# Using SMT solvers for program analysis

Shaz Qadeer
Research in Software Engineering
Microsoft Research

# Satisfiability modulo theories

$(a \vee \neg c)$
$(b \vee \neg c)$
$(a \vee b \vee c)$

$c$ = true
$b$ = true
$a$ = true

$(a \vee \neg c)$
$(b \vee \neg c)$
$(a \vee b \vee c)$
$a \cong f(x-y) = 1$
$b \cong f(y-x) = 2$
$c \cong x = y$

$c$ = false,
$b$ = true,
$a$ = true,
$x = 0$,
$y = 1$,
$f = [-1 \rightarrow 1, 1 \rightarrow 2, \text{else} \rightarrow 0]$

# Communicating theories

$f(x - y) = 1, f(y-x) = 2, x = y$

$x = y$ $q = s$

$f(p) = q, f(r) = s, x = y$

$p = x - y, q = 1, r = y - x, s = 2$

UNSAT

$p = r$

# Applications

- Symbolic execution
  - SAGE
  - PEX
- Static checking of code contracts
  - Spec#
  - Dafny
  - VCC
- Security analysis
  - HAVOC
- Searching program behaviors
  - Poirot

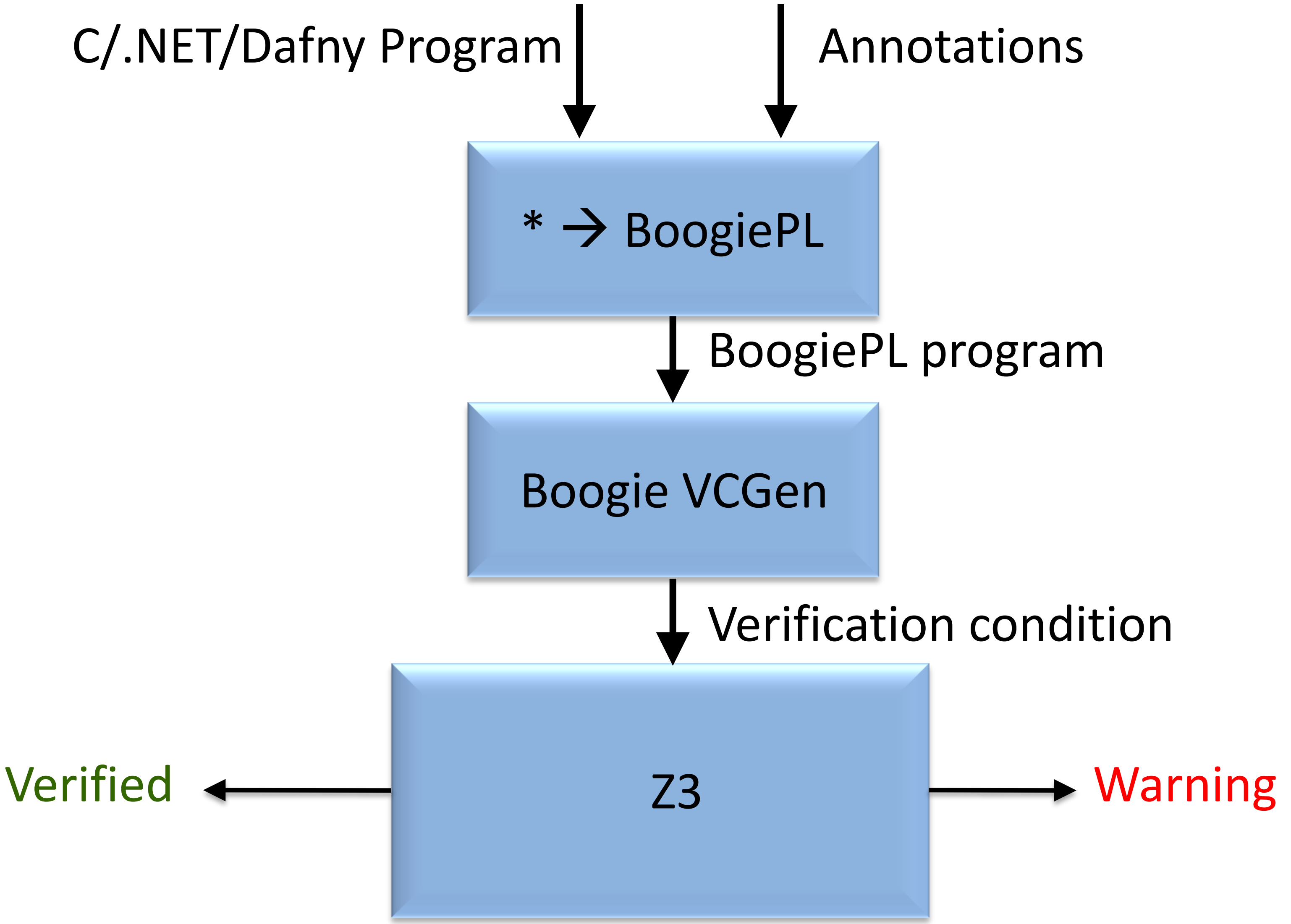
# Anatomy of an application

- The profile of each application determined by
  - Boolean structure
  - theories used
  - theory vs. propositional
  - deep vs. shallow
  - presence/absence of quantifiers
  - …

# Applications

- Symbolic execution
  - SAGE
  - PEX
- Static checking of code contracts
  - Spec#
  - Dafny
  - VCC
- Security analysis
  - HAVOC
- Searching program behaviors
  - Poirot

# SMT in program analysis



```
class C {
  int size;
  int[] data;

  void write(int i, int v) {
    if (i >= data.Length) {
      var t = new int[2*i];
      copy(data, t);
      data = t;
    }
    data[i] = v;
  }

  static copy(int[] from, int[] to) {
    for (int i = 0; i < from.Length; i++) {
      to[i] = from[i];
    }
  }
}
```

```
var size: [Ref]int;
