CMPT 745
Software Engineering

Symbolic Execution

Nick Sumner
wsumner@sfu.ca

Symbolic Execution

- As we have seen, building constraints that model code can be useful

Symbolic Execution

- As we have seen, building constraints that model code can be useful
- With care, we can even try to generate all inputs that are “interesting”

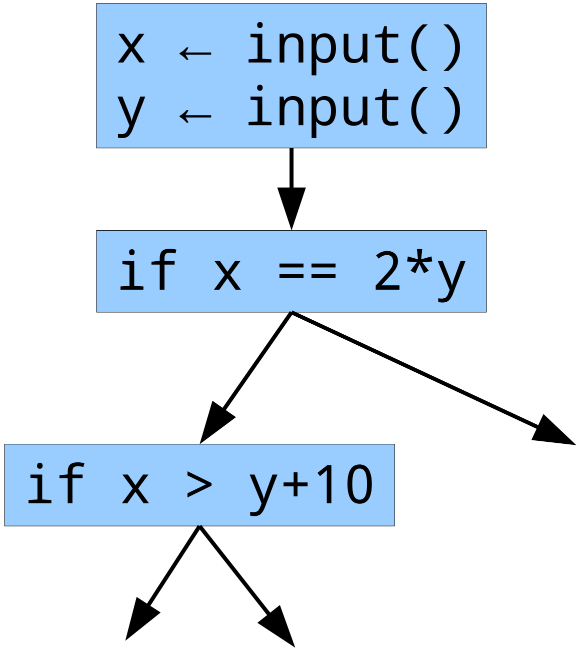
Symbolic Execution

- As we have seen, building constraints that model code can be useful
- With care, we can even try to generate all inputs that are "interesting"
- Techniques for supporting this are known as ***symbolic execution***
  - (SymEx)

Symbolic Execution

- An approach for generating test inputs.

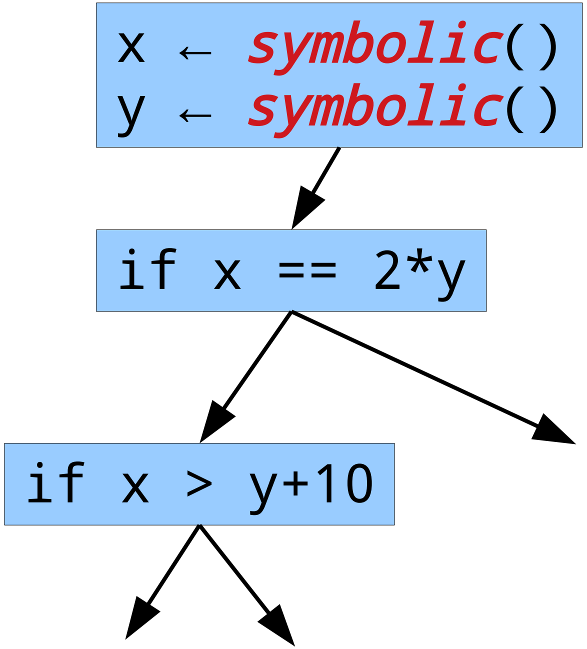
[Cadar & Sen, 2013]



Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values

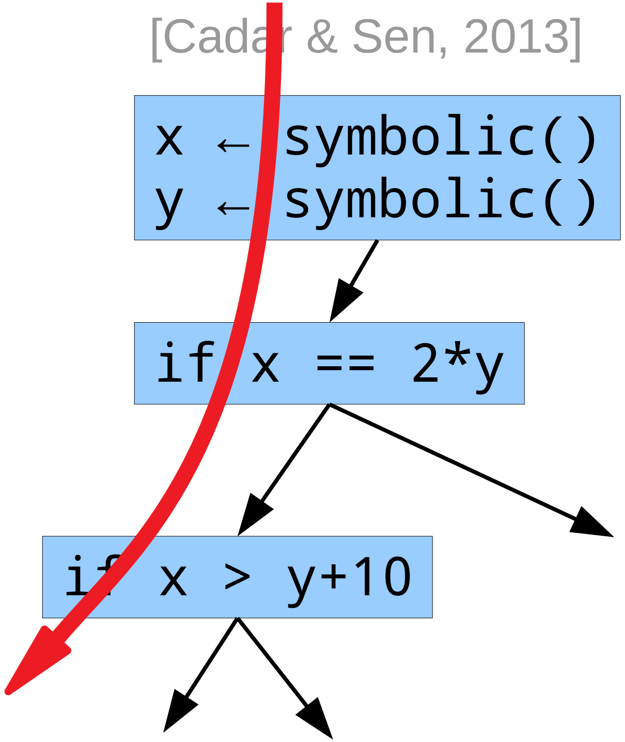
[Cadar & Sen, 2013]



Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.

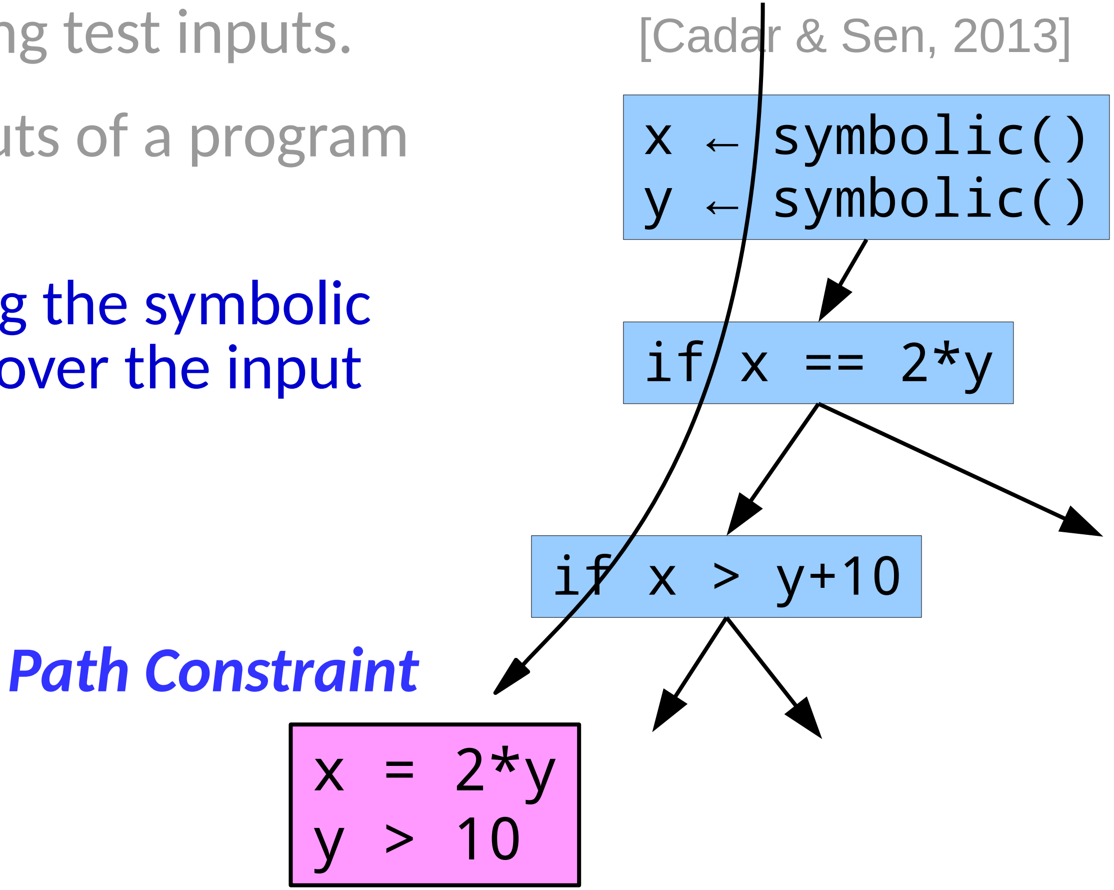
[Cadar & Sen, 2013]



Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.

