Counterexample Guided Inference of Modular Specifications

William T. Hallahan*

Ranjit Jhala†

Ruzica Piskac*

*Yale University

†UCSD

Verification

```
map :: (a -> b) -> xs: [a] -> { ys:[b] | size xs == size ys }
map f [] = []
map f (x:xs) = f x:map f xs
```

Modular Verification

```
add2 :: x:Int -> { y:Int | y == x + 2 }
add2 x = incr (incr x)
```

Modular Verification

```
add2 :: x:Int -> { y:Int | y == x + 2 }
add2 x = incr (incr x)
```

```
incr :: x:Int -> { y:Int | y > x }
```

To verify a caller, modular verifiers use callee's specification

Modular Verification

```
add2 :: x:Int -> { y:Int | y == x + 2 }
add2 x = incr (incr x)
```

```
incr :: x:Int -> { y:Int | y == x + 1}
incr x = x + 1
```

To verify a caller, modular verifiers use callee’s specification

Modular Verification

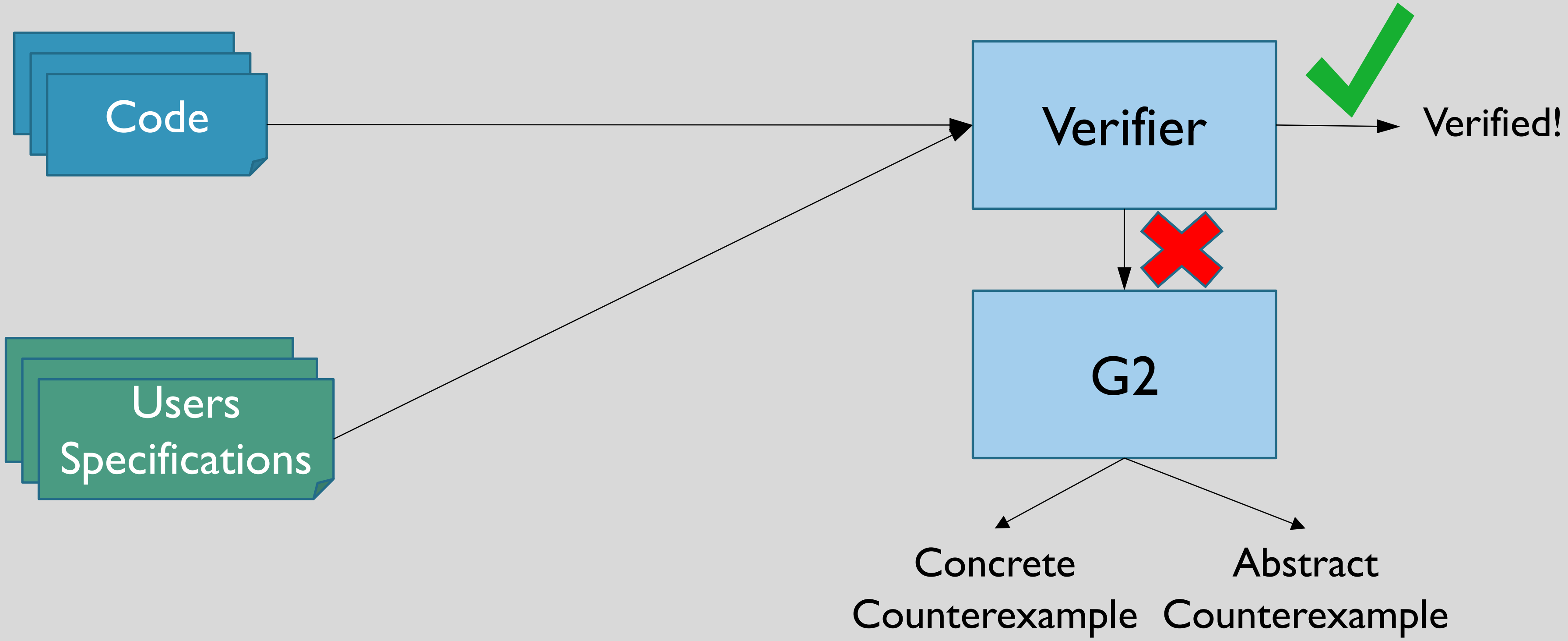
```
concat :: x:[[a]] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] ->   z:[a]
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys  = x:app xs ys
```

