Synthesis in Uclid5

Federico Mora, Kevin Cheang, Elizabeth Polgreen, and Sanjit A. Seshia

Berkeley
UNIVERSITY OF CALIFORNIA

Verification Takes Time & Effort

Verification Takes Time & Effort

- For example, verifying SGX and Sanctum took four months of modelling work [1].

[1] Subramanyan, Pramod, et al. “A Formal Foundation for Secure Remote Execution of Enclaves.” CCS ‘17.

Verification Takes Time & Effort

- For example, verifying SGX and Sanctum took four months of modelling work [1].
- This work includes
  - strengthening invariants,
  - annotating functions with pre- and post-conditions, and
  - modelling system calls and the environment.
- Tasks like these can be automated using SyGuS!

[1] Subramanyan, Pramod, et al. “A Formal Foundation for Secure Remote Execution of Enclaves.” CCS ‘17.

Verification Takes Time & Effort

- For example, verifying SGX and Sanctum took four months of modelling work [1].
- This work includes
  - strengthening invariants,
  - annotating functions with pre- and post-conditions, and
  - modelling system calls and the environment.
- Tasks like these can be automated using SyGuS!

We use SyGuS solvers to unify synthesis-for-verification tasks in a clean way inside of Uclid5.

[1] Subramanyan, Pramod, et al. "A Formal Foundation for Secure Remote Execution of Enclaves." CCS '17.

Synthesis in Uclid5

1. Describe our integration of synthesis into Uclid5.

2. Generate a new set of SyGuS benchmarks.

Synthesis in Uclid5

1. Describe our integration of synthesis into Uclid5.
   - Users define functions to synthesize and use them
     - anywhere in their code,
     - for any verification technique
       - (k-induction, bounded model checking, …), and
     - for any kind of specification
       - (linear temporal logic, invariants, sequential assertions, …).

2. Generate a new set of SyGuS benchmarks.

Synthesis in Uclid5

1. Describe our integration of synthesis into Uclid5.
   - Users define functions to synthesize and use them
     - anywhere in their code,
     - for any verification technique
       - (k-induction, bounded model checking, …), and
     - for any kind of specification
       - (linear temporal logic, invariants, sequential assertions, …).
   - Uclid5 solves the queries using existing SyGuS engines.
2. Generate a new set of SyGuS benchmarks.

Related Work

Related Work

- Strengthening invariants (e.g. [2])
- Annotating functions with pre/post-conditions (e.g. [3])
- Modelling the environment and system calls (e.g. [4])

[2] Dillig, Isil, et al. “Inductive invariant generation via abductive inference.” OOPSLA ‘13.
[3] Padhi, Saswat, et al. “Data-driven precondition inference with learned features.” PLDI ‘16.
[4] Das, Ankush, et al. "Angelic verification: Precise verification modulo unknowns.” CAV ‘15.

Related Work

- Strengthening invariants (e.g. [2])
- Annotating functions with pre/post-conditions (e.g. [3])
- Modelling the environment and system calls (e.g. [4])
- Program sketching (e.g. [5])

[2] Dillig, Isil, et al. “Inductive invariant generation via abductive inference.” OOPSLA ‘13.
[3] Padhi, Saswat, et al. “Data-driven precondition inference with learned features.” PLDI ‘16.
[4] Das, Ankush, et al. "Angelic verification: Precise verification modulo unknowns.” CAV ‘15.
[5] Solar-Lezama, Armando, and Rastislav Bodik. Program synthesis by sketching. UC Berkeley PhD, 2008.

Related Work

- Strengthening invariants (e.g. [2])
- Annotating functions with pre/post-conditions (e.g. [3])
- Modelling the environment and system calls (e.g. [4])
- Program sketching (e.g. [5])
- Program repair (e.g. [6])

[2] Dillig, Isil, et al. “Inductive invariant generation via abductive inference.” OOPSLA ‘13.
[3] Padhi, Saswat, et al. “Data-driven precondition inference with learned features.” PLDI ‘16.
[4] Das, Ankush, et al. "Angelic verification: Precise verification modulo unknowns.” CAV ‘15.
[5] Solar-Lezama, Armando, and Rastislav Bodik. Program synthesis by sketching. UC Berkeley PhD, 2008.
[6] Le, Xuan-Bach D., et al. "S3: syntax-and semantic-guided repair synthesis via programming by examples." FSE ‘17.

Related Work

- Strengthening invariants (e.g. [2])
- Annotating functions with pre/post-conditions (e.g. [3])
- Modelling the environment and system calls (e.g. [4])
- Program sketching (e.g. [5])
- Program repair (e.g. [6])
- Verification engines with synthesis capabilities (e.g. [7])

[2] Dillig, Isil, et al. “Inductive invariant generation via abductive inference.” OOPSLA ‘13.
[3] Padhi, Saswat, et al. “Data-driven precondition inference with learned features.” PLDI ‘16.
[4] Das, Ankush, et al. "Angelic verification: Precise verification modulo unknowns.” CAV ‘15.
[5] Solar-Lezama, Armando, and Rastislav Bodik. Program synthesis by sketching. UC Berkeley PhD, 2008.
[6] Le, Xuan-Bach D., et al. "S3: syntax-and semantic-guided repair synthesis via programming by examples." FSE ‘17.
[7] Torlak, Emina, and Rastislav Bodik. "A lightweight symbolic virtual machine for solver-aided host languages." PLDI ‘14.

Running Example

Strengthening an Invariant

Uclid5 Fibonacci Module

```
module main {

    // Part 1: System description.
    var a, b : integer;
    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    // Part 2: System specification.
    invariant a_le_b: a <= b;

    // Part 3: Proof script.
    control {
        induction;
        check;
        print_results;
    }
}
```

Uclid5 Fibonacci Module

```
module main {

    // Part 1: System description.
    var a, b : integer;
    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    // Part 2: System specification.
    invariant a_le_b: a <= b;

    // Part 3: Proof script.
    control {
        induction;
        check;
        print_results;
    }
}
```

Uclid5 Fibonacci Module

```
module main {

    // Part 1: System description.
    var a, b : integer;
    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    // Part 2: System specification.
    invariant a_le_b: a <= b;

    // Part 3: Proof script.
    control {
        induction;
        check;
        print_results;
    }
}
```

Uclid5 Fibonacci Module

```
module main {

    // Part 1: System description.
    var a, b : integer;
    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    // Part 2: System specification.
    invariant a_le_b: a <= b;

    // Part 3: Proof script.
    control {
        induction;
        check;
        print_results;
    }
}
```

Fibonacci Induction Attempt

```
module main {

    // Part 1: System description.
    var a, b : integer;
    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    // Part 2: System specification.
    invariant a_le_b: a <= b;

    // Part 3: Proof script.
    control {
        induction;
        check;
        print_results;
    }
}
```

Fibonacci Induction Attempt

```
module main {

    // Part 1: System description.
    var a, b : integer;
    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    // Part 2: System specification.
    invariant a_le_b: a <= b;

    // Part 3: Proof script.
    control {
        induction;
        check;
        print_results;
    }
}
```

The induction algorithm checks

Fibonacci Induction Attempt

```
module main {

    // Part 1: System description.
    var a, b : integer;
    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    // Part 2: System specification.
    invariant a_le_b: a <= b;

    // Part 3: Proof script.
    control {
        induction;
        check;
        print_results;
    }
}
```

The induction algorithm checks

- P_1: a_le_b holds at init

Fibonacci Induction Attempt

```
module main {

    // Part 1: System description.
    var a, b : integer;
    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    // Part 2: System specification.
    invariant a_le_b: a <= b;

    // Part 3: Proof script.
    control {
        induction;
        check;
        print_results;
    }
}
```

The induction algorithm checks

- P_1: a_le_b holds at init
- P_2: if a_le_b holds on entry to next, then it will hold on exit

Fibonacci Induction Attempt

```
module main {

    // Part 1: System description.
    var a, b : integer;
    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    // Part 2: System specification.
    invariant a_le_b: a <= b;

    // Part 3: Proof script.
    control {
        induction;
        check;
        print_results;
    }
}
```

The induction algorithm checks

- P_1: a_le_b holds at init
- P_2: if a_le_b holds on entry to next, then it will hold on exit

a_le_b actually does hold, but it is not inductive (P_2 is not valid)

Fibonacci With Synthesis

```
module main {
    synthesis function h(x : integer, y : integer) : boolean;
    var a, b : integer;

    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    invariant a_le_b: a <= b && h(a, b);

    control {
        induction;
        check;
        print_results;
    }
}
```

Fibonacci With Synthesis

```
module main {
    synthesis function h(x : integer, y : integer) : boolean;
    var a, b : integer;

    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    invariant a_le_b: a <= b && h(a, b);

    control {
        induction;
        check;
        print_results;
    }
}
```

Fibonacci With Synthesis

```
module main {
    synthesis function h(x : integer, y : integer) : boolean;
    var a, b : integer;

    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    invariant a_le_b: a <= b && h(a, b);

    control {
        induction;
        check;
        print_results;
    }
}
```