[Cadar & Sen, 2013]

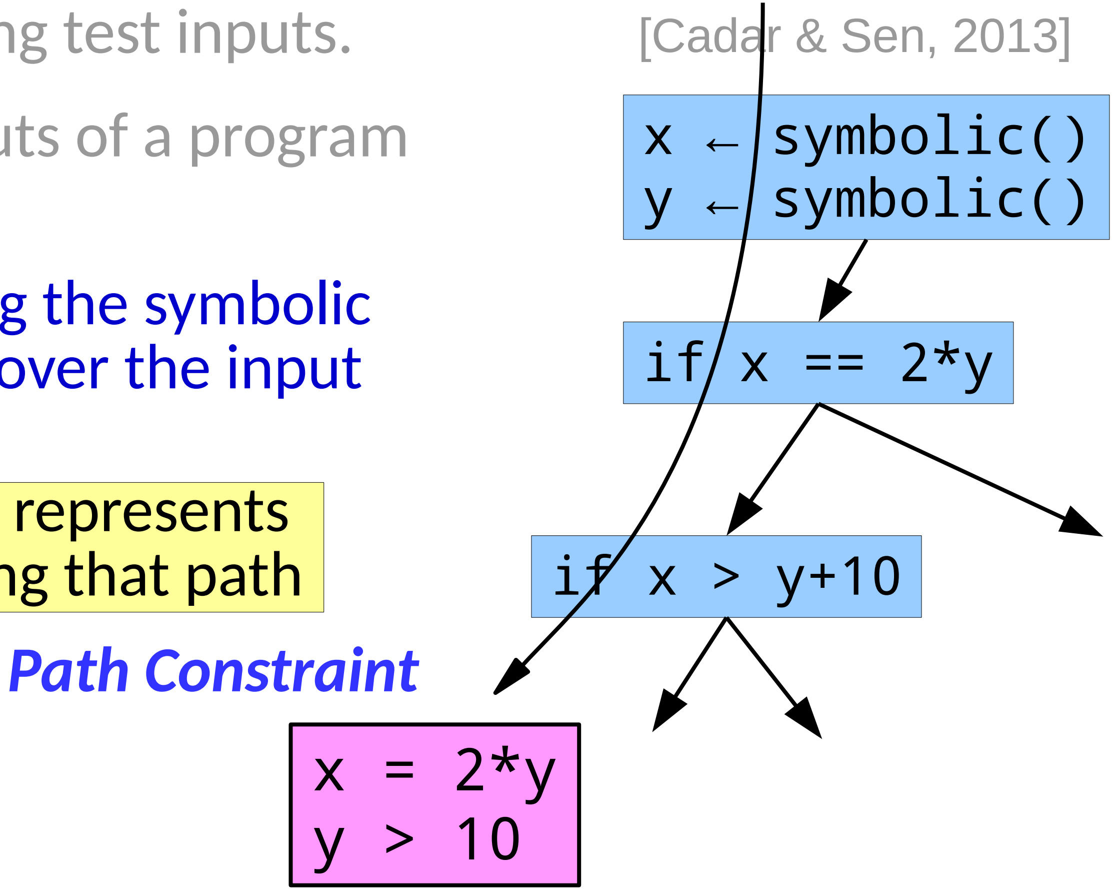


Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.

A path constraint represents all executions along that path

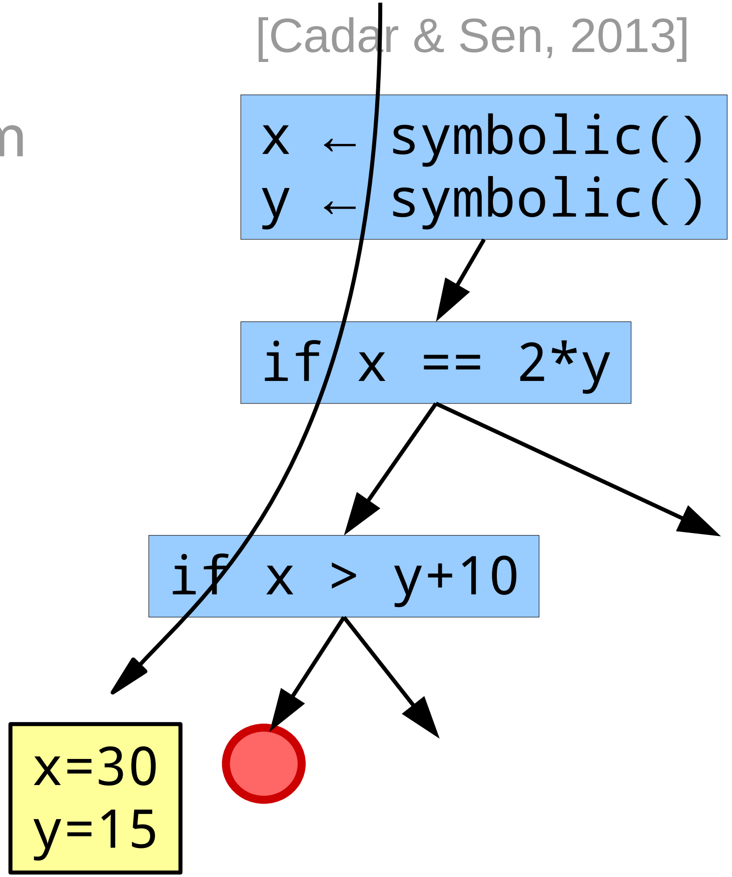
[Cadar & Sen, 2013]



Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.
- Solve for the symbolic symbols to find inputs that yield the path.

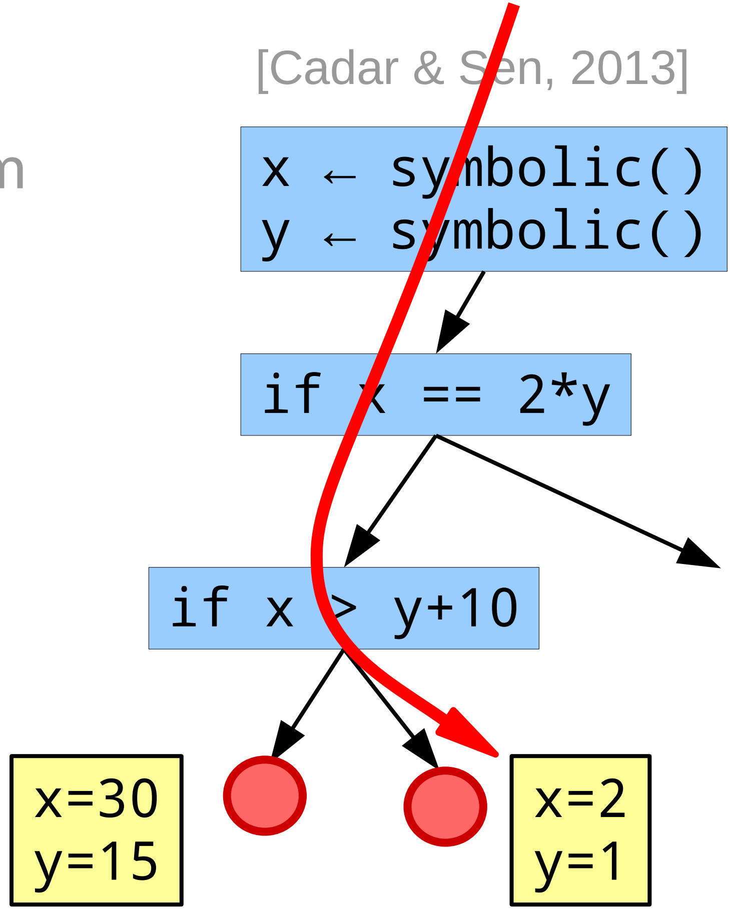
[Cadar & Sen, 2013]



Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.
- Solve for the symbolic symbols to find inputs that yield the path.