To verify a caller, modular verifiers use callee’s specification

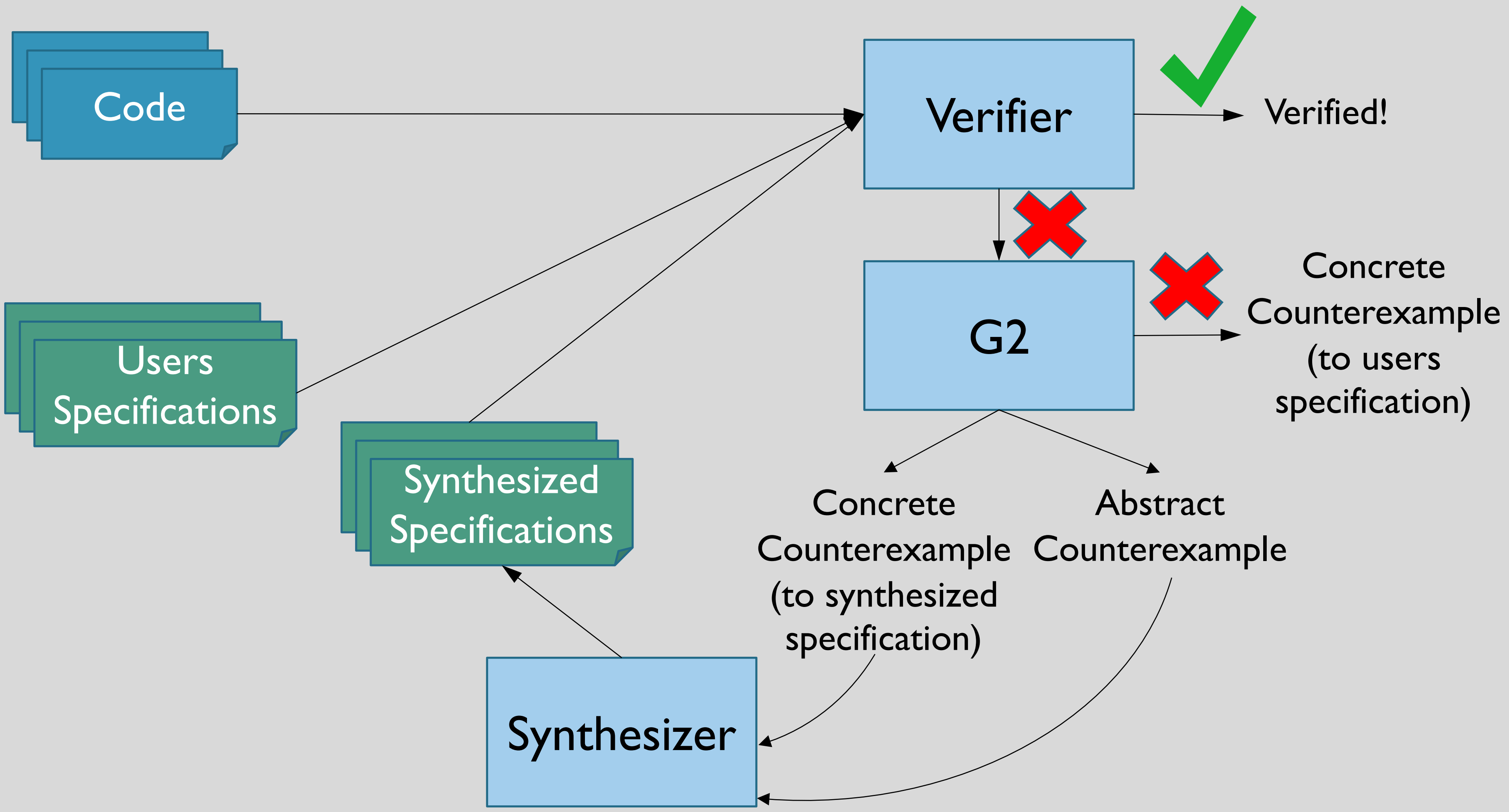
Question: How can we automatically find the required specifications?

Overview

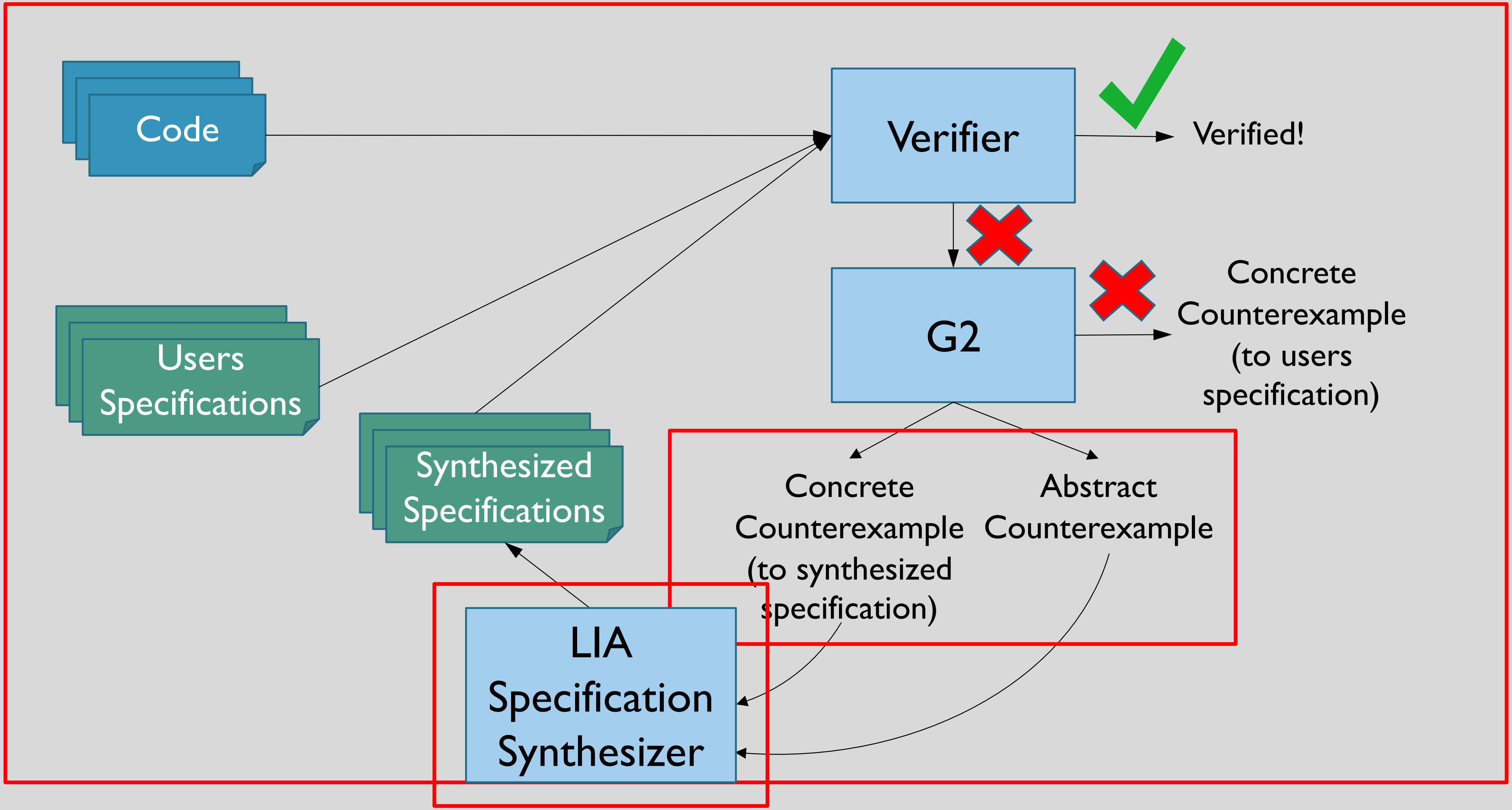


William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. *Lazy Counterfactual Symbolic Execution.* PLDI 2019.