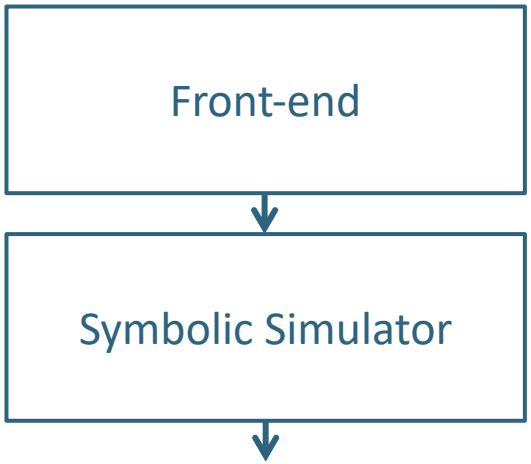
Fibonacci With Synthesis

```
module main {
    synthesis function h(x : integer, y : integer) : boolean;
    var a, b : integer;

    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    invariant a_le_b: a <= b && h(a, b); // h(a, b) := a >= 0

    control {
        induction;
        check;
        print_results;
    }
}
```

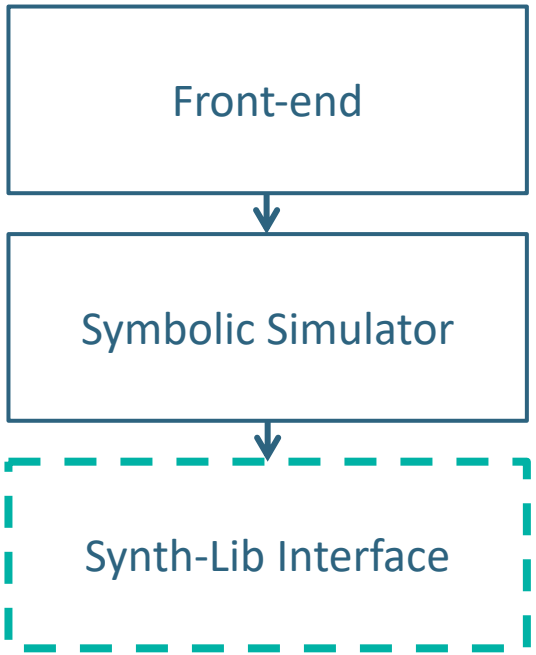
Under the Hood

Synth-Lib Intermediate Representation

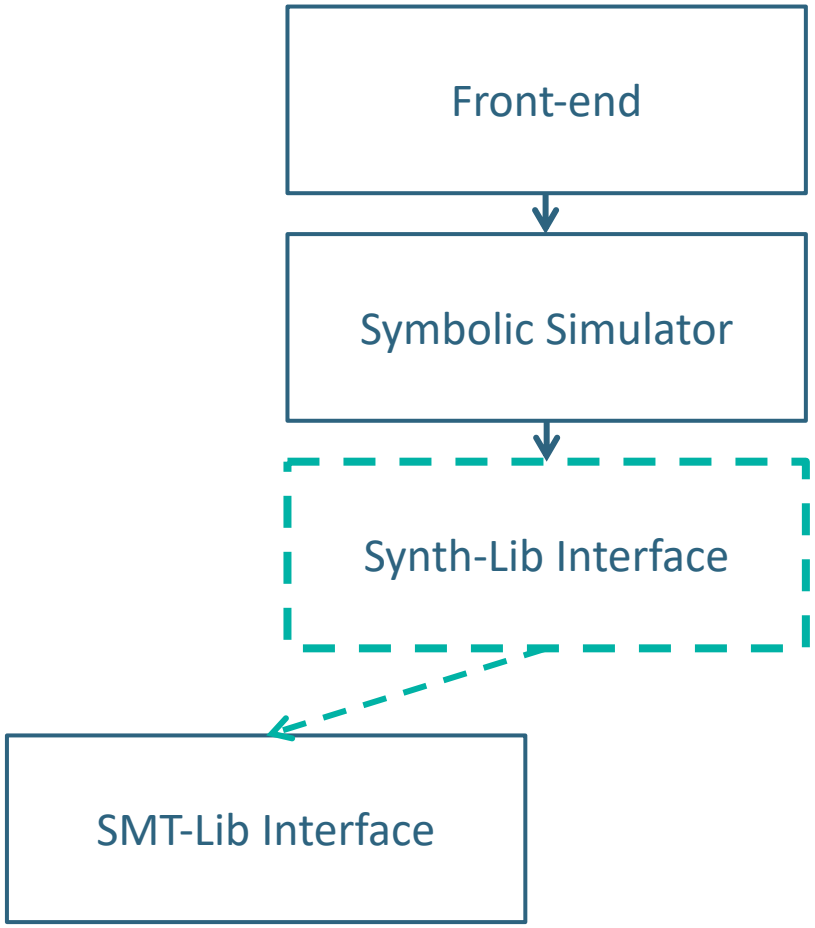
Architecture of Synthesis in Uclid5



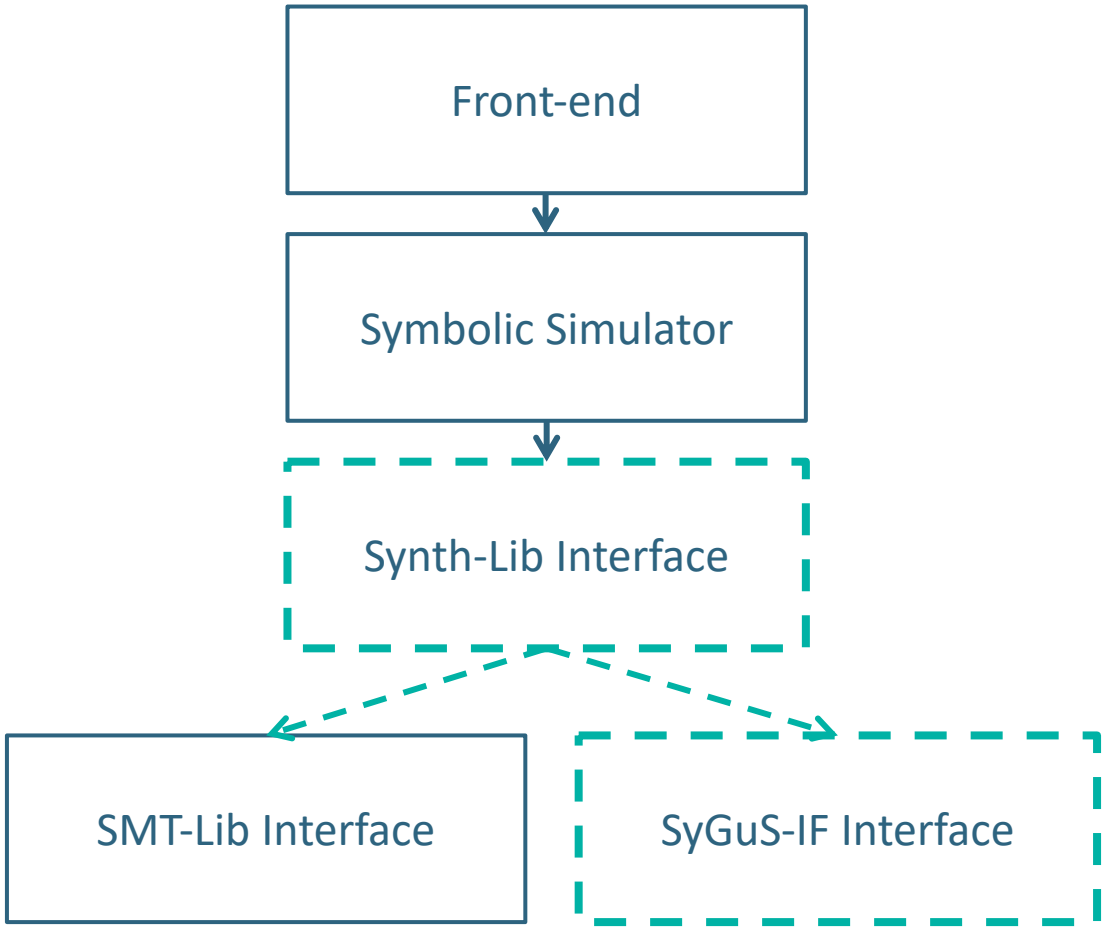
Architecture of Synthesis in Uclid5



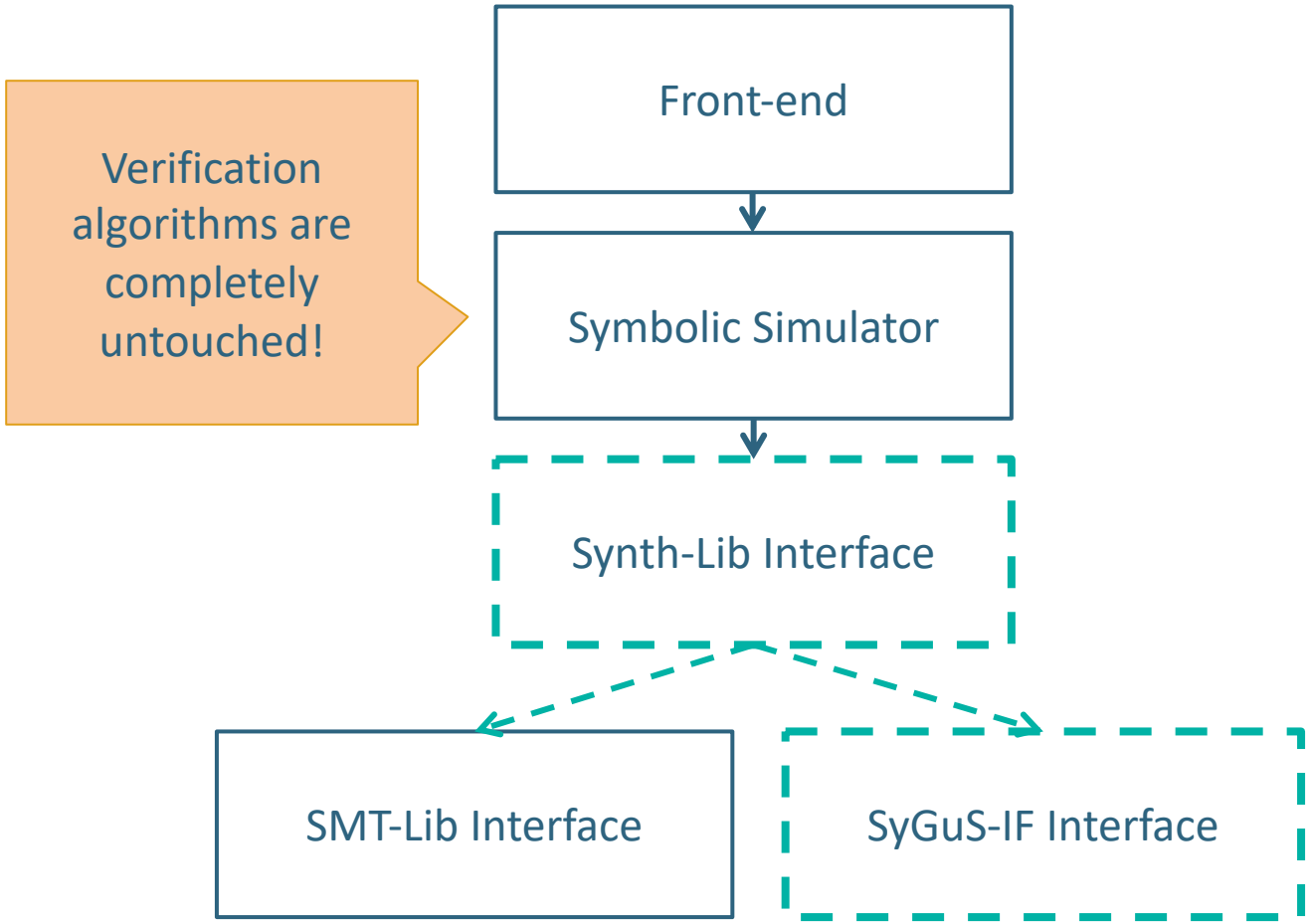
Architecture of Synthesis in Uclid5



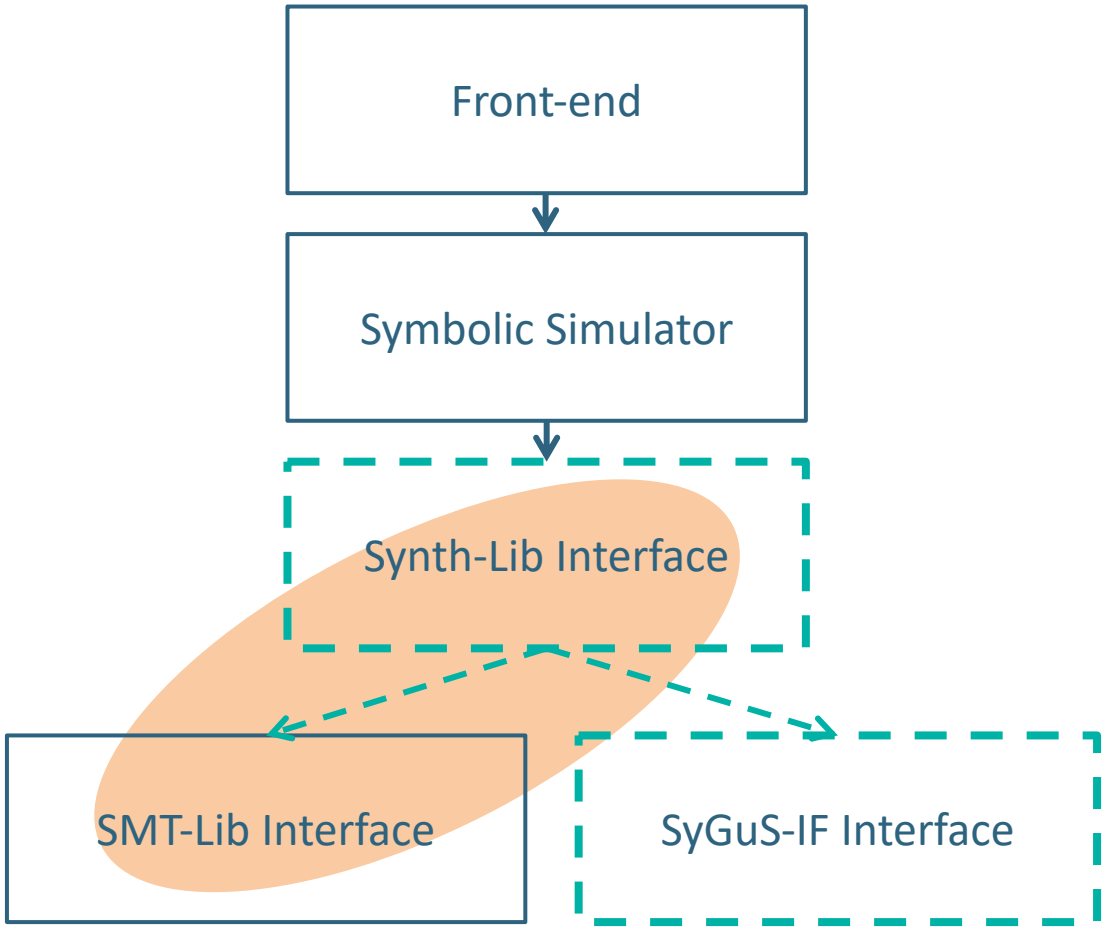
Architecture of Synthesis in Uclid5



Architecture of Synthesis in Uclid5



Architecture of Synthesis in Uclid5



Synth-Lib Encoding

```
module main {
    synthesis function h(x : integer, y : integer) : boolean;
    var a, b : integer;

    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    invariant a_le_b: a <= b && h(a, b);

    control {
        induction;
        check;
        print_results;
    }
}
```

Synth-Lib Encoding

```
module main {
    synthesis function h(x : integer, y : integer) : boolean;
    var a, b : integer;

    init {
        a = 0;
        b = 1;
    }
    next {
        a', b' = b, a + b;
    }

    invariant a_le_b: a <= b && h(a, b);

    control {
        induction;