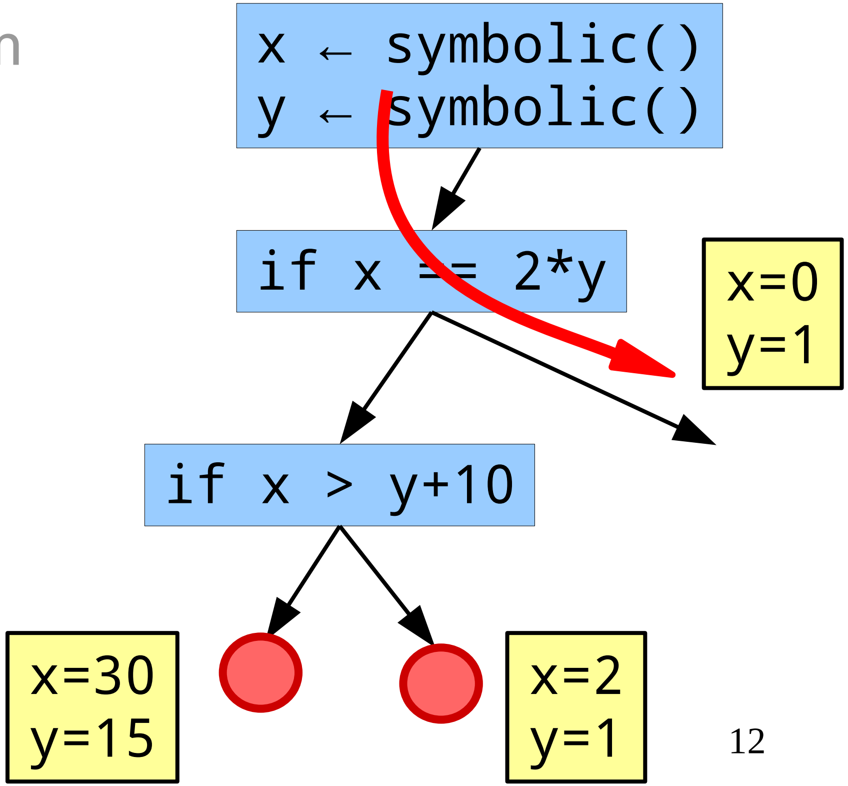
[Cadar & Sen, 2013]



Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.
- Solve for the symbolic symbols to find inputs that yield the path.

[Cadar & Sen, 2013]



Using SymEx to solve problems

- Note that we described SymEx over *traces*.

Using SymEx to solve problems

- Note that we described SymEx over *traces*.
  - This is dynamic symbolic execution.
  - What we saw before was essentially static symbolic execution.

Using SymEx to solve problems

- Note that we described SymEx over *traces*.
  - This is dynamic symbolic execution.
  - What we saw before was essentially static symbolic execution.
- Applying constraint based reasoning on traces can also yield insights

Using SymEx to solve problems

- Note that we described SymEx over *traces*.
  - This is dynamic symbolic execution.
  - What we saw before was essentially static symbolic execution.
- Applying constraint based reasoning on traces can also yield insights
  - e.g. Suppose you are given two versions of a program v_1,v_2

Using SymEx to solve problems

- Note that we described SymEx over *traces*.
  - This is dynamic symbolic execution.
  - What we saw before was essentially static symbolic execution.
- Applying constraint based reasoning on traces can also yield insights
  - e.g. Suppose you are given two versions of a program v_1,v_2 and constraints on output φ_i in each from an input I

Using SymEx to solve problems

- Note that we described SymEx over *traces*.
  - This is dynamic symbolic execution.
  - What we saw before was essentially static symbolic execution.
- Applying constraint based reasoning on traces can also yield insights
  - e.g. Suppose you are given two versions of a program v_1,v_2 and constraints on output φ_i in each from an input I

What is $wp(\varphi_1) \land \neg wp(\varphi_2)$?

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

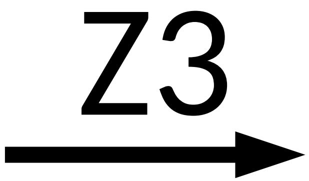
Z3 →

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```




```
sat
(model
   (define-fun y () Int 11)
   (define-fun x () Int 22)
)
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

Z3 →

```
x=22
y=11
```

```
sat
(model
   (define-fun y () Int 11)
   (define-fun x () Int 22)
)
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

Z3 →

```
x=22
y=11
```

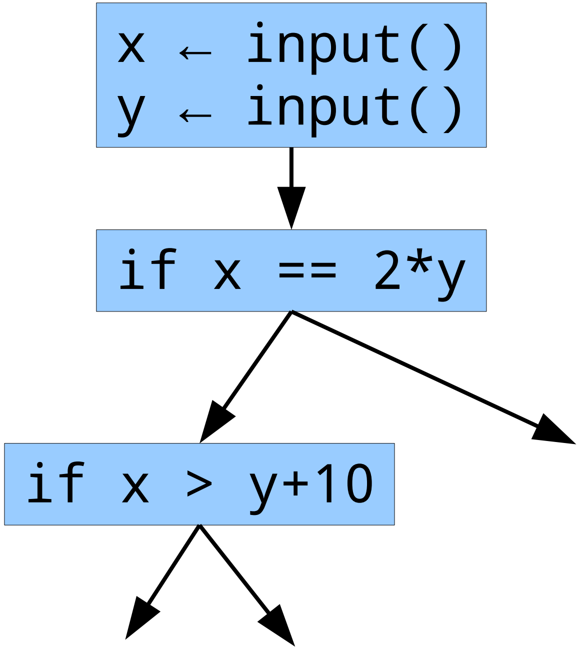
```
sat
(model
   (define-fun y () Int 11)
   (define-fun x () Int 22)
)
```

Try it online:
http://www.rise4fun.com/Z3/tutorial/

Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.

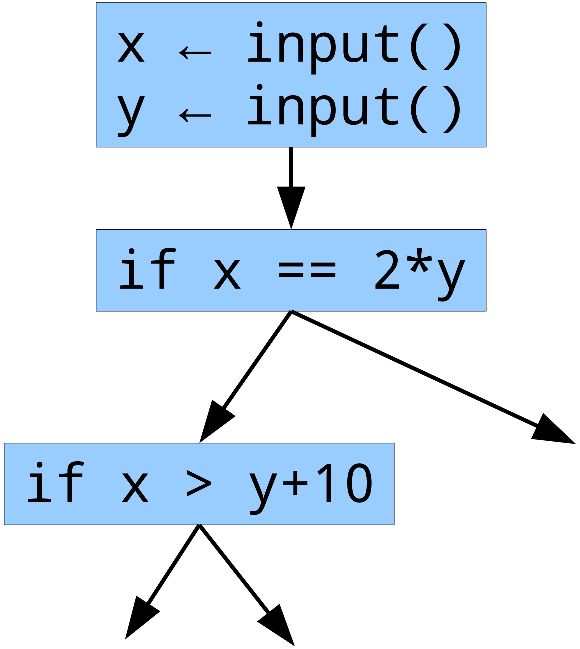
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.

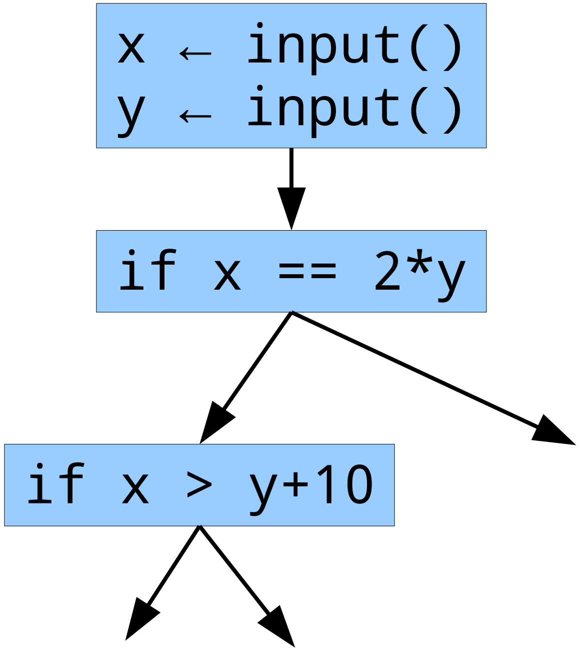
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches

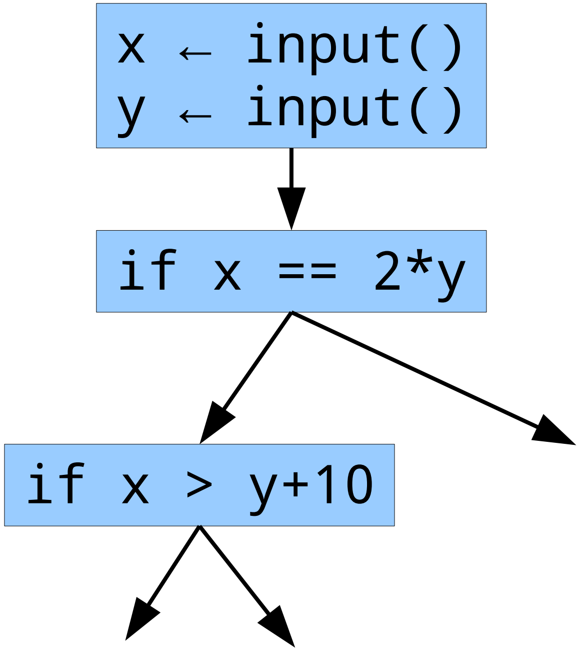
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

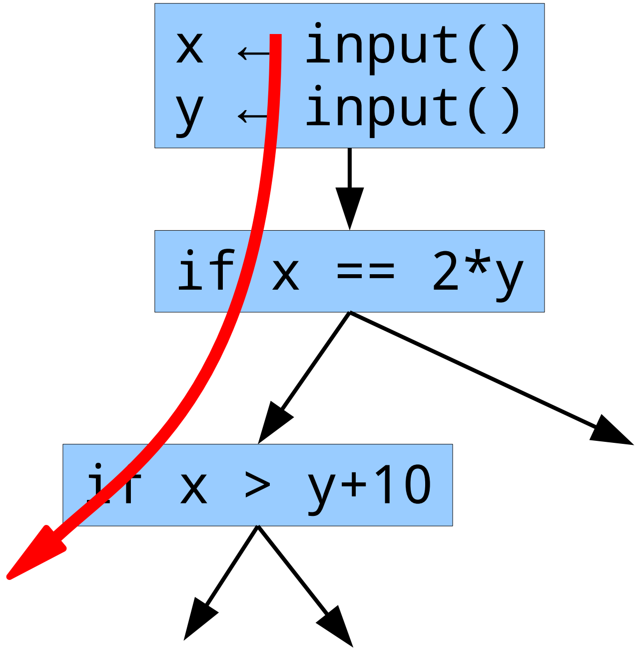
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

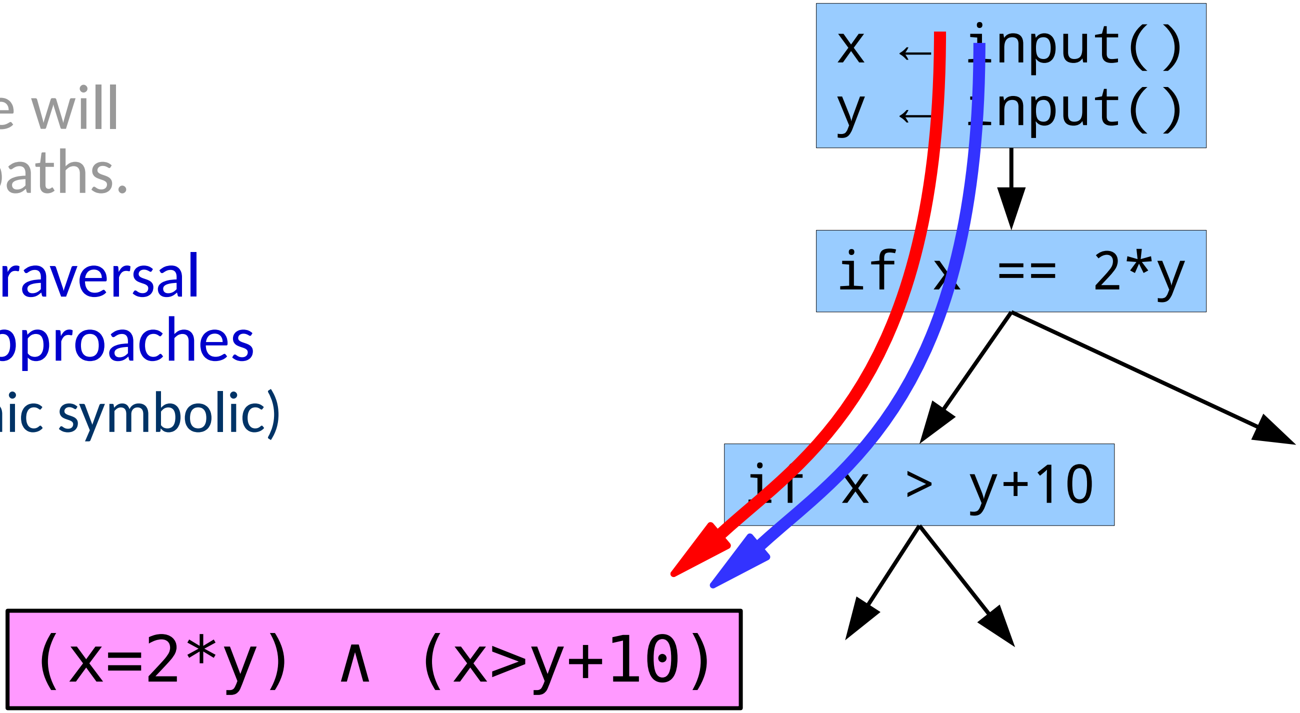
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

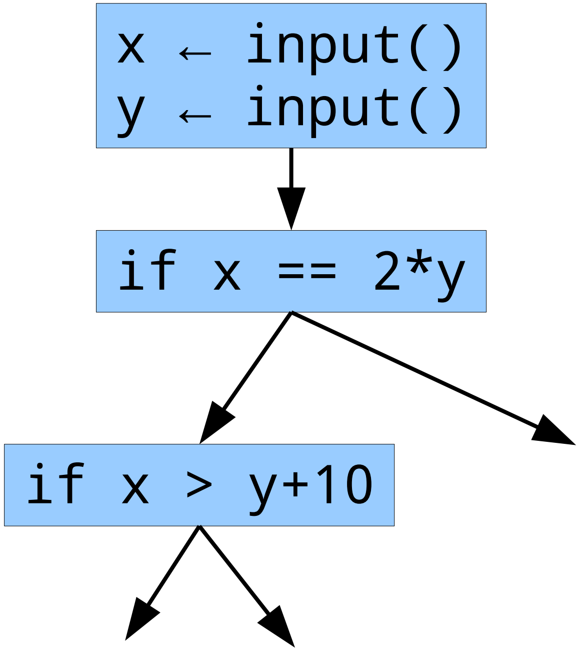
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

[Cadar & Sen, 2013]