Overview

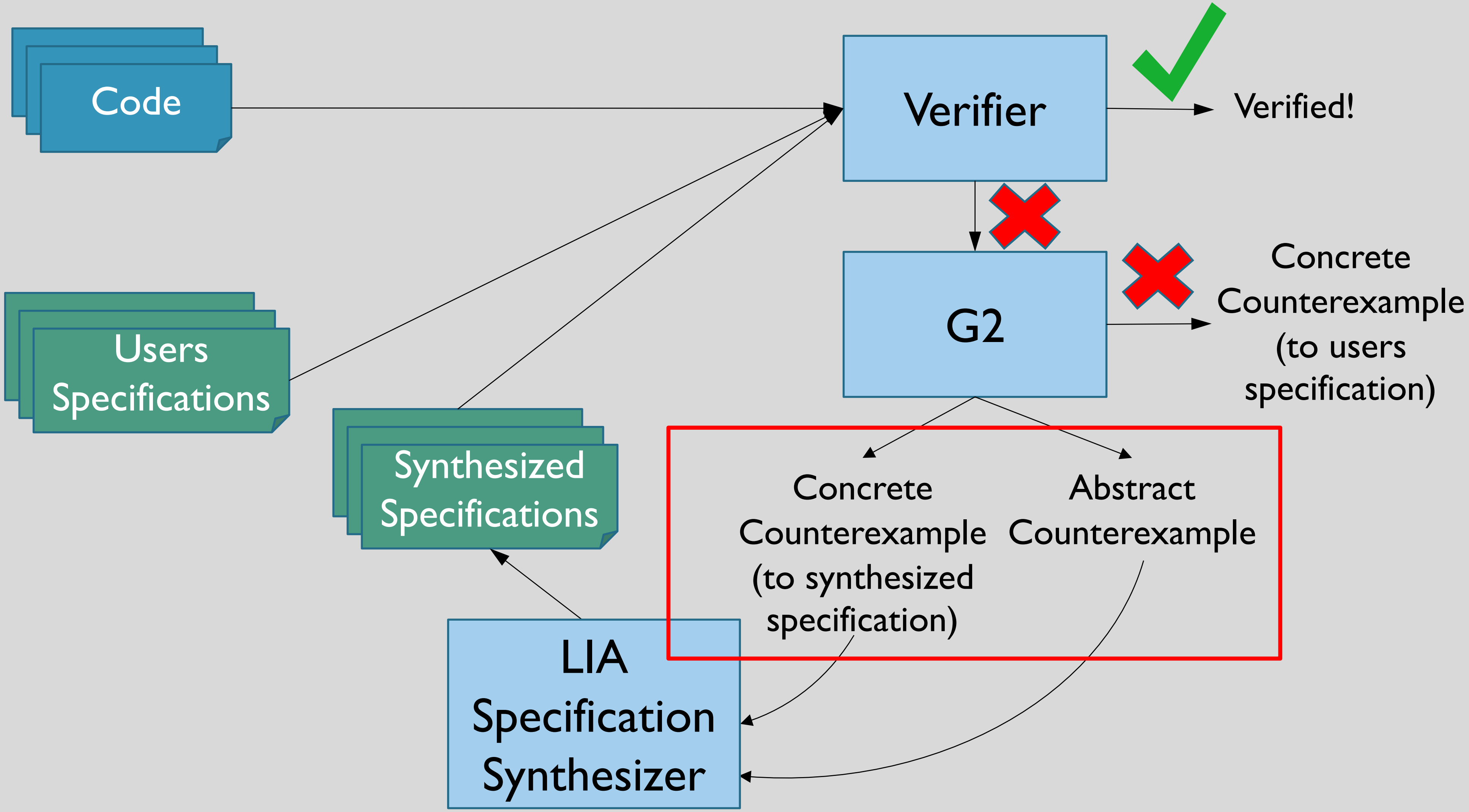

Code
Verifier
Verified!
Users Specifications
Synthesized Specifications
G2
Concrete Counterexample (to users specification)
Concrete Counterexample (to synthesized specification)
Abstract Counterexample
Synthesizer

Overview



Code
Verifier
Verified!
Users Specifications
G2
Concrete Counterexample (to users specification)
Synthesized Specifications
Concrete Counterexample (to synthesized specification)
Abstract Counterexample
LIA Specification Synthesizer

Overview



Code
Verifier
Verified!
Users Specifications
Synthesized Specifications
G2
Concrete Counterexample (to users specification)
Concrete Counterexample (to synthesized specification)
Abstract Counterexample
LIA Specification Synthesizer

Counterexamples

Concrete Counterexample

```
map :: (a -> b) -> xs:[a] -> { ys:[b] | size xs == size ys}
map f [] = []
map f (x:xs) = map f xs
```

map id [1] = []