        check;
        print_results;
    }
}
```

The induction algorithm checks

- P_1: a_le_b holds at init
- P_2: if a_le_b holds on entry to next, then it will hold on exit

Synth-Lib Encoding

```
(synth-blocking-fun h ((x Int) (y Int)) Bool)

(declare-fun initial_b () Int)
(declare-fun initial_a () Int)
(declare-fun new_a () Int)
(declare-fun new_b () Int)

(assert (or
    (not (and (<= initial_a initial_b) (h 0 1))) ;(not P1)
    (and (and (<= initial_a initial_b) (h initial_a initial_b))
        (= new_a initial_b)
        (= new_b (+ initial_a initial_b ))
        (not (and (<= new_a new_b) (h new_a new_b)))))) ;(not P2)

(check-sat)
```

The induction algorithm checks

- P_1: a_le_b holds at init
- P_2: if a_le_b holds on entry to next, then it will hold on exit

Synth-Lib Encoding

```
(synth-blocking-fun h ((x Int) (y Int)) Bool)

(declare-fun initial_b () Int)
(declare-fun initial_a () Int)
(declare-fun new_a () Int)
(declare-fun new_b () Int)

(assert (or
    (not (and (<= initial_a initial_b) (h 0 1))) ;(not P1)
    (and (and (<= initial_a initial_b) (h initial_a initial_b))
        (= new_a initial_b)
        (= new_b (+ initial_a initial_b ))
        (not (and (<= new_a new_b) (h new_a new_b)))))) ;(not P2)

(check-sat)
```

$\exists h \neg \exists a, b \neg P_1(h, a, b) \vee \neg P_2(h, a, b)$

The induction algorithm checks

- P_1: a_le_b holds at init
- P_2: if a_le_b holds on entry to next, then it will hold on exit