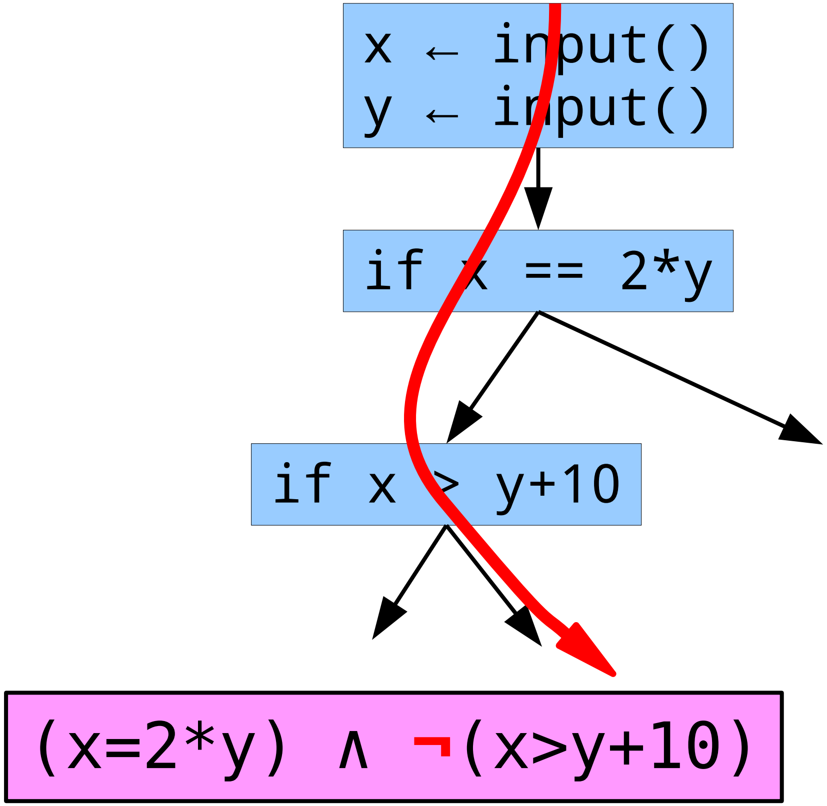


$(x=2*y) \wedge \neg(x>y+10)$

Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

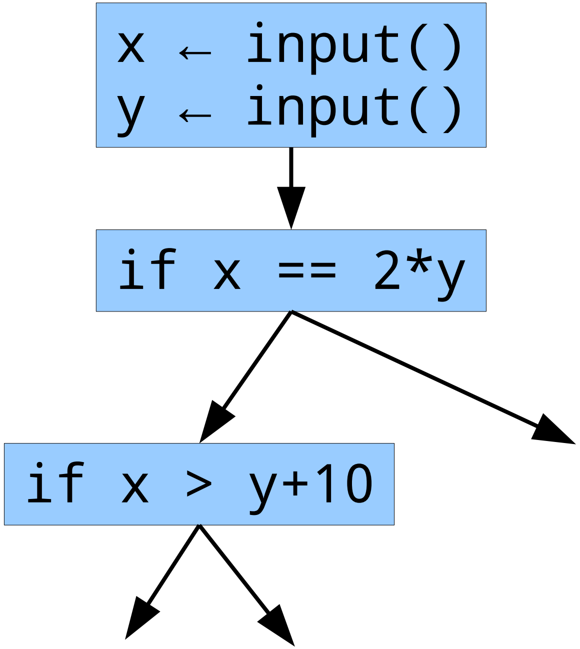
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)
  - Execution Generated Testing

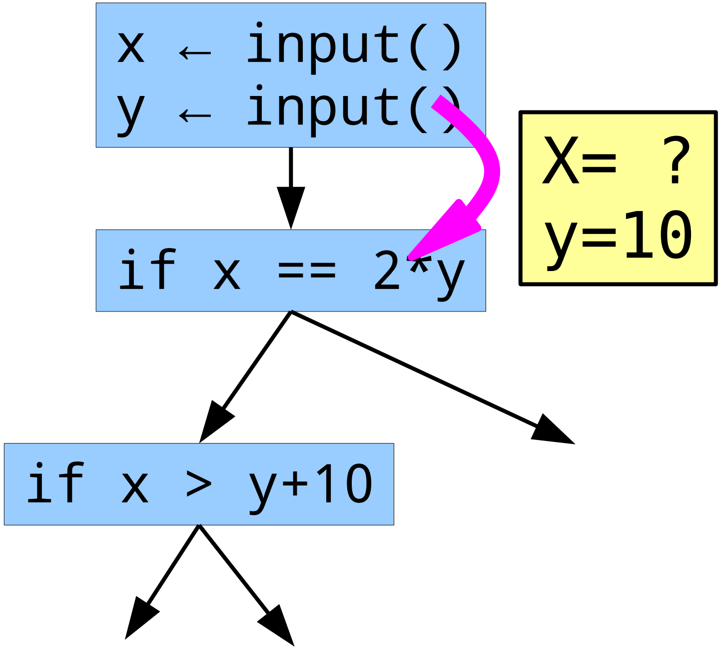
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)
  - Execution Generated Testing

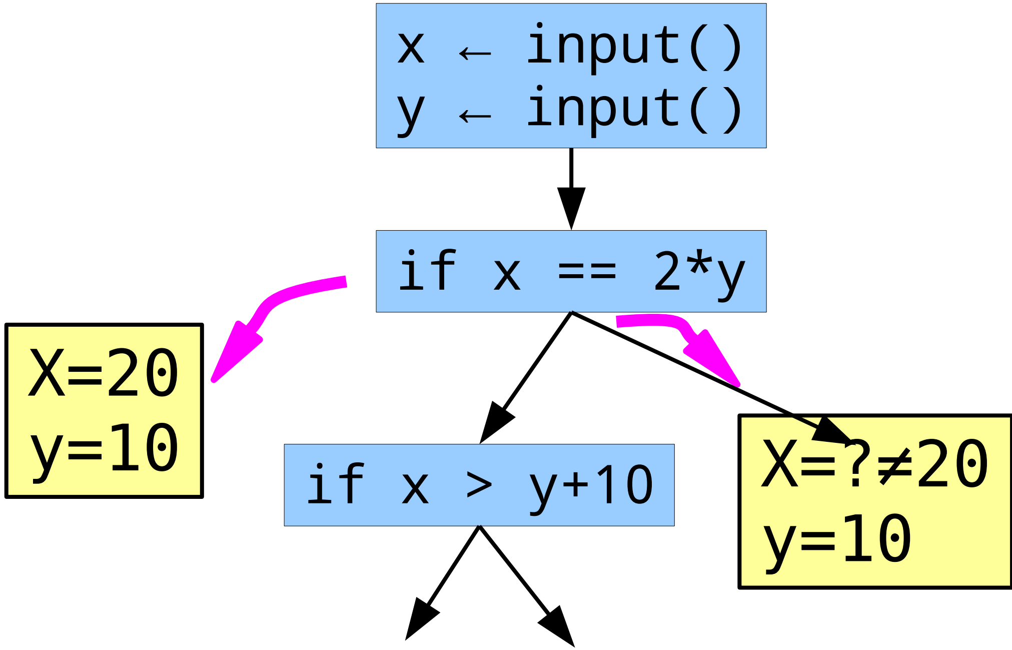
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)
  - Execution Generated Testing

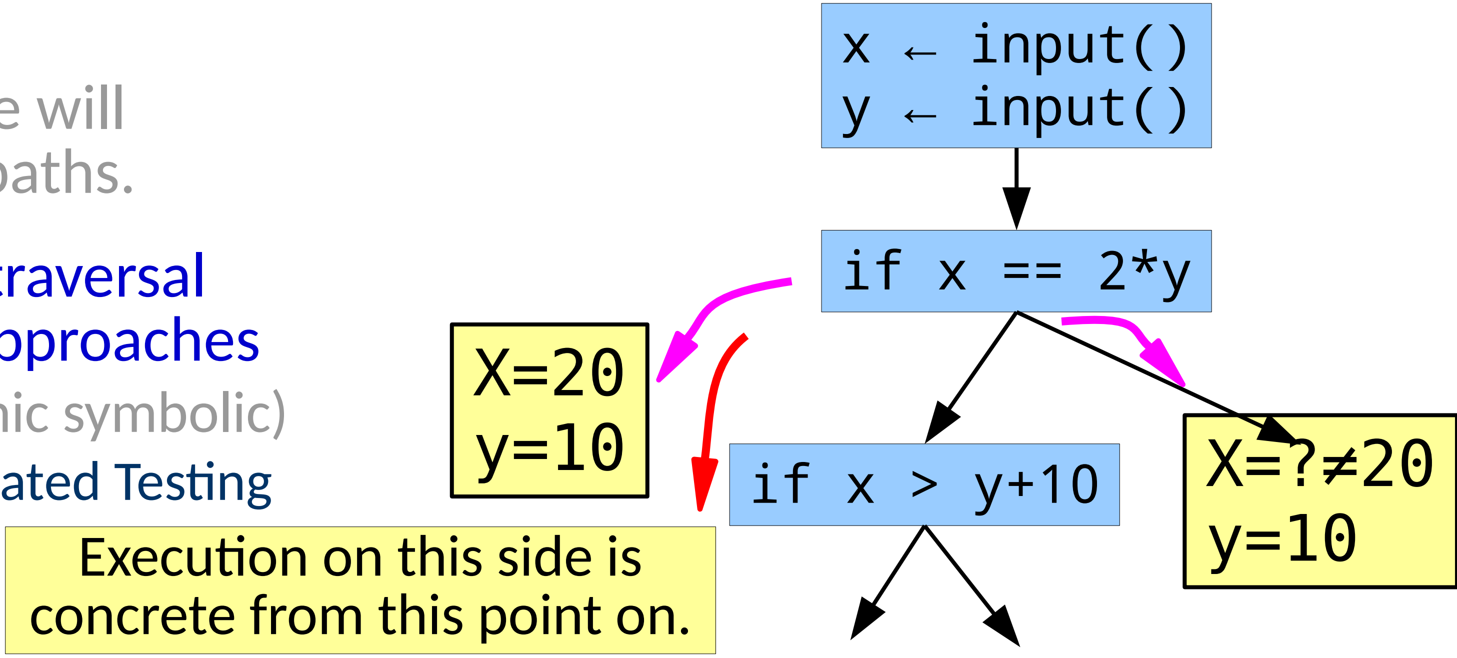
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)
  - Execution Generated Testing