var data: [Ref]Ref;
var Contents: [Ref][int]int
function Length(Ref): int;

proc write(this: Ref, i: int, v: int) {
  var t: Ref;
  if (i >= Length(data)) {
    call t := alloc();
    assume Length(t) == 2*i;
    call copy(data[this], t);
    data[this] := t;
  }
  assert 0 <= i && i < Length(data[this]);
  Contents[data[this]][i] := v;
}

proc copy(from: Ref, to: Ref) {
  var i: int;
  i := 0;
  while (i < Length(from)) {
    assert 0 <= i && i < Length(from);
    assert 0 <= i && i < Length(to);
    Contents[to][i] := Contents[from][i];
    i := i + 1;
  }
}
```

# Modeling the heap

```
var Alloc: [Ref]bool;
proc alloc() returns (x: int) {
   assume !Alloc[x];
   Alloc[x] := true;
}
```

Theory of arrays: Select, Store

for all f, i, v :: Select(Update(f, i, v), i) = v

for all f, i, v, j :: i = j $\vee$ Select(Update(f, i, v), j) = Select(f, j)

for all f, g :: f = g $\vee$ (exists i :: Select(f, i) $\neq$ Select(g, i))

Contents[data[this]][i] := v

Contents[Select(data, this)][i] := v

Contents[Select(data, this)] := Update(Contents[Select(data, this)], i, v)

Contents := Update(Contents, Select(data, this), Update(Contents[Select(data, this)], i, v))

# Program correctness

- Floyd-Hoare triple

  {P} S {Q}

  P, Q : predicates/property

  S : a program

- From a state satisfying P, if S executes,
  - No assertion in S fails, and
  - Terminating executions end up in a state satisfying Q

# Annotations

- Assertions over program state
- Can appear in
  - Assert
  - Assume
  - Requires
  - Ensures
  - Loop invariants
- Program state can be extended with ghost variables
  - State of a lock
  - Size of C buffers