SMT-Lib Encoding

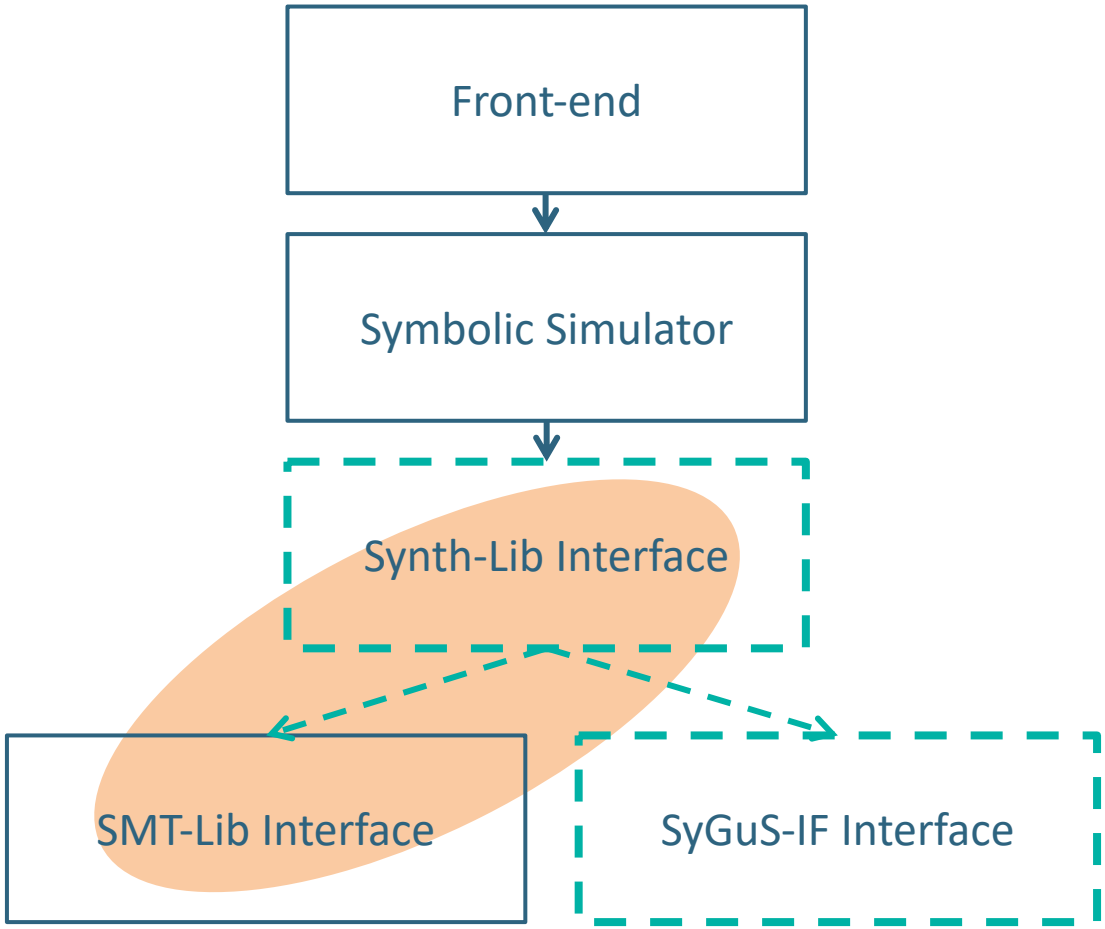
```
(define-fun h ((x Int) (y Int)) Bool (>= x 0))

(declare-fun initial_b () Int)
(declare-fun initial_a () Int)
(declare-fun new_a () Int)
(declare-fun new_b () Int)

(assert (or
    (not (and (<= initial_a initial_b) (h 0 1))) ;(not P1)
    (and (and (<= initial_a initial_b) (h initial_a initial_b))
        (= new_a initial_b)
        (= new_b (+ initial_a initial_b ))
        (not (and (<= new_a new_b) (h new_a new_b))))))) ;(not P2)

(check-sat)
```

UNSAT iff $\neg\exists a, b\ \neg P_1(h, a, b) \vee \neg P_2(h, a, b)$

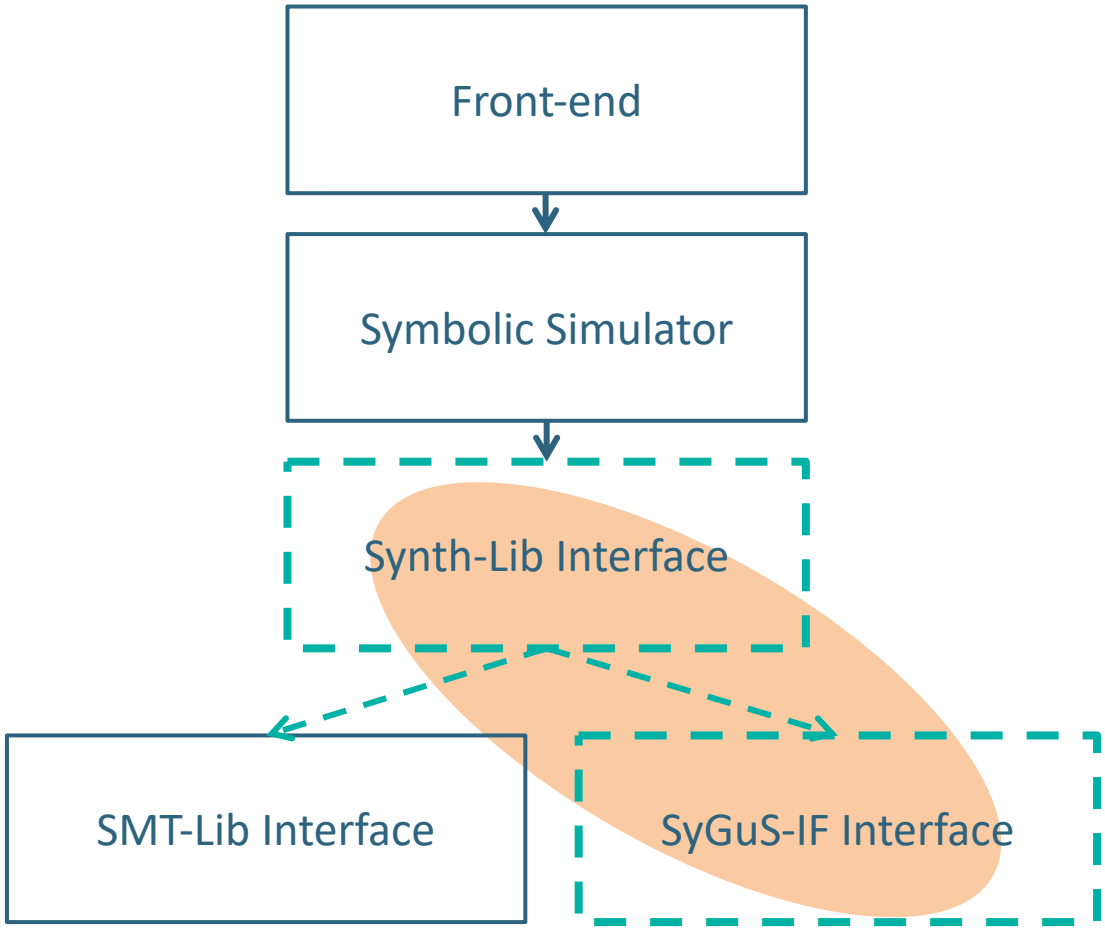
The induction algorithm checks

- P_1: a_le_b holds at init
- P_2: if a_le_b holds on entry to next, then it will hold on exit