[Cadar & Sen, 2013]



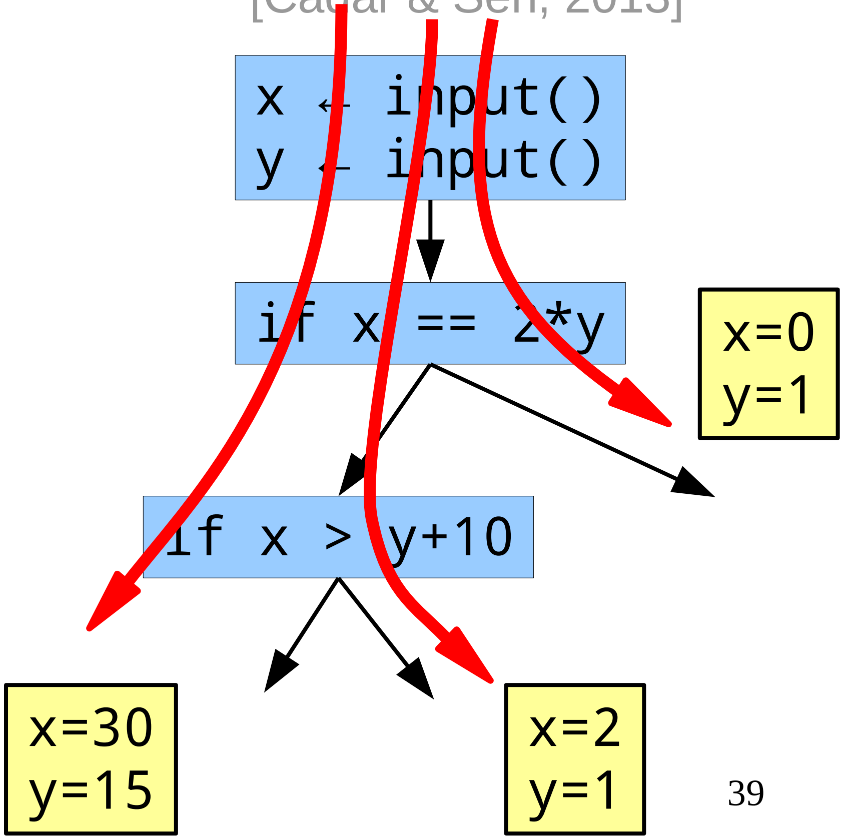
(Some) Applications

- Constructing test suites

(Some) Applications

- Constructing test suites

[Cadar & Sen, 2013]

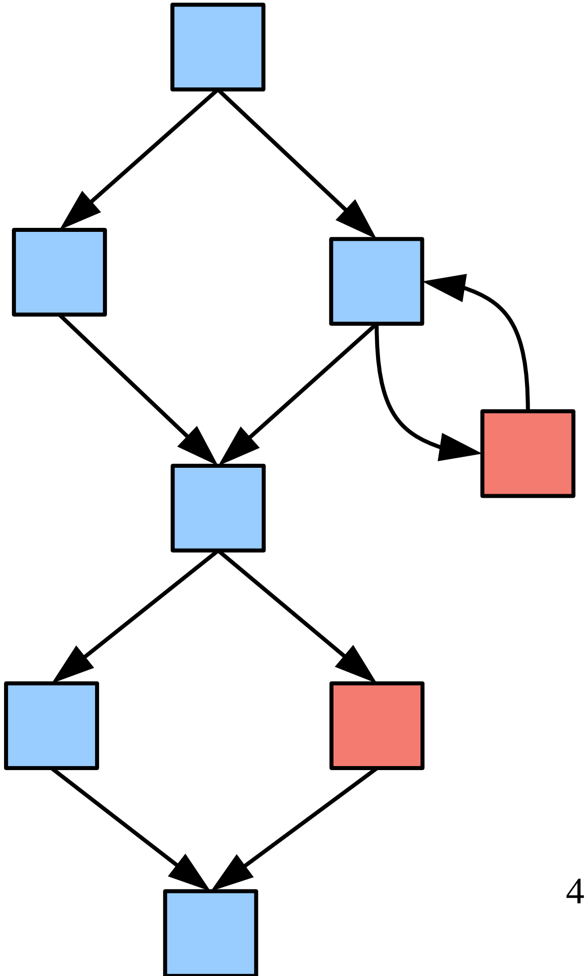


(Some) Applications

- Constructing test suites
- Targeted tests

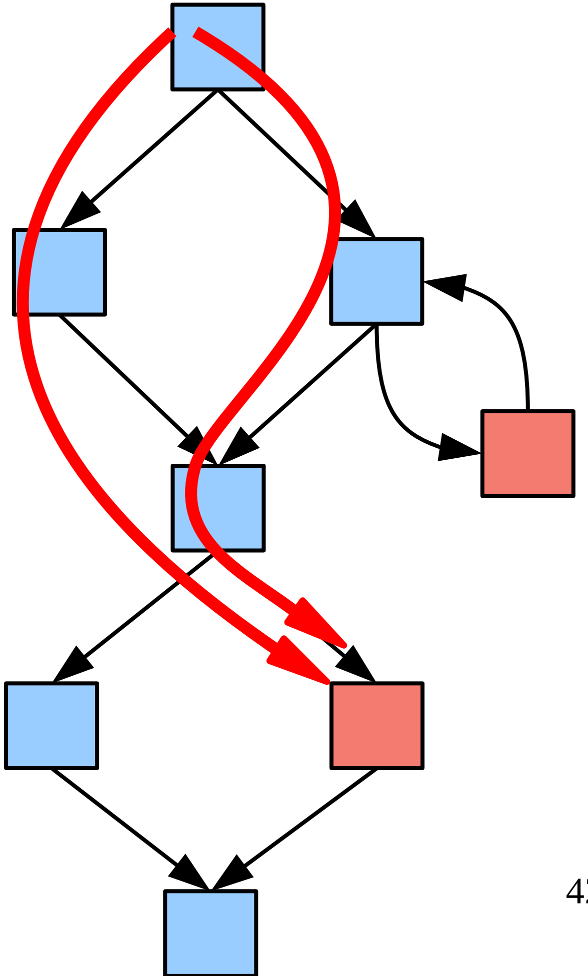
(Some) Applications

- Constructing test suites
- Targeted tests



(Some) Applications

- Constructing test suites
- Targeted tests

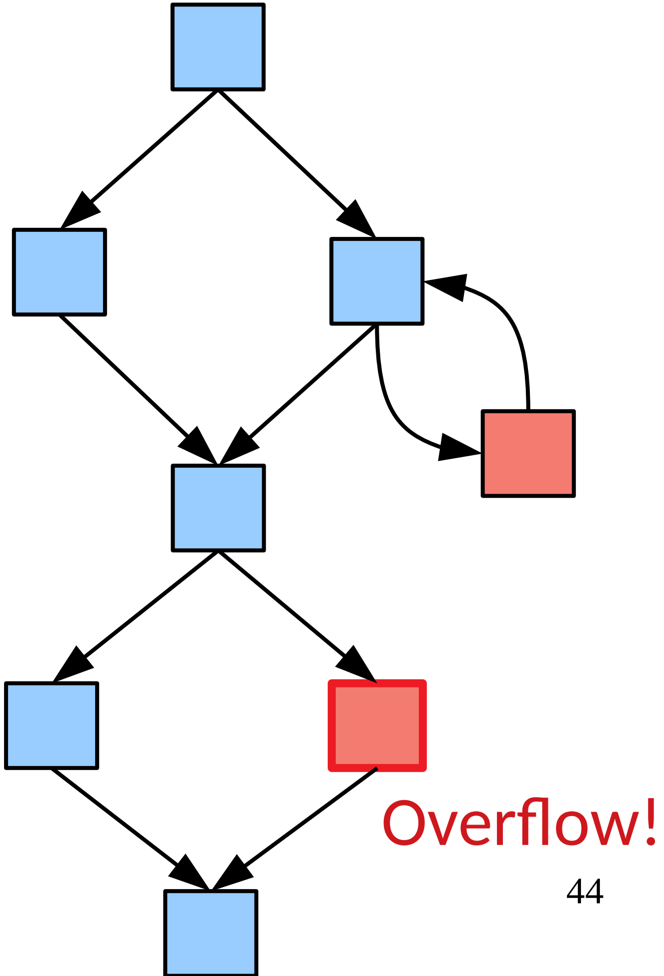


(Some) Applications

- Constructing test suites
- Targeted tests
- Automated exploit discovery & synthesis