# Weakest liberal precondition

| | | |
|---|---|---|
| wlp( assert E, Q ) | = | $E \wedge Q$ |
| wlp( assume E, Q ) | = | $E \Rightarrow Q$ |
| wlp( S;T, Q ) | = | wlp(S, wlp(T, Q)) |
| wlp( if E then S else T, Q ) | = | if E then wlp(S, Q) else wlp(T, Q) |
| wlp( x := E, Q ) | = | Q[E/x] |
| wlp( havoc x, Q ) | = | $\forall x.\ Q$ |

# Desugaring loops

- **inv J** while B do S end

- Replace loop with loop-free code:

```
assert J;
havoc modified(S);
assume J;
```

Check J at entry

```
if (B) {
  S;
  assert J;
  assume false;
}
```

Check J is inductive

# Desugaring procedure calls

- Each procedure verified separately
- Procedure calls replaced with their specifications

```
procedure Foo();
requires  pre;
ensures  post;
modifies V;
```

- requires pre → precondition
- ensures post → postcondition
- modifies V → set of variables possibly modified in Foo

```
call Foo();
```

→

```
assert pre;
havoc V;
assume post;
```

# Inferring annotations

- Problem statement
  - Given a set of procedures P1, …, Pn
  - A set of C of candidate annotations for each procedure
  - Returns a subset of the candidate annotations such that each procedure satisfies its annotations

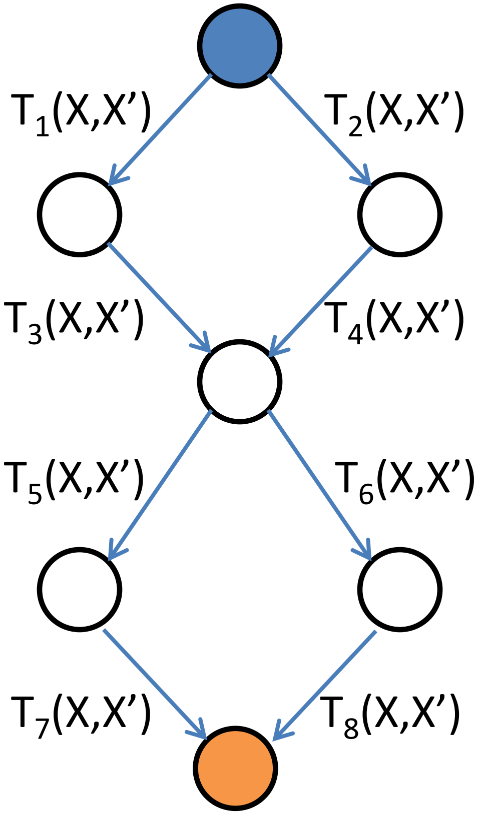
- Houdini algorithm
  - Performs a greatest-fixed point starting from all annotations
    - Remove annotations that are violated
  - Requires a quadratic $(n * |C|)$ number of queries to a modular verifier

# Limits of modular analysis

- Supplying invariants and contracts may be difficult for developers
- Other applications may be enabled by whole program analysis
  - Answering developer questions: how did my program get to this line of code?
  - Crash-dump analysis: reconstruct executions that lead to a particular failure

# Reachability modulo theories

Variables: X




$T_i(X, X')$ are transition predicates for transforming input state X to output state X'

- assume satisfiability for $T_i(X, X')$ is "efficiently" decidable

Is there a feasible path from blue to orange node?

Parameterized in two dimensions

- theories: Boolean, arithmetic, arrays, …
- control flow: loops, procedure calls, threads, …