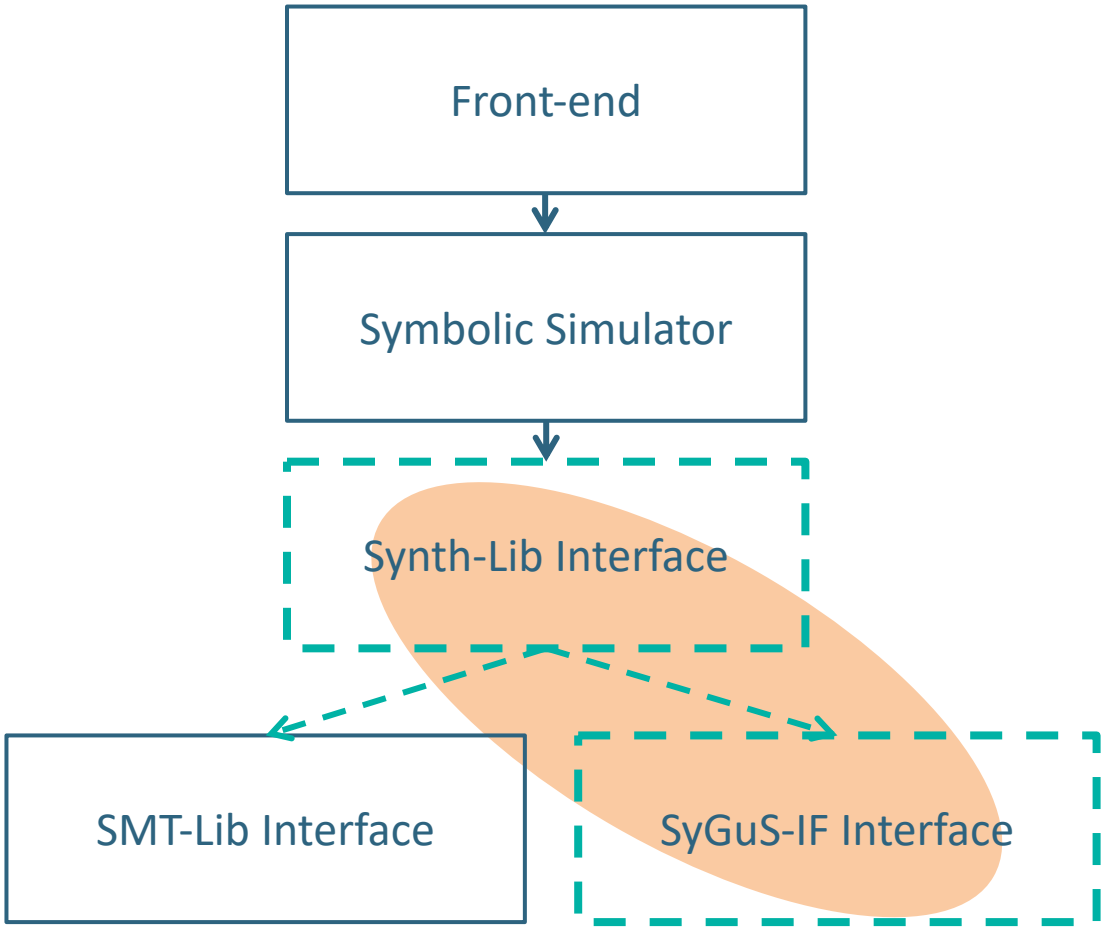
Architecture of Synthesis in Uclid5



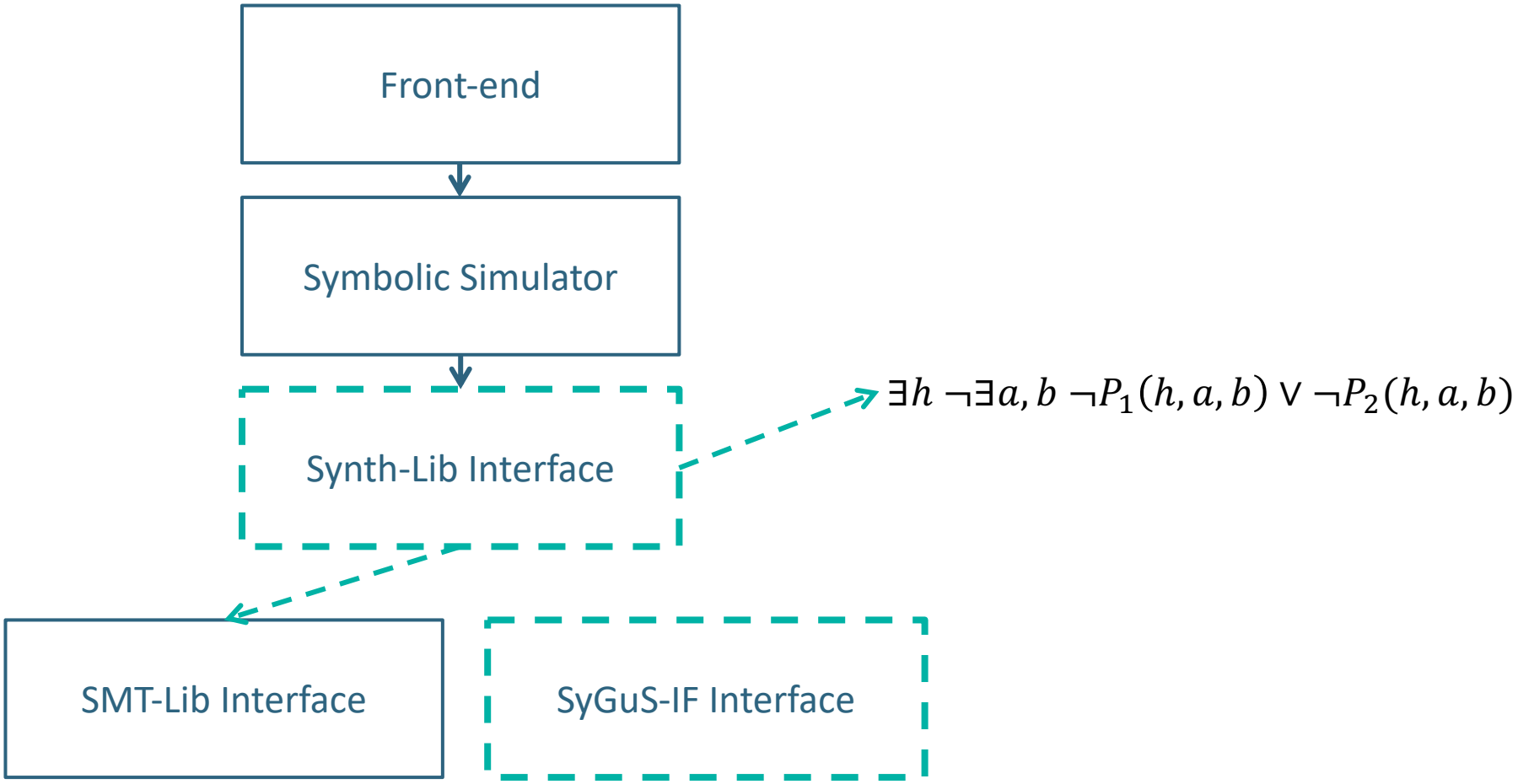
Architecture of Synthesis in Uclid5



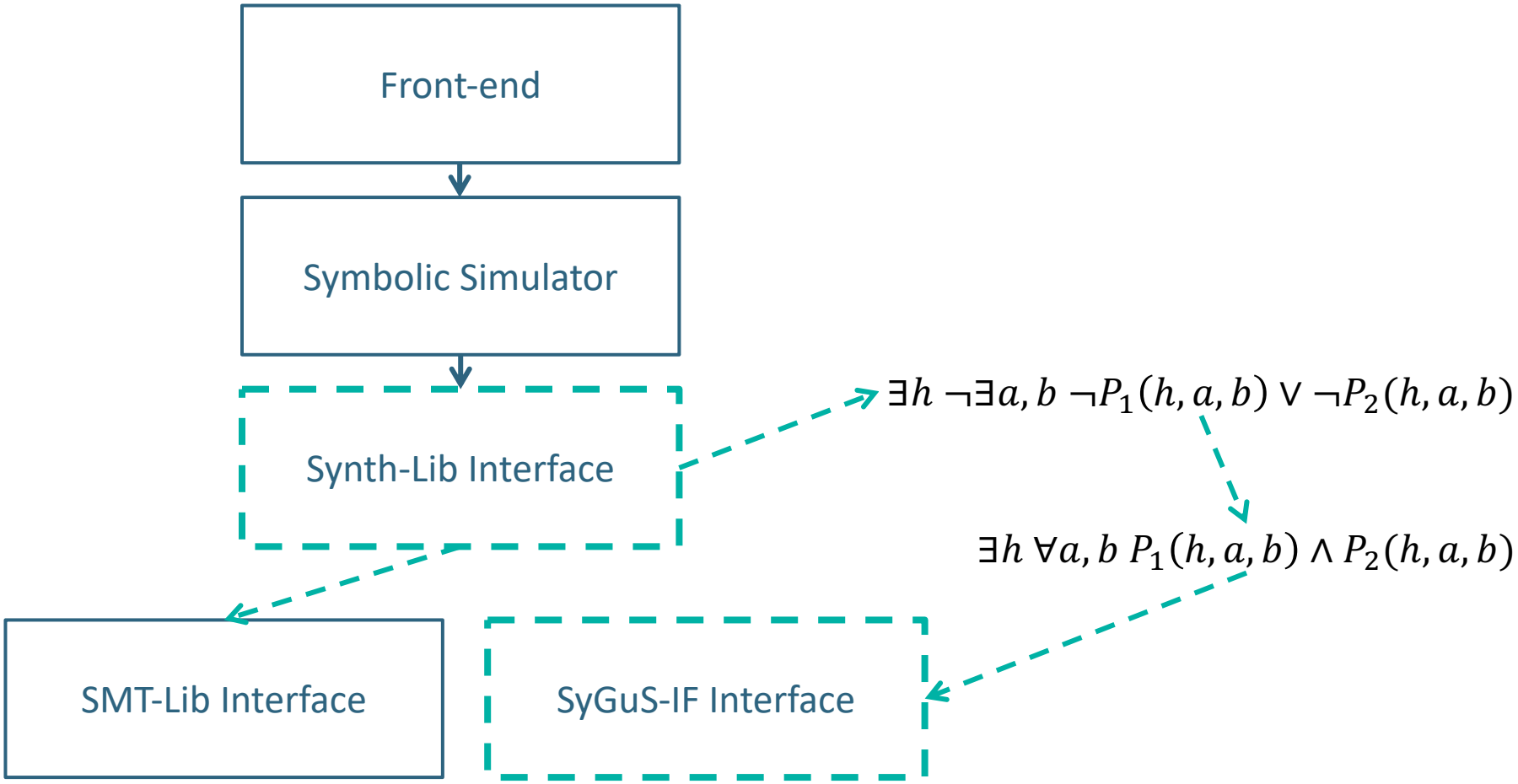
Architecture of Synthesis in Uclid5



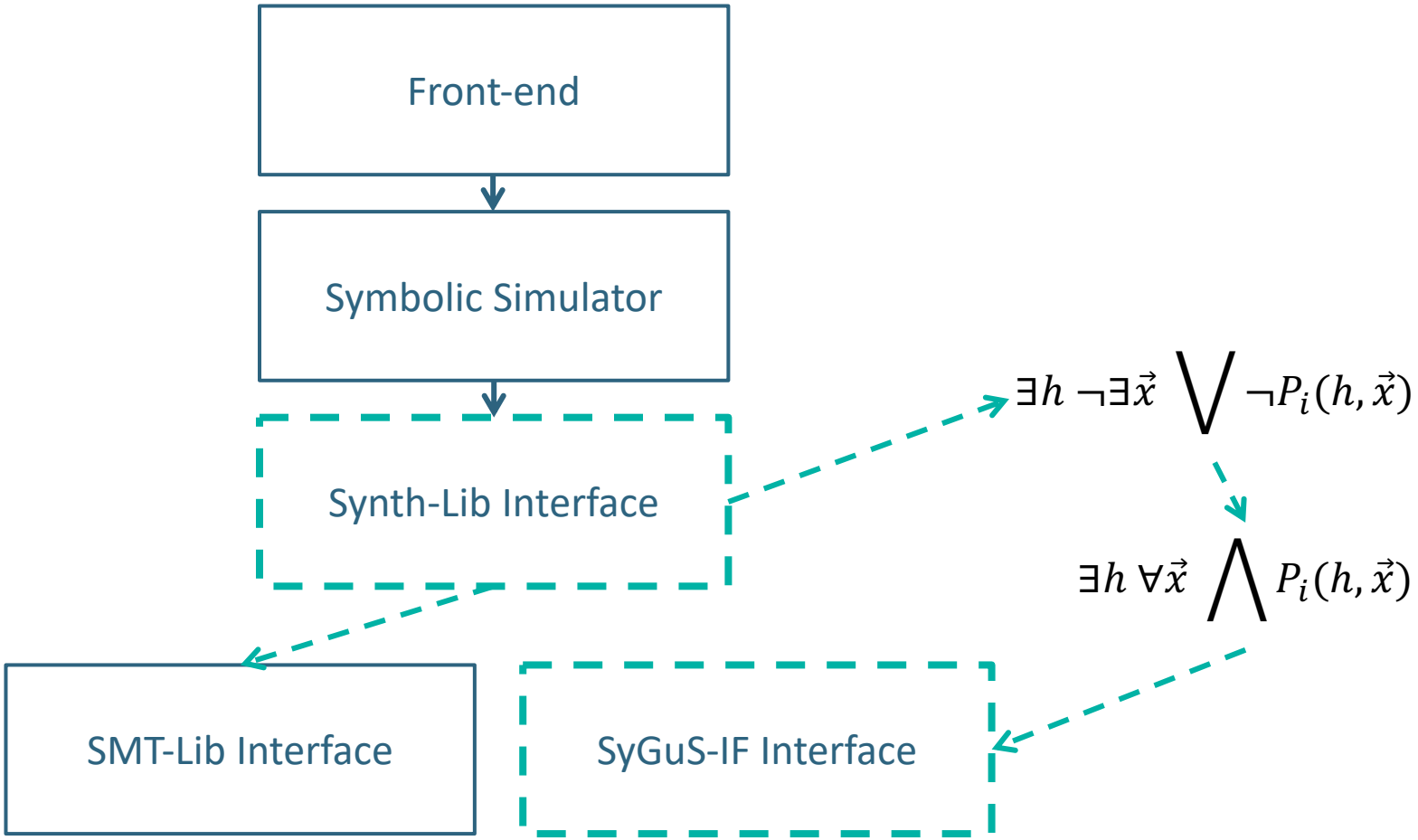
Architecture of Synthesis in Uclid5



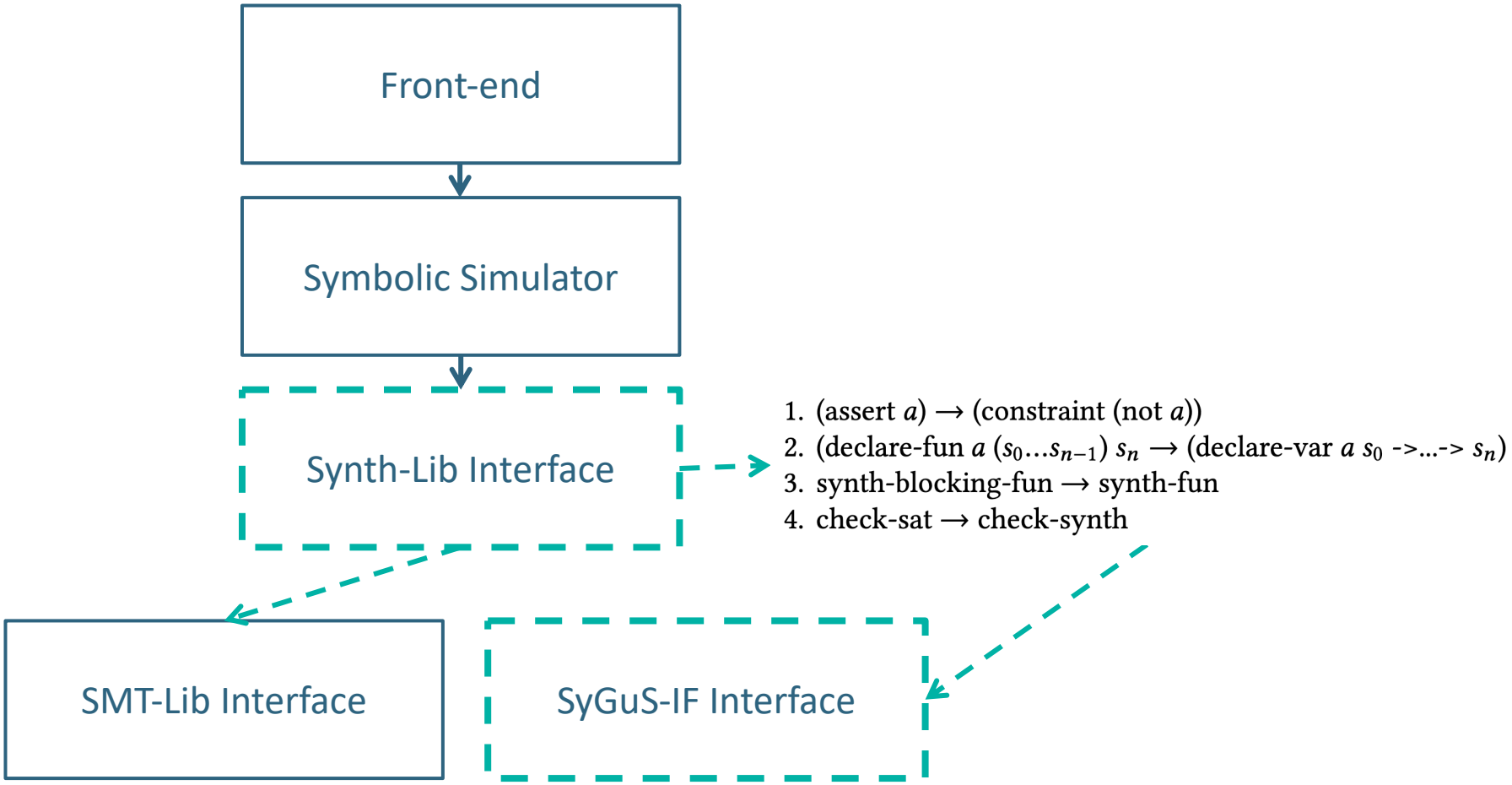
Architecture of Synthesis in Uclid5



Architecture of Synthesis in Uclid5



Architecture of Synthesis in Uclid5



The Benchmarks

One example out of 25

Benchmark Example [5]

```
…
define pi_balance(p : bv8, b : bv16) : bv16 =
    b[15:8] ++ (b[7:0] + p);
…
init {
    /*-------------------------------------------------------------+
    | Injective Trace enumeration Witness (I): Property 12 |
    | ==================================== |
    | Makes use of enumeration predicate (pi_balance) defined above.
    |
    | - Mapping initial state. |
    +-------------------------------------------------------------*/
    assume (acct1.balance == pi_balance(p1, acct0.balance));
    assume (acct2.balance == pi_balance(p2, acct0.balance));
}
…
```

[5] Sahai, Shubham, et al. "Verification of Quantitative Hyperproperties Using Trace Enumeration Relations." CAV '20

Benchmark Example [5]

```
…
define pi_balance(p : bv8, b : bv16) : bv16 =
    b[15:8] ++ (b[7:0] + p);
…
init {
    /*-----------------------------------------------------------+
    | Injective Trace enumeration Witness (I): Property 12 |
    | ==================================== |
    | Makes use of enumeration predicate (pi_balance) defined above.
    |
    | - Mapping initial state. |
    +-----------------------------------------------------------*/
    assume (acct1.balance == pi_balance(p1, acct0.balance));
    assume (acct2.balance == pi_balance(p2, acct0.balance));
}
…
```

[5] Sahai, Shubham, et al. "Verification of Quantitative Hyperproperties Using Trace Enumeration Relations." CAV '20