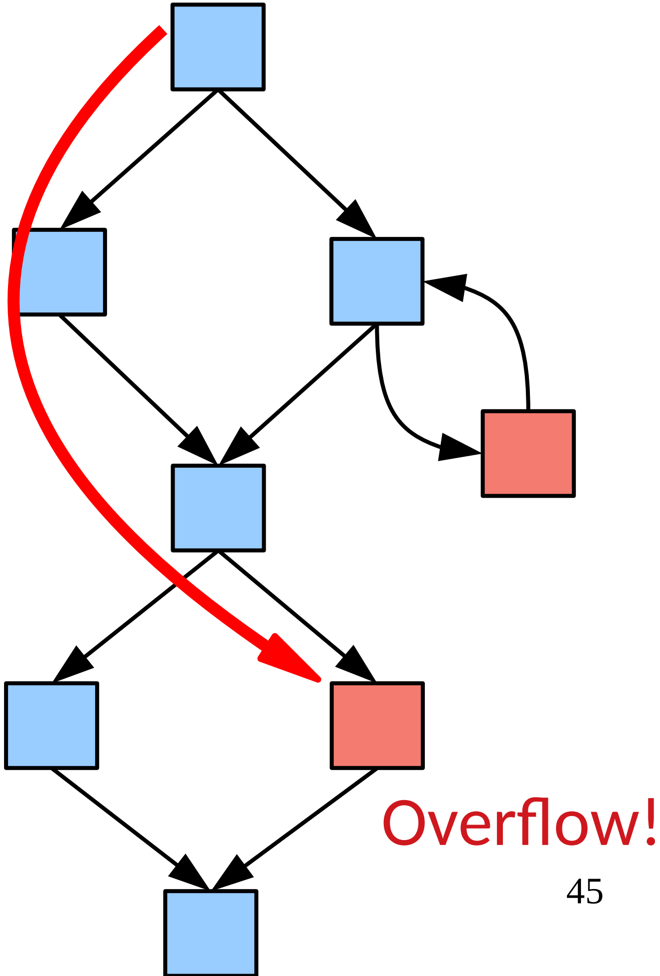
(Some) Applications

- Constructing test suites
- Targeted tests
- Automated exploit discovery & synthesis



(Some) Applications

- Constructing test suites
- Targeted tests
- Automated exploit discovery & synthesis

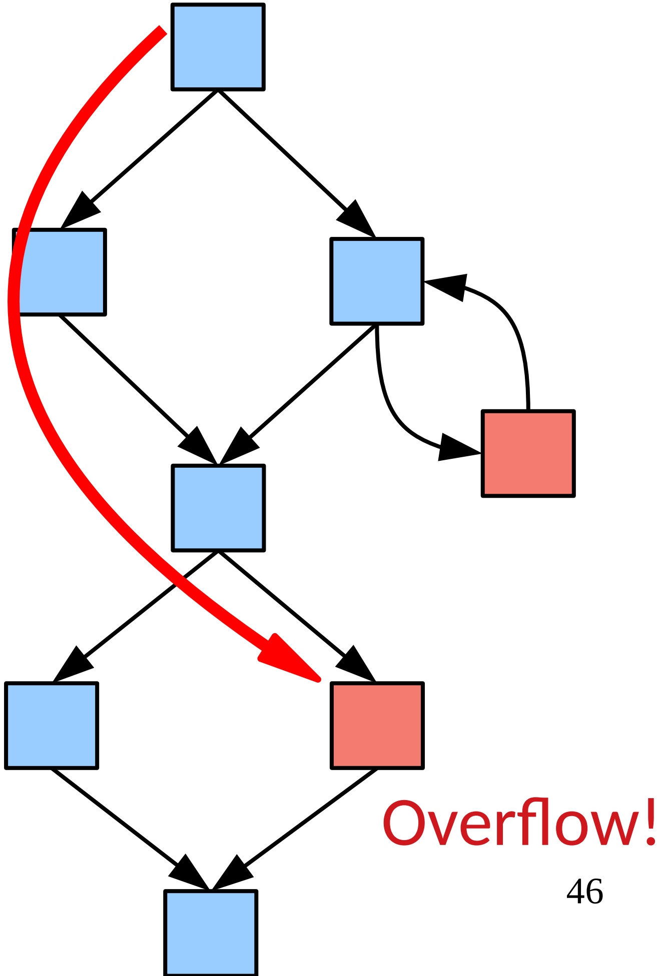


(Some) Applications

- Constructing test suites
- Targeted tests
- Automated exploit discovery & synthesis

Input ⊢ Overflow ^ StartsShellcode

This is the core process for
Darpa Cybersecurity Grand Challenge entries!



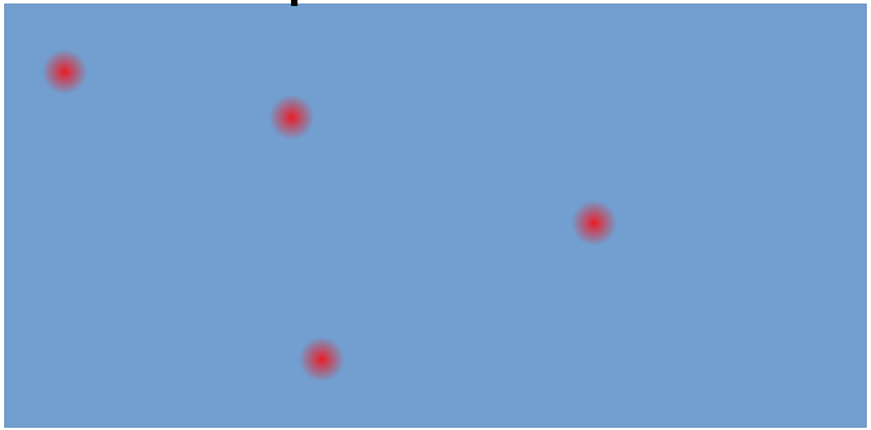
(Some) Applications

- Constructing test suites
- Targeted tests
- Automated exploit discovery & synthesis
- Test driven model checking (Yogi)
- …

Application: Test Driven Model Checking

- While traditional testing is *sampling*,

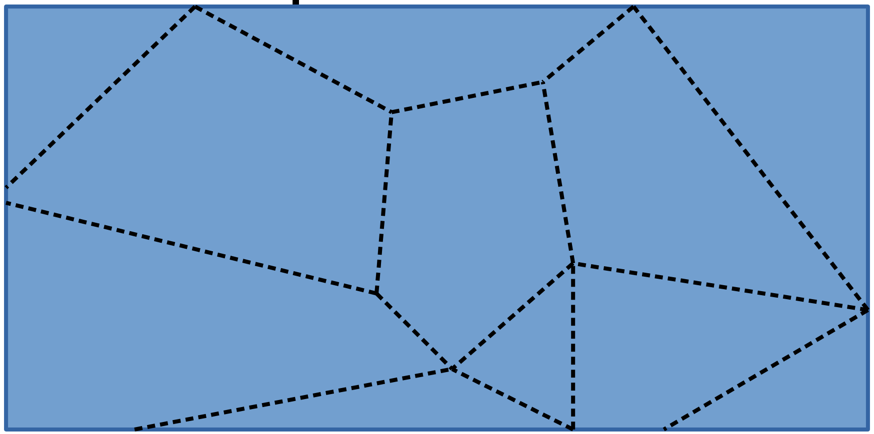
Input Domain



Application: Test Driven Model Checking

- While traditional testing is sampling,
  SymEx enables targeted tests that answer questions about a program