Abstract Counterexample

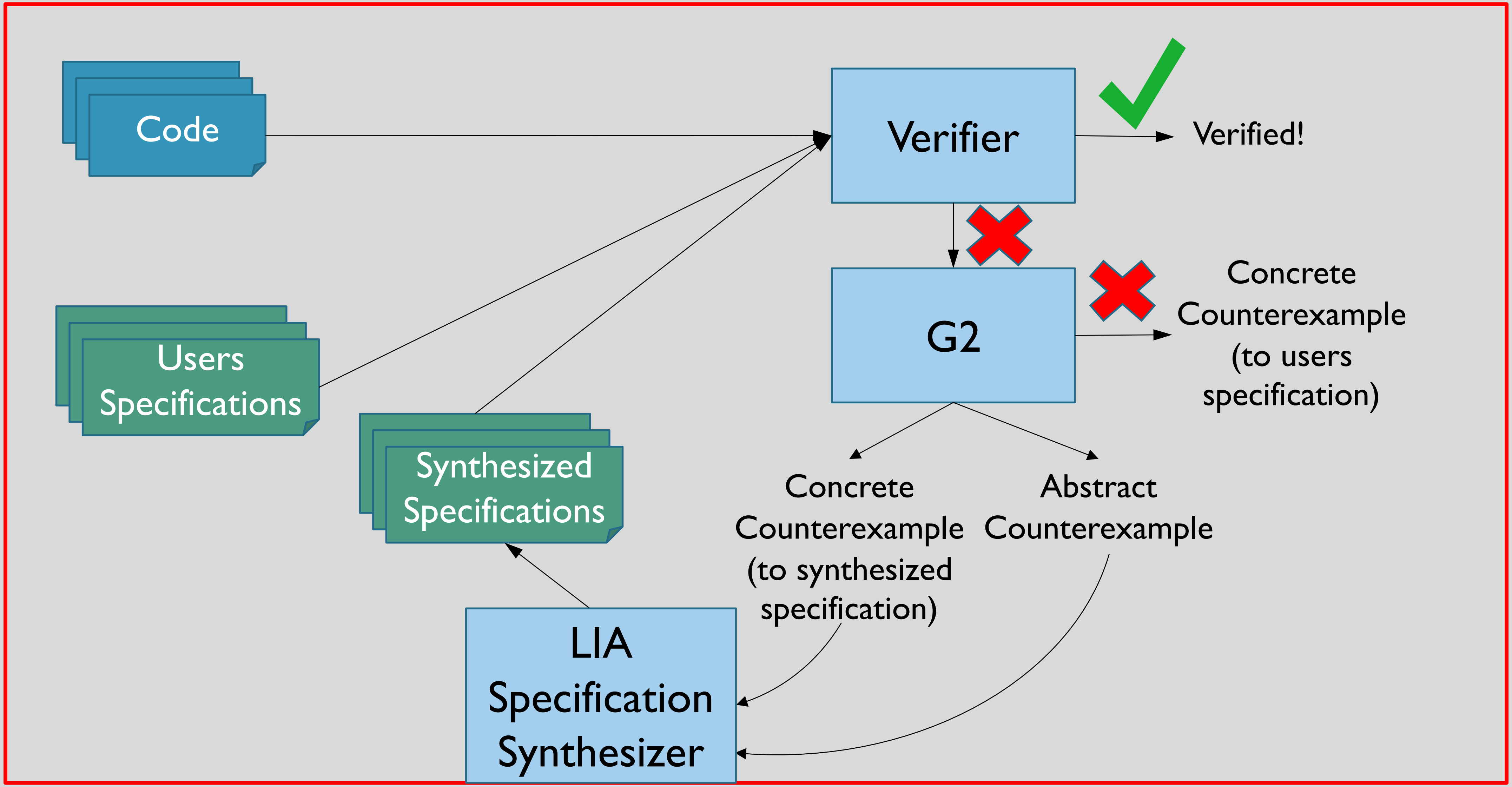
```
add2 :: x:Int -> { y:Int | y == x + 2 }
add2 x = incr (incr x)

incr :: x:Int -> { y:Int | y > x }
incr x = x + 1
```

add2 0 = 3
if incr 0 = 2

William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. *Lazy Counterfactual Symbolic Execution*. PLDI 2019.

Overview



Code
Users Specifications
Synthesized Specifications
Verifier
Verified!
G2
Concrete Counterexample (to users specification)
Concrete Counterexample (to synthesized specification)
Abstract Counterexample
LIA Specification Synthesizer

Example

```
concat :: x:[[a]] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] ->   z:[a]
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys  = x:app xs ys
```

Abstract counterexample:

```
concat [[], []] = [0]
if app [] [] = [0]
```

Real evaluation:

```
app [] [] = []
```

Synthesis constraints:

$pre_{app}([], []) \Rightarrow \neg post_{app}([], [], [0])$

$pre_{app}([], []) \Rightarrow post_{app}([], [], [])$

Example

```
concat :: x:[[a]] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] -> { z:[a] | size z == 0 }
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys  = x:app xs ys
```

Abstract counterexample:

```
concat [[], []] = [0]
if app [] [] = [0]
```

Real evaluation:

```
app [] [] = []
```

Synthesis constraints:

$pre_{app}([], []) \Rightarrow \neg post_{app}([], [], [0])$

$pre_{app}([], []) \Rightarrow post_{app}([], [], [])$

Example

```
concat :: x:[[a]] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] -> { z:[a] | size z == 0 }
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys  = x:app xs ys
```

Concrete counterexample:

```
app [0] [] = [0]
```

Synthesis constraints:

$pre_{app}([], []) \Rightarrow \neg post_{app}([], [], [0])$

$pre_{app}([], []) \Rightarrow post_{app}([], [], [])$

$pre_{app}([0], []) \Rightarrow post_{app}([0], [], [0])$

Example

```
concat :: x:[[a]] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] -> { z:[a] | size z == size x }
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys  = x:app xs ys
```

Concrete counterexample:

app [0] [] = [0]

Synthesis constraints:

$pre_{app}([], []) \Rightarrow \neg post_{app}([], [], [0])$

$pre_{app}([], []) \Rightarrow post_{app}([], [], [])$

$pre_{app}([0], []) \Rightarrow post_{app}([0], [], [0])$

Example

```
concat :: x:[[a]] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] -> { z:[a] | size z == size x }
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys  = x:app xs ys
```

Concrete counterexample:

app [0] [0] = [0, 0]

Synthesis constraints:

$pre_{app}([], []) \Rightarrow \neg post_{app}([], [], [0])$

$pre_{app}([], []) \Rightarrow post_{app}([], [], [])$

$pre_{app}([0], []) \Rightarrow post_{app}([0], [], [0])$

$pre_{app}([0], [0]) \Rightarrow post_{app}([0], [0], [0])$

Example

```
concat :: x:[[a]] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] -> { z:[a] | size z == size x + size y}
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys  = x:app xs ys
```

Concrete counterexample:

app [0] [0] = [0, 0]

Synthesis constraints:

$pre_{app}([], []) \Rightarrow \neg post_{app}([], [], [0])$

$pre_{app}([], []) \Rightarrow post_{app}([], [], [])$

$pre_{app}([0], []) \Rightarrow post_{app}([0], [], [0])$

$pre_{app}([0], [0]) \Rightarrow post_{app}([0], [0], [0])$

Example

```
concat :: x:[[a]] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] -> { z:[a] | size z == size x + size y}
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys  = x:app xs ys
```

✓

Synthesis constraints:

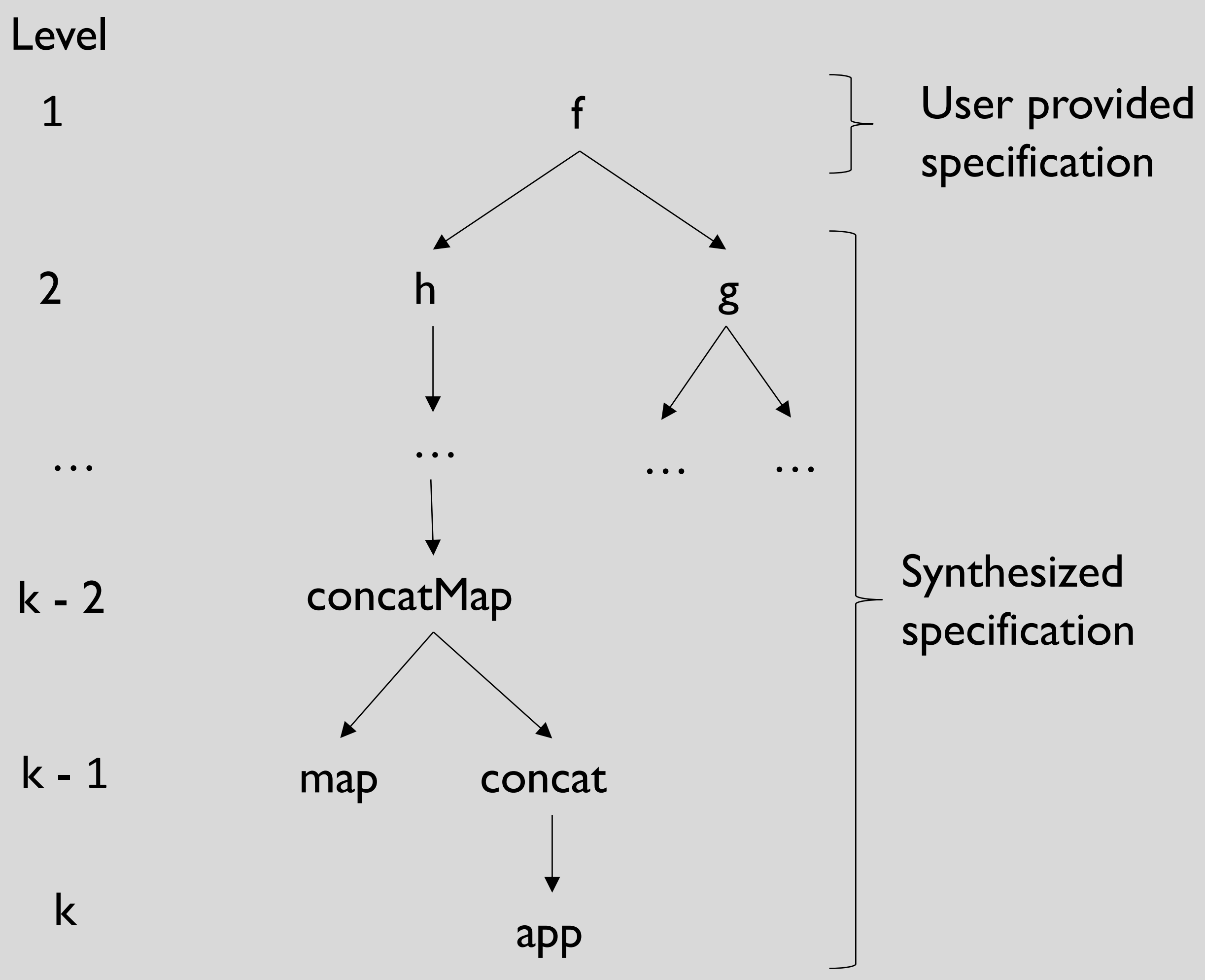
$pre_{app}([], []) \Rightarrow \neg post_{app}([], [], [0])$