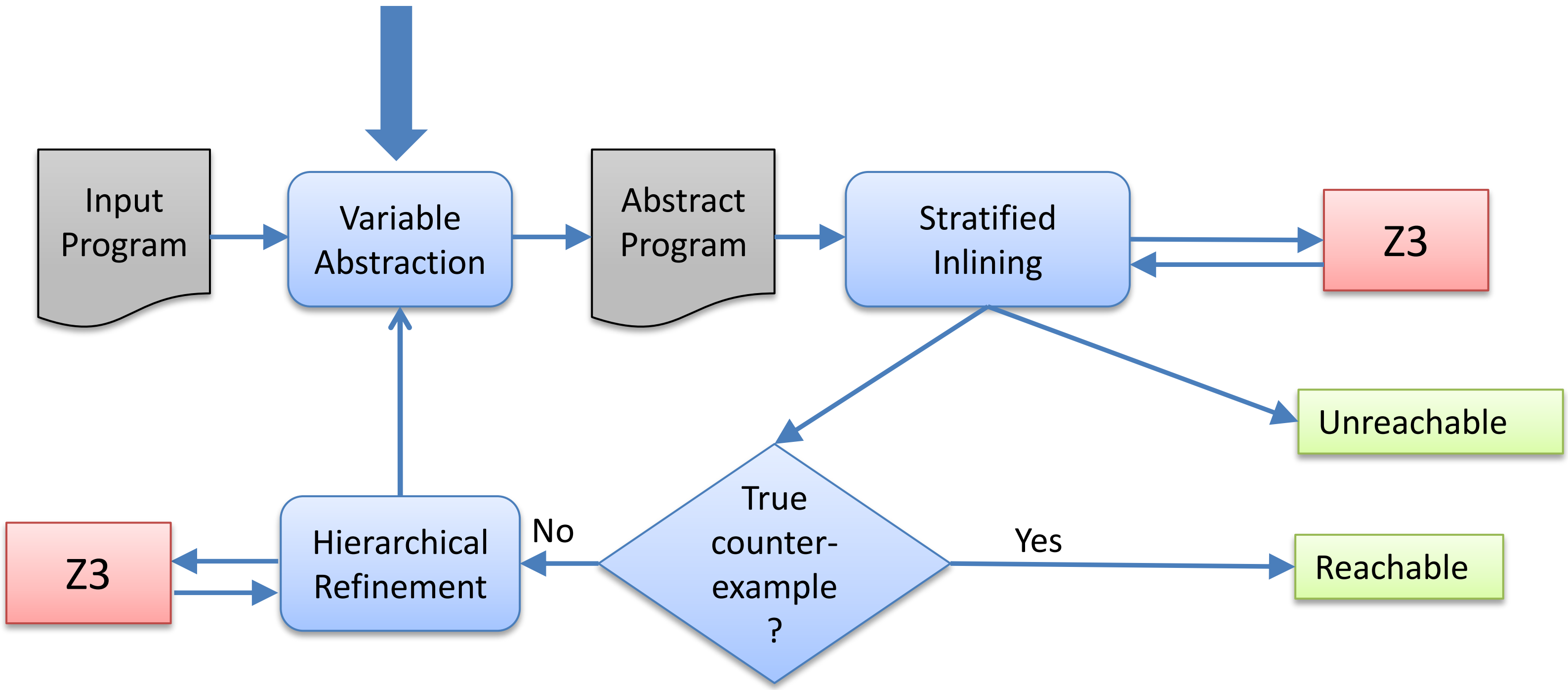
# Complexity of (sequential) reachability-modulo-theories

- Undecidable in general
  - as soon as unbounded executions are possible
- Decidable for hierarchical programs
  - PSPACE-hard (with only Boolean variables)
  - NEXPTIME-hard (with uninterpreted functions)
  - in NEXPTIME (if satisfiability-modulo-theories in NP)

