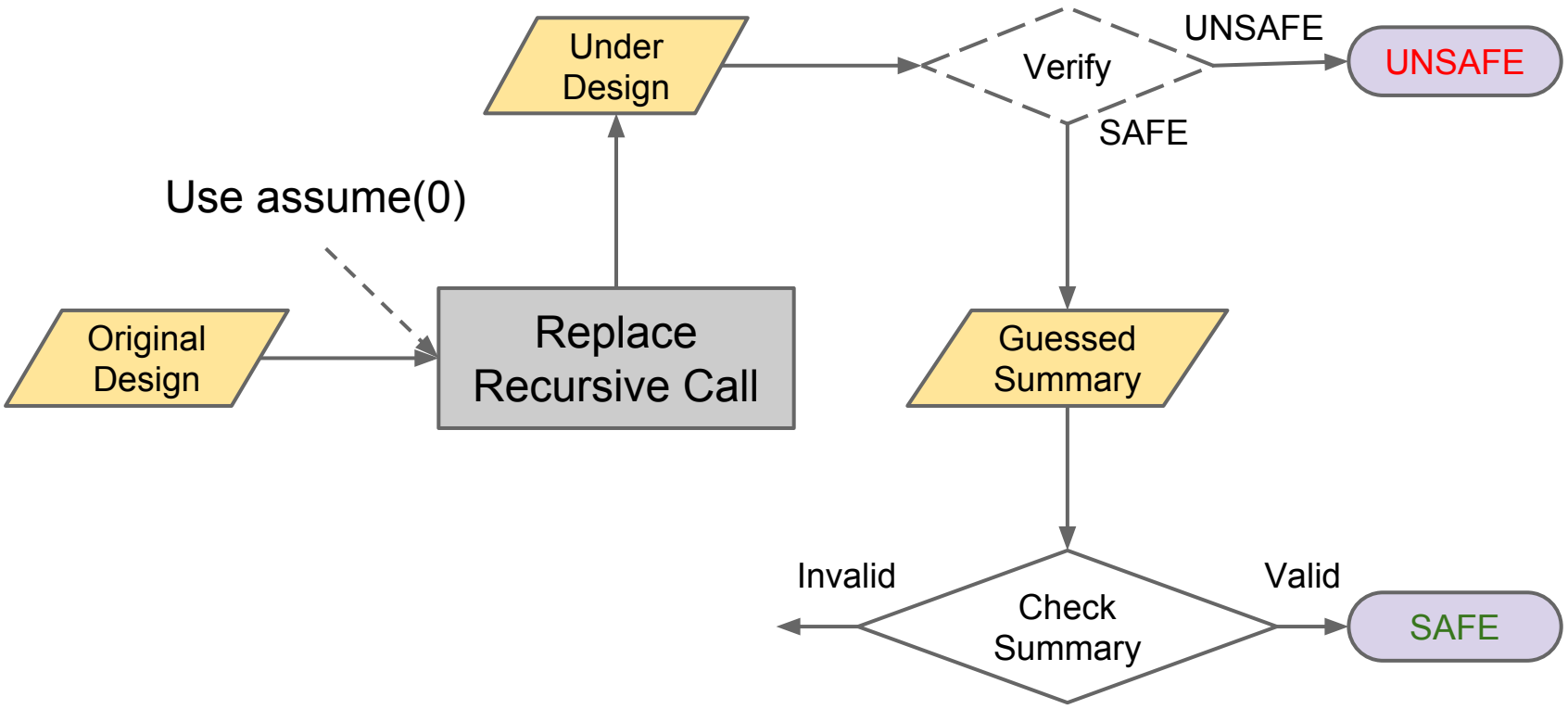
Least Fixedpoint for SAFE design



Valid Guess



Flow

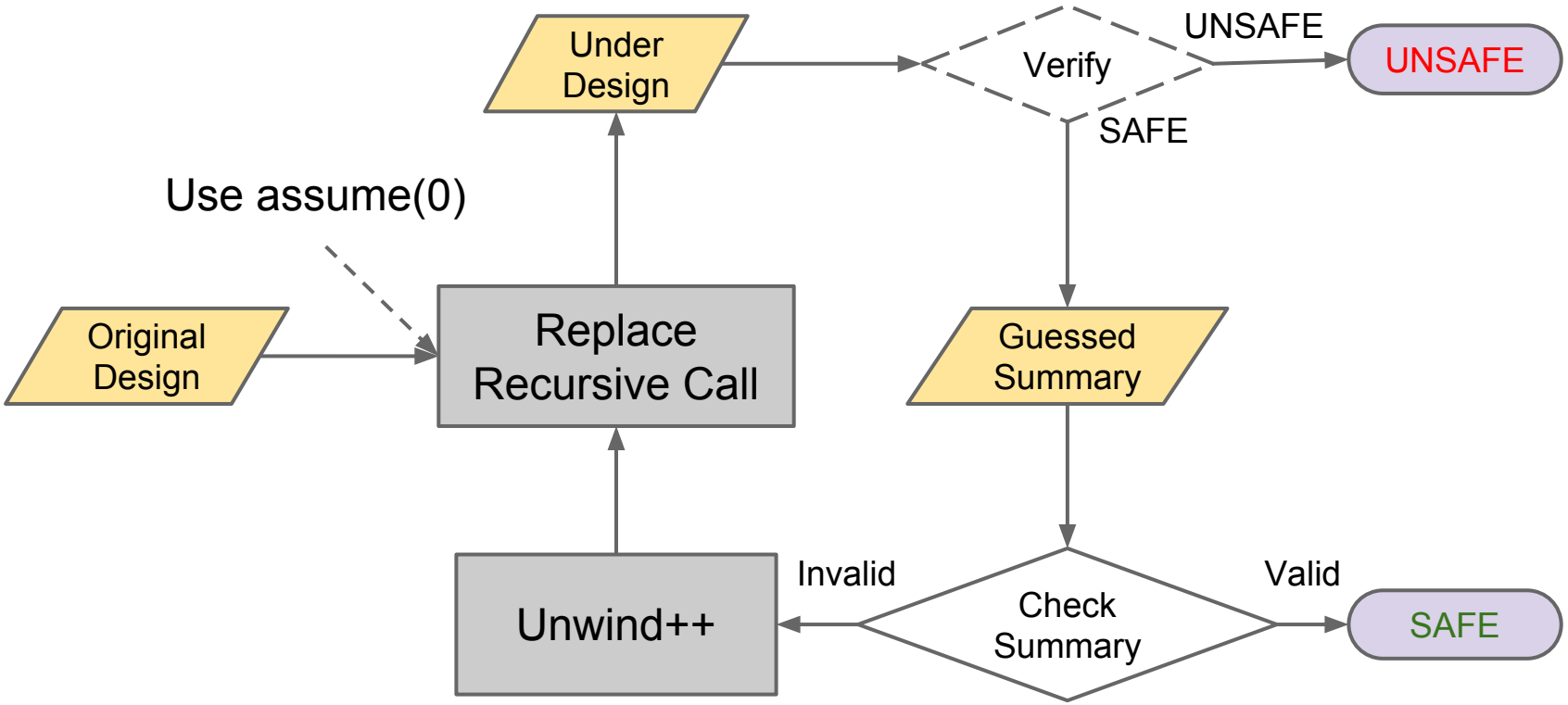

Under Design
Verify
UNSAFE
UNSAFE
SAFE
Use assume(0)
Original Design
Replace Recursive Call
Guessed Summary
Invalid
Check Summary
Valid
SAFE

Invalid Guessed Summary

- What if this guess is invalid?
- Guess another summary
  - From another under-approximation
  - Produced new under-approximation by further unwinding recursive call

- Refine current guess
  - Currently working on this

Flow


Under Design
Verify
UNSAFE
UNSAFE
SAFE
Use assume(0)
Original Design
Replace Recursive Call
Guessed Summary
Unwind++
Invalid
Check Summary
Valid
SAFE

Experiment Result

- Benchmarks from SVCOMP 14
  - Category “recursive”
  - 23 cases in total with each case an assertion
  - Timeout: 900 sec. for a case
- Second Place Winner, Ultimate
  - Because 1st place winner is a little controversial
  - Solved: 8 cases
- Our Result
  - Solved: 10 cases
  - Solved after timeout: 2

Discussion

- Correctness
  - For UNSAFE design
    - A bug must be found after enough unwinding.
  - For SAFE design
    - Find a summary for recursive function
    - Provide a check method to validate its correctness
- Effectiveness on SAFE design
  - Assertion is usually an over-approximation of the recursive function being verified.
  - Our approach will frequently choose the assertion as our guessed summary.

Discussion~

- Inefficiency on UNSAFE design
  - Depend on unwinding enough times
  - Cause exponential growth of verified design size

- Case Study on unsolved cases
  - Assertion is a nonlinear property
    - Limited by the capability of chosen verifier

Thank you.