$pre_{app}([], []) \Rightarrow post_{app}([], [], [])$

$pre_{app}([0], []) \Rightarrow post_{app}([0], [], [0])$

$pre_{app}([0], [0]) \Rightarrow post_{app}([0], [0], [0])$

Call Graph Traversal



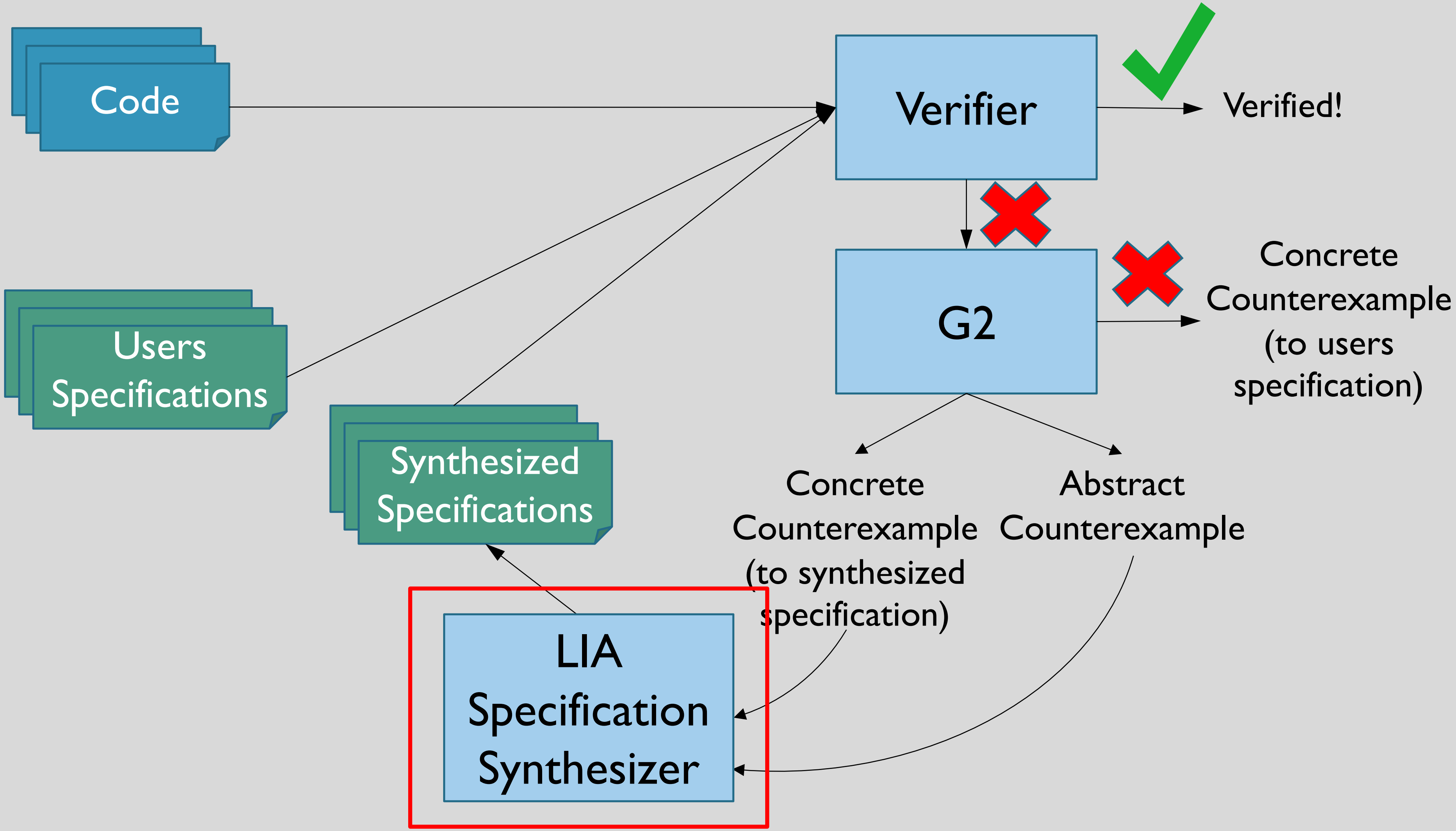
Walk **down** the call graph, from level 1 to level k.

At level i, synthesize specifications for the functions at level i + 1 that **would** (if correct) prove specifications of functions at level i.

Backtrack if:

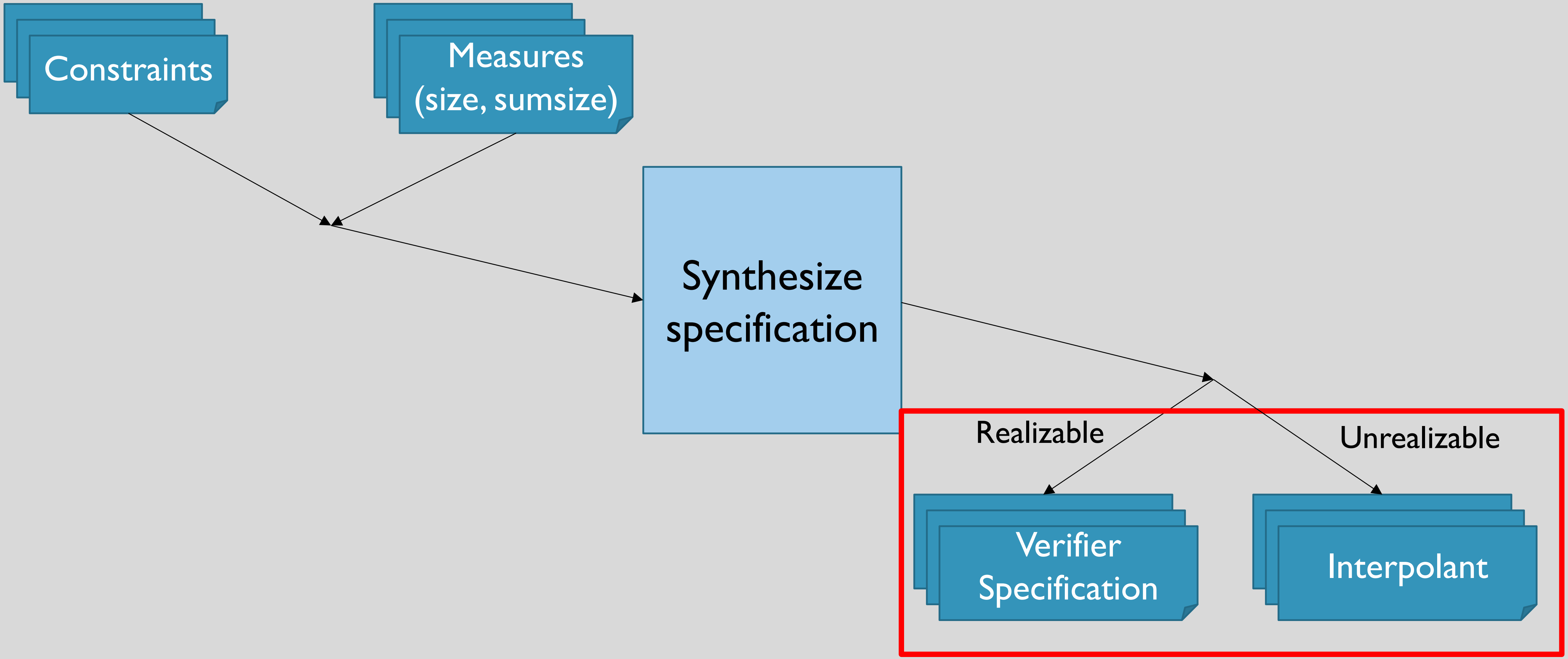
- a concrete counterexamples to a specification at level <= i is found
- specification synthesis problem becomes unrealizable