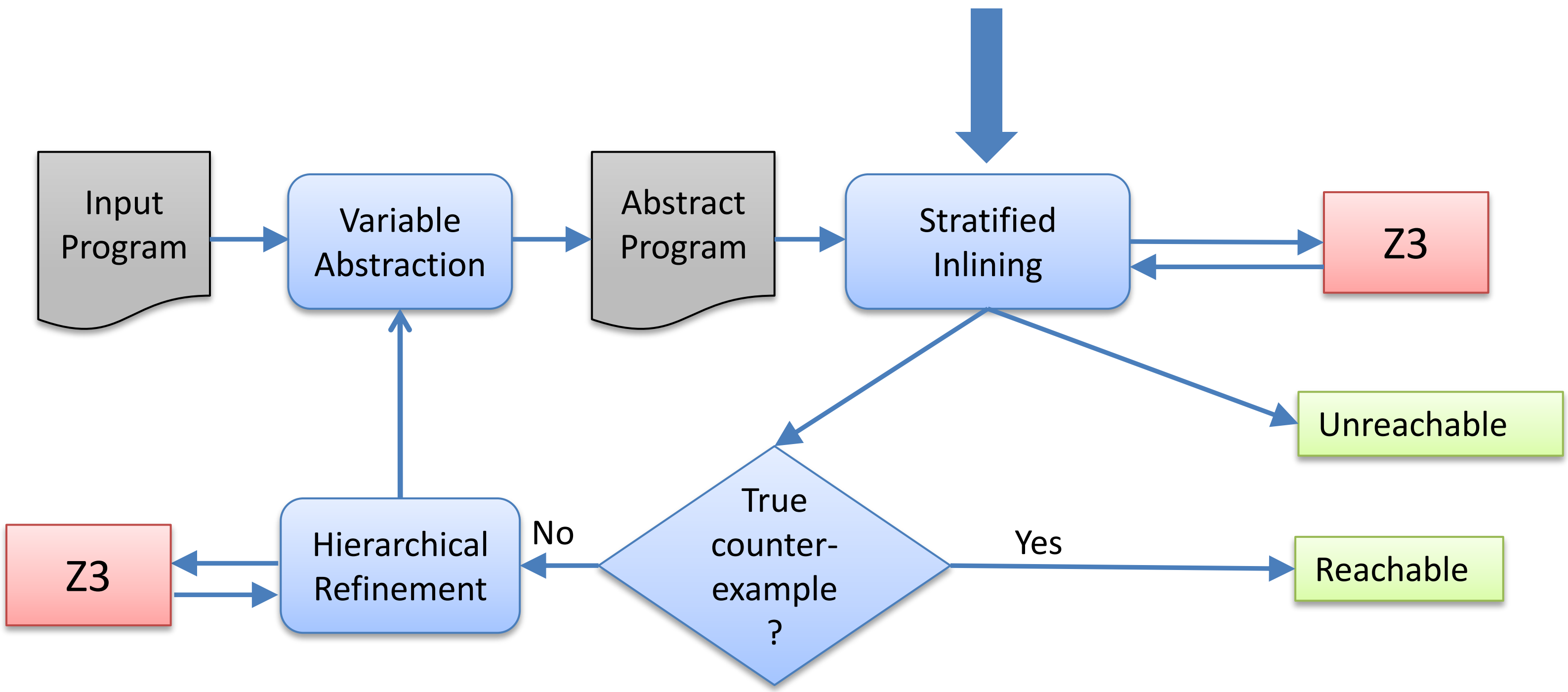
# Corral: A solver for reachability-modulo-theories

- Solves queries up to a finite recursion depth
  – reduces to hierarchical programs
- Builds on top of Z3 solver for satisfiability-modulo-theories
- Design goals
  – exploit efficient goal-directed search in Z3
  – use abstractions to speed-up search
  – avoid the exponential cost of static inlining