Benchmark Example [5]

```
…
synthesis function pi_balance(p : bv8, b : bv16) : bv16;

…
init {
    /*-------------------------------------------------------------+
    | Injective Trace enumeration Witness (I): Property 12 |
    | ==================================== |
    | Makes use of enumeration predicate (pi_balance) defined above.
    |
    | - Mapping initial state. |
    +-------------------------------------------------------------*/
    assume (acct1.balance == pi_balance(p1, acct0.balance));
    assume (acct2.balance == pi_balance(p2, acct0.balance));
}
…
```

[5] Sahai, Shubham, et al. "Verification of Quantitative Hyperproperties Using Trace Enumeration Relations." CAV '20

Synthesis in Uclid5: Takeaways

Synthesis in Uclid5: Takeaways

- Verification takes time & effort.

Synthesis in Uclid5: Takeaways

- Verification takes time & effort.
- Techniques exist to automate parts of the job, but until now, not in one framework!

Synthesis in Uclid5: Takeaways

- Verification takes time & effort.
- Techniques exist to automate parts of the job, but until now, not in one framework!
- Uclid5 lets users define functions to synthesize and use them
  - anywhere in their code,
  - for any verification technique
    - (k-induction, bounded model checking, …), and
  - for any kind of specification
    - (linear temporal logic, invariants, sequential assertions, …).

Synthesis in Uclid5: Takeaways

- Verification takes time & effort.
- Techniques exist to automate parts of the job, but until now, not in one framework!
- Uclid5 lets users define functions to synthesize and use them
  - anywhere in their code,
  - for any verification technique
    - (k-induction, bounded model checking, …), and
  - for any kind of specification
    - (linear temporal logic, invariants, sequential assertions, …).
- Unfortunately, we are pushing the limits of state-of-the-art synthesis engines.