Overview

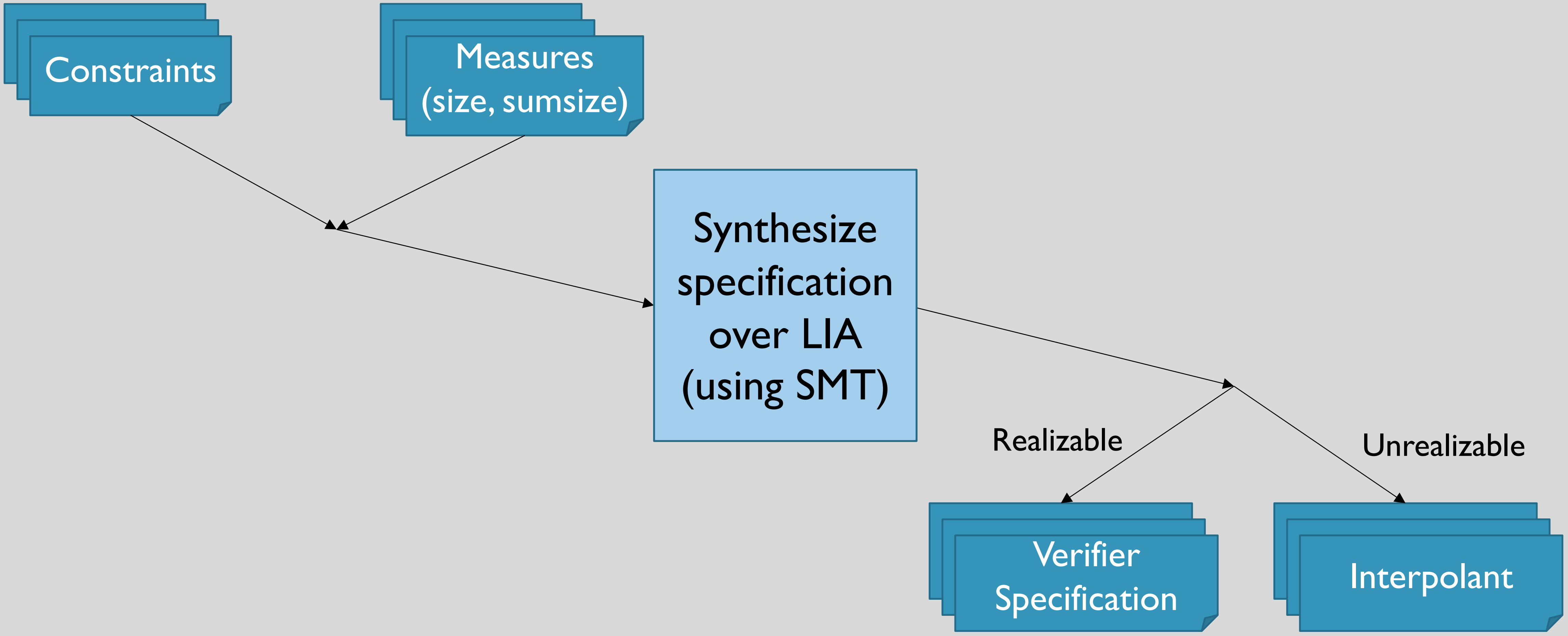


Code
Verifier
Verified!
Users Specifications
G2
Concrete Counterexample (to users specification)
Synthesized Specifications
Concrete Counterexample (to synthesized specification)
Abstract Counterexample
LIA Specification Synthesizer

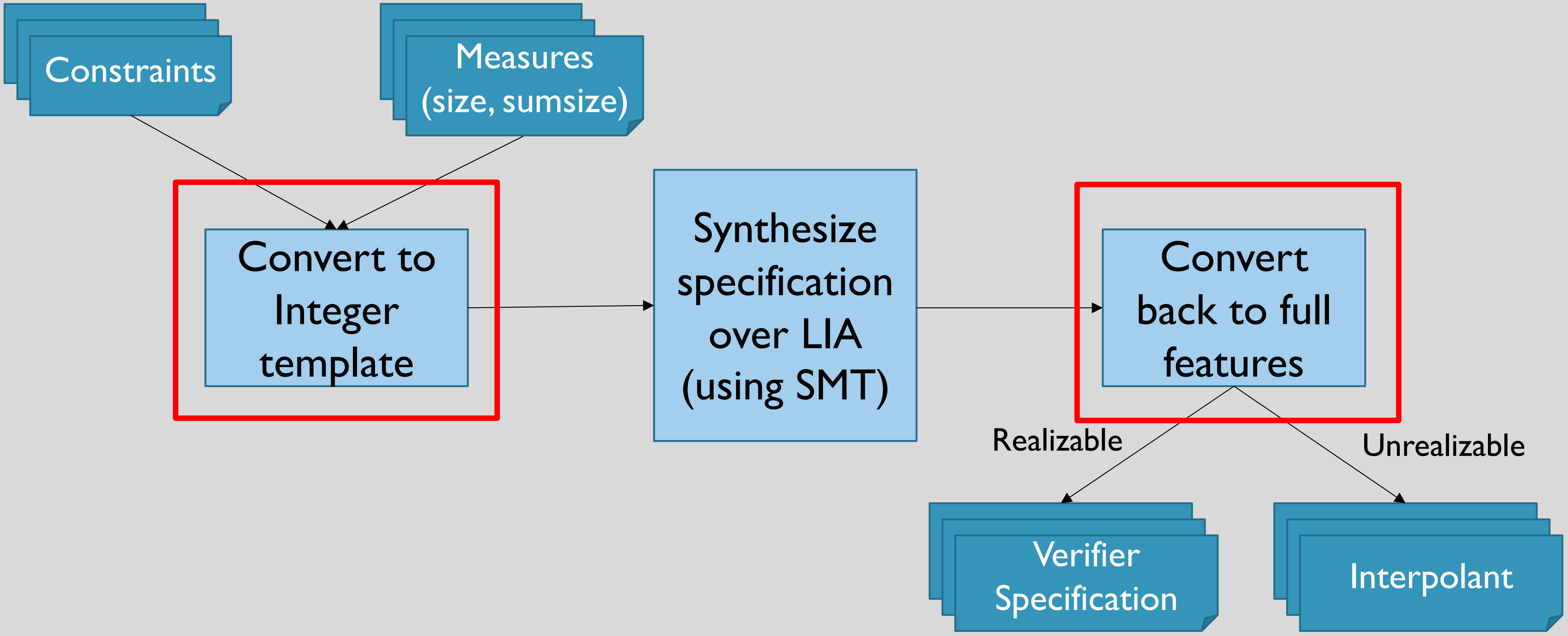
Synthesizer



Synthesizer



Synthesizer



Synthesizer

Synthesize LIA specifications for: f :: Int -> Int -> [Int]

f x y = [x + 4, y + 4]