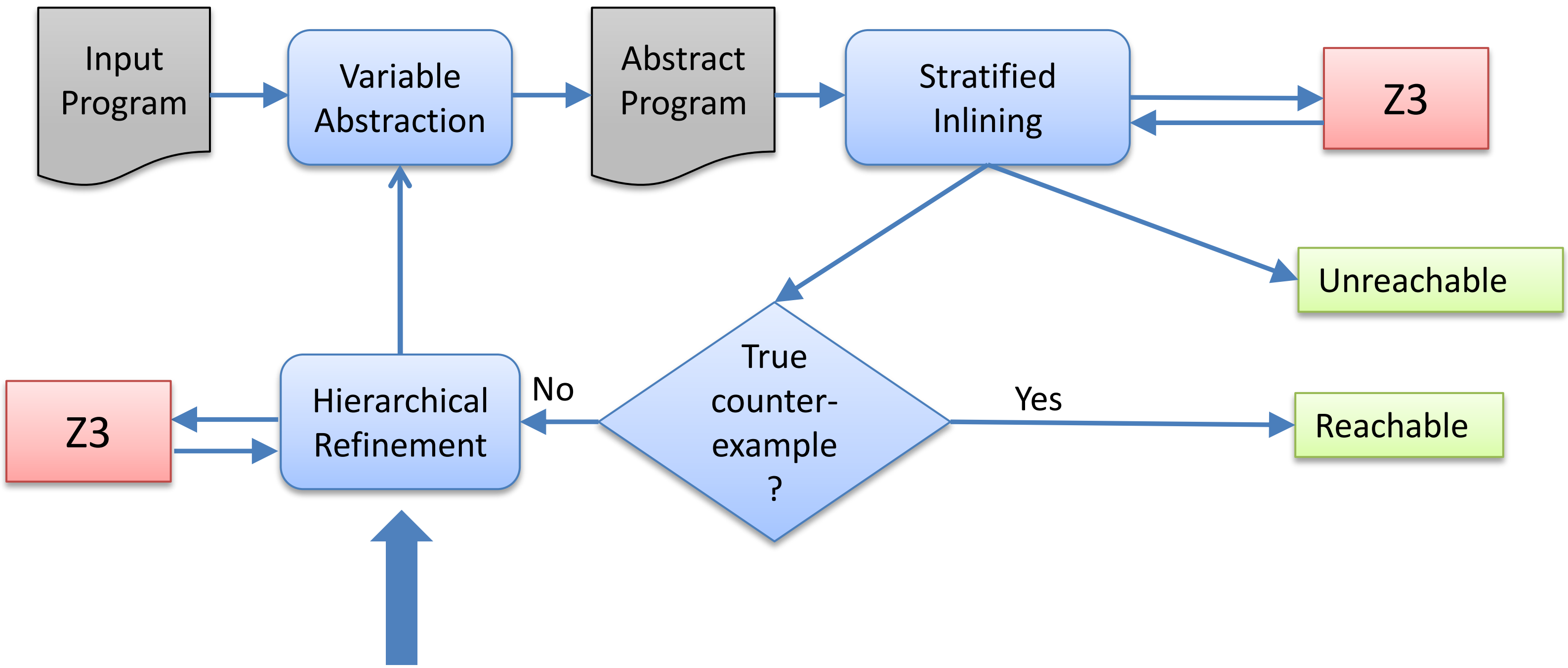
# Corral architecture for sequential programs