Synthesis in Uclid5: Takeaways

- Verification takes time & effort.
- Techniques exist to automate parts of the job, but until now, not in one framework!
- Uclid5 lets users define functions to synthesize and use them
  - anywhere in their code,
  - for any verification technique
    - (k-induction, bounded model checking, …), and
  - for any kind of specification
    - (linear temporal logic, invariants, sequential assertions, …).
- Unfortunately, we are pushing the limits of state-of-the-art synthesis engines.
- Fortunately, that means there's a lot of work left to be done!
  - For example, optimizing synthesis-for-verification encodings,
  - improving solvers, and so on…

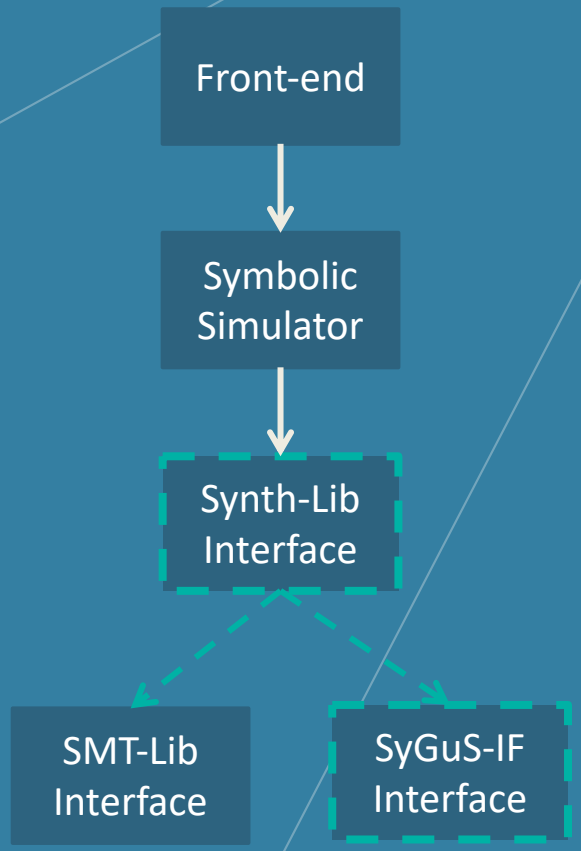
Thank you!

Paper

Code

Benchmarks

```
module main {
  synthesis function h(x : integer, y : integer) : boolean;
  var a, b : integer;

  init {
    a = 0;
    b = 1;
  }

  next {
    a', b' = b, a + b;
  }

  invariant a_le_b: a <= b && h(a, b);

  control {
    induction;
    check;
    print_results;
  }
}
```