Over:

Specification Type	Specification Example
Integer Literal Inputs/Outputs	f :: {x:Int | x < 0} -> { y:Int | y > 0} -> [Int]
Integer Measures	f :: Int -> Int -> { xs:[Int] | size xs > 0 } size :: [a] -> Int sumsize :: [[a]] -> Int
ADT Contents	f :: Int -> Int -> [{ x:Int | x > 0 }]

Conversion

Synthesize LIA specifications for: f :: Int -> Int -> [Int]

f x y = [x + 4, y + 4]

Constraint $pre_f(0, 1) \Rightarrow post_f(0, 1, [4, 5])$

Integer Measures

size [4, 5] = 2

$pre_f(0, 1) \Rightarrow post_f(0, 1, 2)$

$post_f(x, y, z) = z > 0$

$\downarrow$

$post_f(x, y, z) = \{ z:[a] \mid \text{size } z > 0 \}$

Conversion

Synthesize LIA specifications for: f :: Int -> Int -> [Int]

f x y = [x + 4, y + 4]

Constraint $pre_f(0, 1) \Rightarrow post_f(0, 1, [4, 5])$

$post_{f\_cons}(x, y, r)$

ADT Contents

$$pre_f(0, 1) \Rightarrow post_f(0, 1, 2) \wedge post_{f\_cons}(0, 1, 4) \wedge post_{f\_cons}(0, 1, 5)$$

[4, 5]

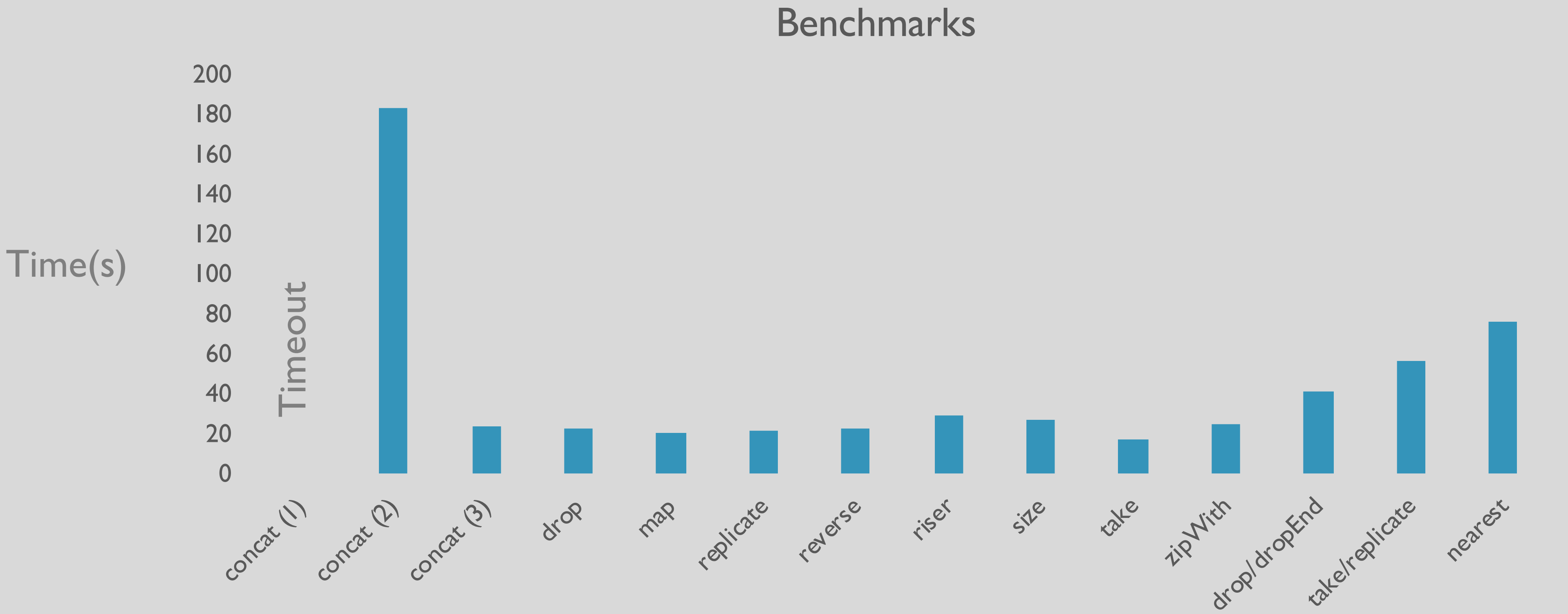
$post_f(x, y, r) = r > 0$

$post_{f\_cons}(x, y, r) = r > 2$

{ r:[{ x:Int | x > 2 }] | size r > 0 }

Evaluation

Ran the inference algorithm on 15 benchmarks, some created by us, some drawn from a graduate student level class homework assignment.



Largest benchmark is the inner loop of a kmeans implementation, involving 34 functions. We prove the codes specifications in 596 seconds (slightly under 10 minutes.)

Conclusion

- For verification to succeed, modular verifiers require specifications to not only be correct, but be sufficiently supported by callee's specifications.
- Given specifications written by the user, our inference algorithm **automatically** finds the required set of specifications for a modular verifier to succeed.
- Using an SMT solver to synthesizer LIA specifications allows us SyGuS like synthesis, but to also prove unrealizability and get interpolants.
- Our approach is implemented to find LiquidHaskell specifications, using G2 as a counterexamples generator, and it's effectiveness is demonstrated on a variety of benchmarks.