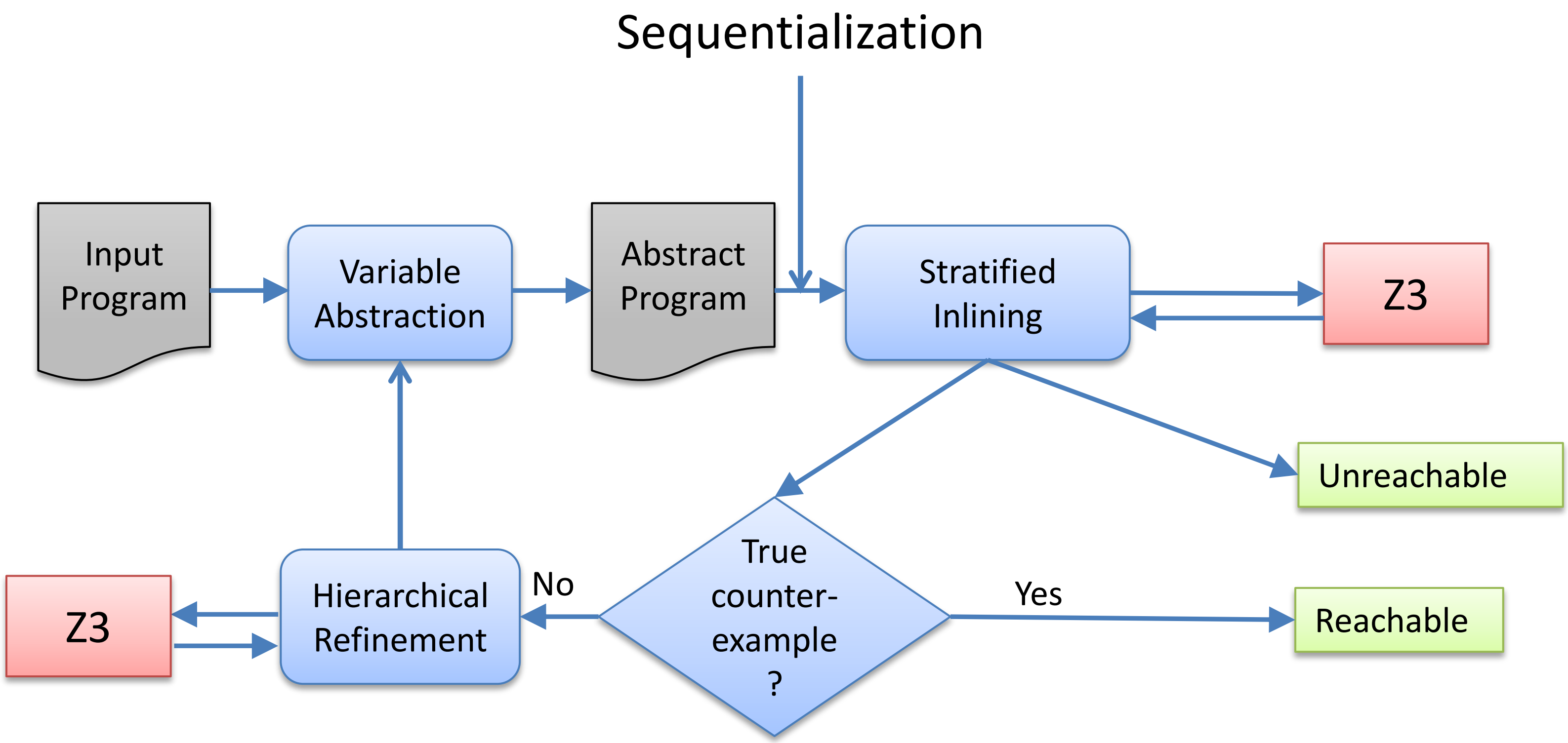
# Corral architecture for sequential programs



# Corral architecture for sequential programs



# Handling concurrency



# What is sequentialization?

- Given a concurrent program P, construct a sequential program Q such that Q $\subseteq$ P
- Drop each occurrence of async-call
- Convert each occurrence of async-call to call
- Make Q as large as possible

# Parameterized sequentialization

- Given a concurrent program P, construct a family of programs $Q_i$ such that
  - $Q_0 \subseteq Q_1 \subseteq Q_2 \subseteq \ldots \subseteq P$
  - $\cup_i Q_i = P$
- Even better if interesting behaviors of P manifest in $Q_i$ for low values of i

# Context-bounding

- Captures a notion of interesting executions in concurrent programs

- Under-approximation parameterized by $K \geq 0$
  - executions in which each thread gets at most K contexts to execute
  - as $K \to \infty$, we get all behaviors

# Context-bounding is sequentializable

- For any concurrent program P and $K \geq 0$, there is a sequential program $Q_K$ that captures all executions of P up to context bound K
- Simple source-to-source transformation
  - linear in |P| and K
  - each global variable is copied K times

# Challenges

# Programming SMT solvers

- Little support for decomposition
  - Floyd-Hoare is the only decomposition rule
- Little support for abstraction
  - SMT solvers are a black box
  - difficult to influence search
- How do we calculate program abstractions using an SMT solver?

# Mutable dynamically-allocated memory

- Select-Update theory is expensive
- Select-Update theory is not expressive enough
  - to represent heap shapes
  - to encode frame conditions

# Quantifiers

- Appear due to
  - partial axiomatizations
  - frame conditions
  - assertions
- Undecidable in general
- A few decidability results
  - based on finite instantiations
  - brittle