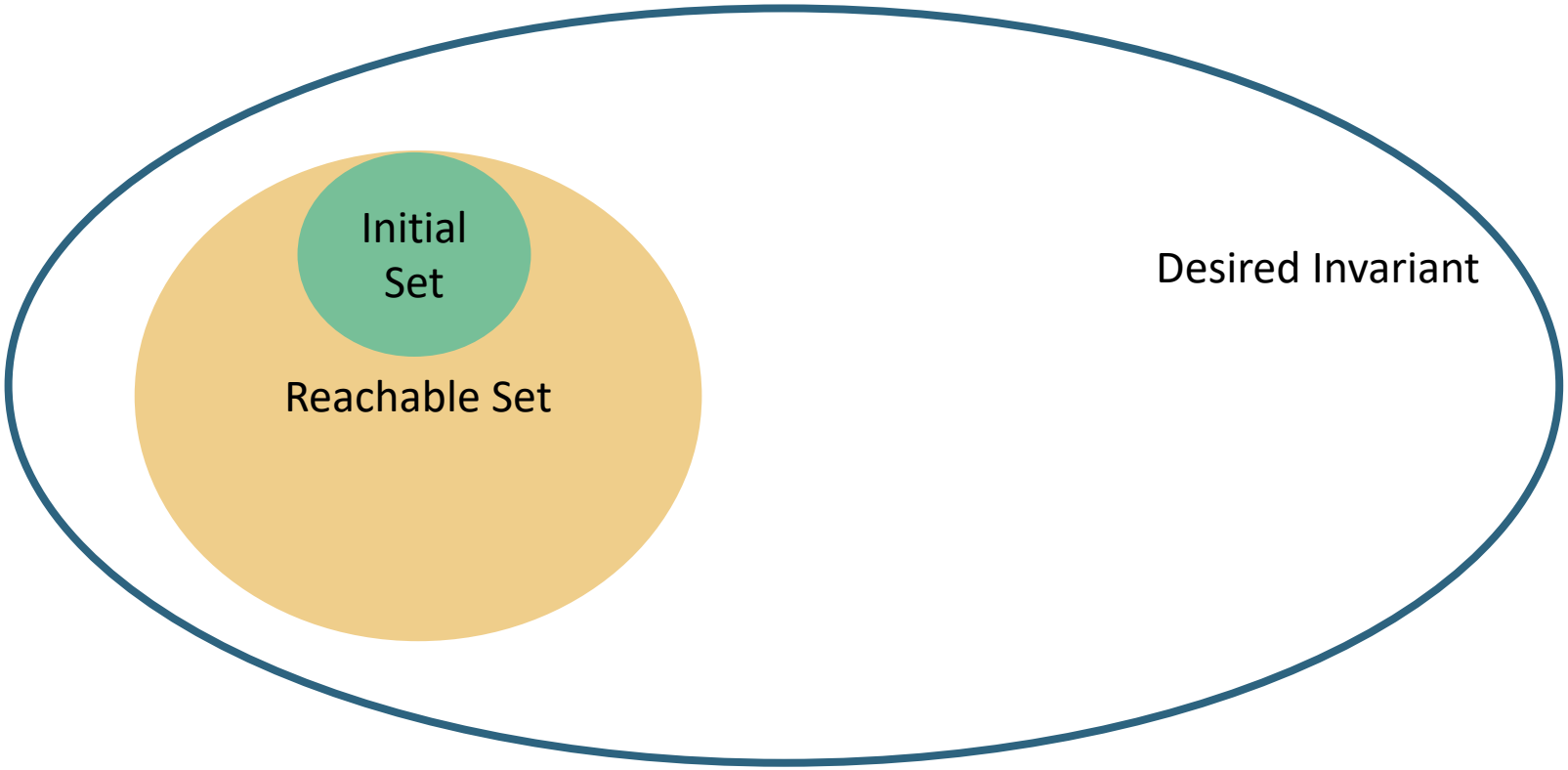


Extra Stuff

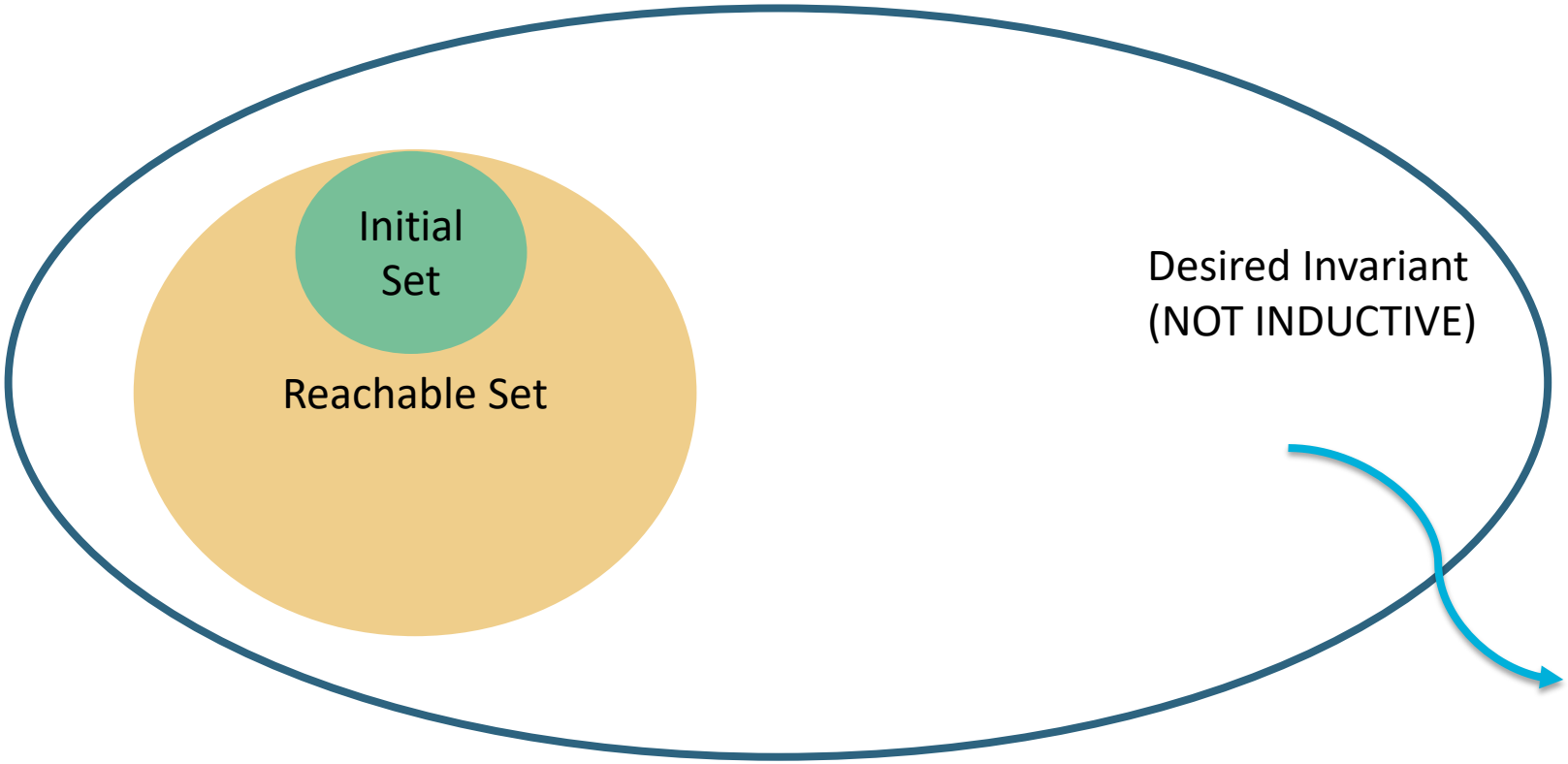
Manual Work #1 of 3

Strengthening Invariants

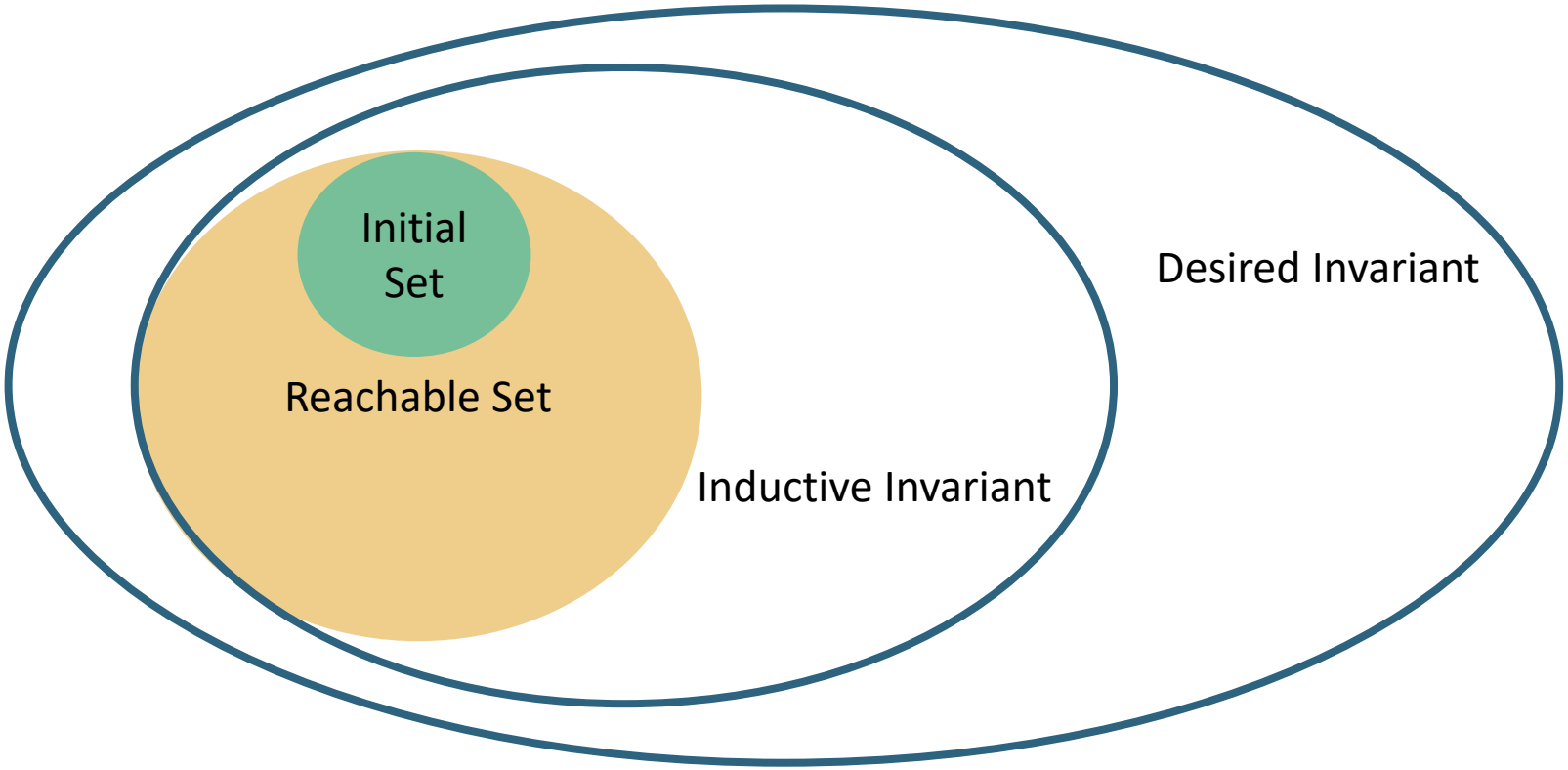
Strengthening Invariants



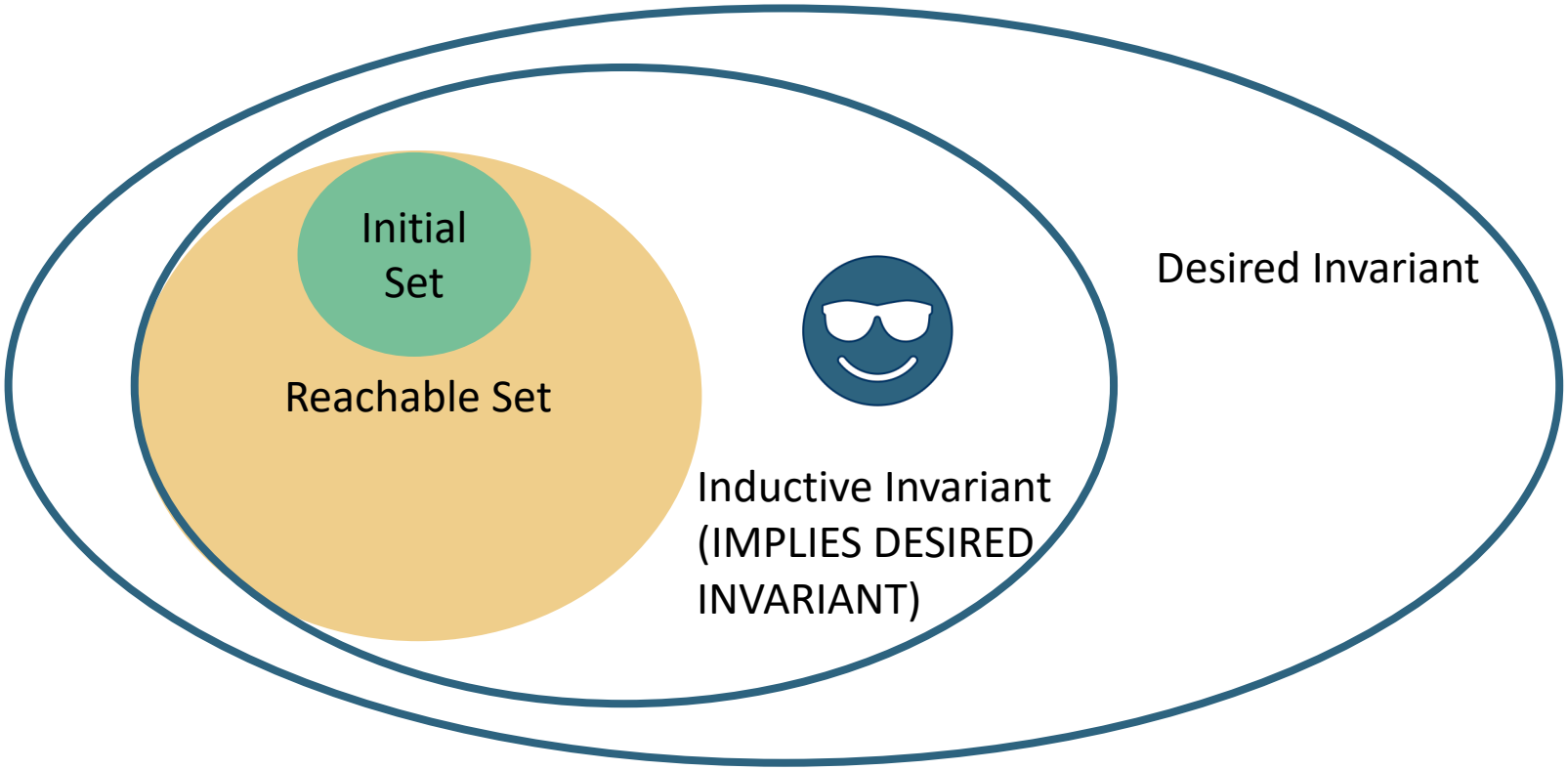
Strengthening Invariants



Strengthening Invariants



Strengthening Invariants



Manual Work #2 of 3

Annotating functions with pre- and post-conditions

Annotating functions with pre- and post-conditions

```
procedure searchQ() returns (found : boolean)
    requires (count >= 0 && count <= SIZE);
    ensures (in_queue(data) <==> found);
{
    var i : integer;
    i = 0;
    found = false;
    while (i < count)
        invariant (i >= 0 && i <= count);
    {
        if (contents[itemIndex(i)] == data) {
            found = true;
        }
        i = i + 1;
    }
}
```

Annotating functions with pre- and post-conditions

```
procedure searchQ() returns (found : boolean)
    requires (count >= 0 && count <= SIZE);
    ensures (in_queue(data) <==> found);
{
    var i : integer;
    i = 0;
    found = false;
    while (i < count)
        invariant (i >= 0 && i <= count);
    {
        if (contents[itemIndex(i)] == data) {
            found = true;
        }
        i = i + 1;
    }
}
```

Manual Work #3 of 3

Modelling the Environment and System Calls

Modelling the Environment and System Calls

```
var a, b : integer;

init {
    a = time();
    b = time();
}

property bigger_than_a : (b >= a);
```

Modelling the Environment and System Calls

```
var a, b : integer;

init {
   a = time();
   b = time();
}

property bigger_than_a : (b >= a);
```