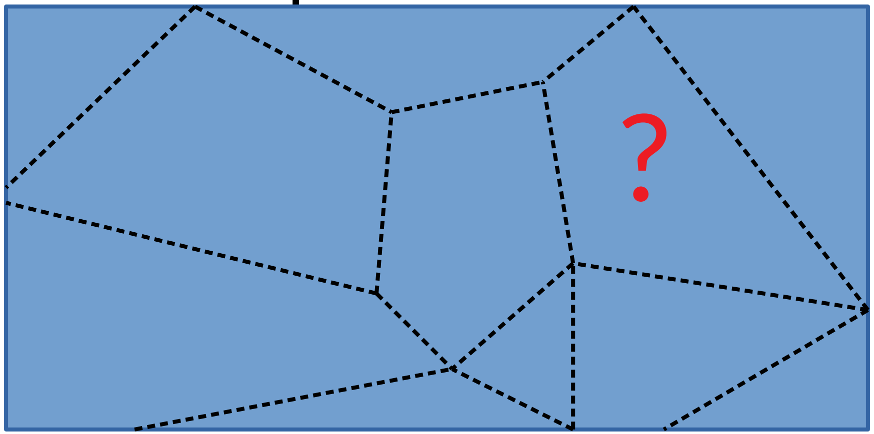
Input Domain



Application: Test Driven Model Checking

- While traditional testing is sampling,
  SymEx enables targeted tests that answer questions about a program

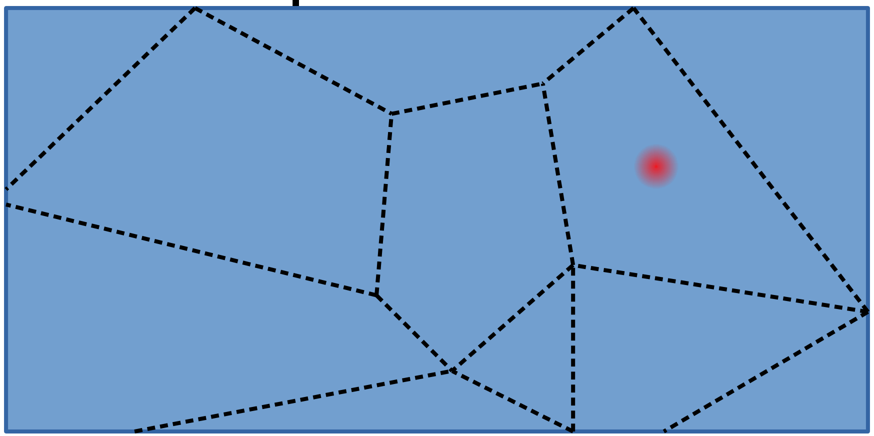
Input Domain



Application: Test Driven Model Checking

- While traditional testing is sampling,
  SymEx enables targeted tests that answer questions about a program

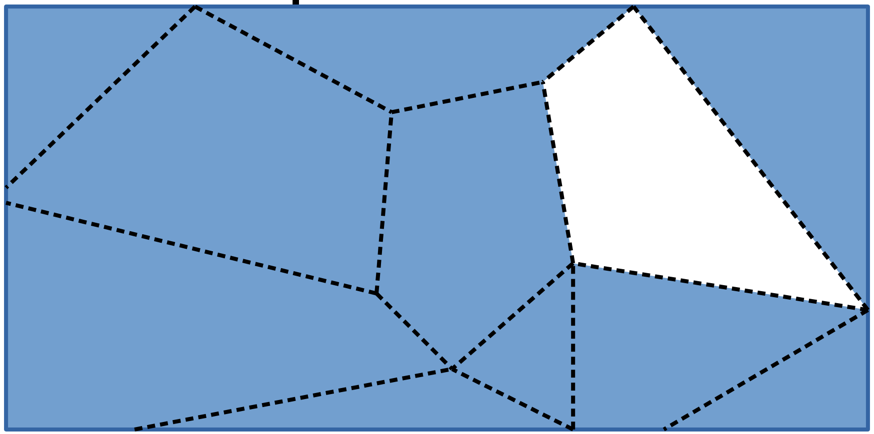
Input Domain



Application: Test Driven Model Checking

- While traditional testing is sampling,
  SymEx enables targeted tests that answer questions about a program

Input Domain

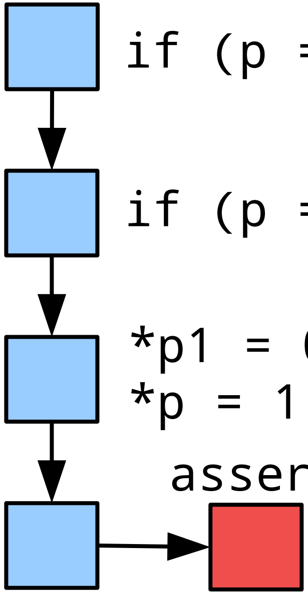


Application: Test Driven Model Checking

- While traditional testing is sampling,
  SymEx enables targeted tests that answer questions about a program
- Carefully choosing which questions to ask can allow us to prove properties of programs!

Application: Test Driven Model Checking

- While traditional testing is sampling,
  SymEx enables targeted tests that answer questions about a program
- Carefully choosing which questions to ask can allow us to prove properties of programs!

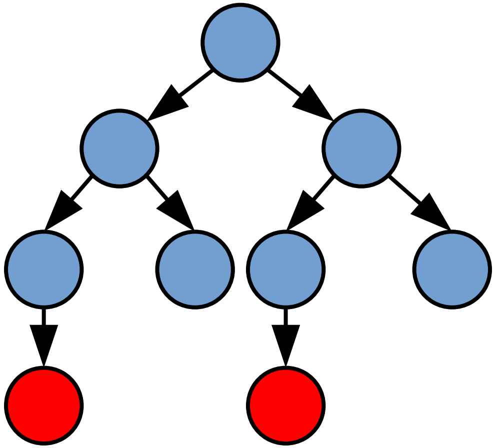


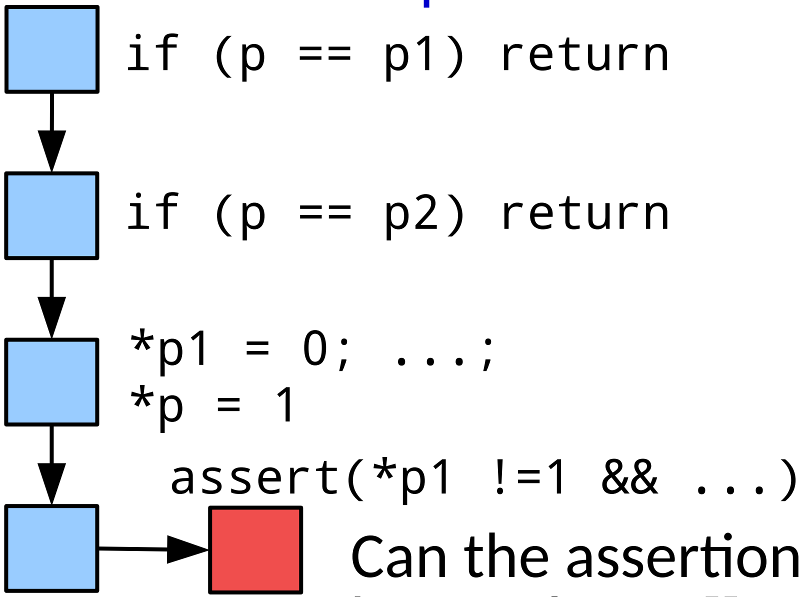
Can the assertion here fail?

Application: Test Driven Model Checking

- While traditional testing is sampling,
  SymEx enables targeted tests that answer questions about a program
- Carefully choosing which questions to ask can allow us to prove properties of programs!

Execution Tree



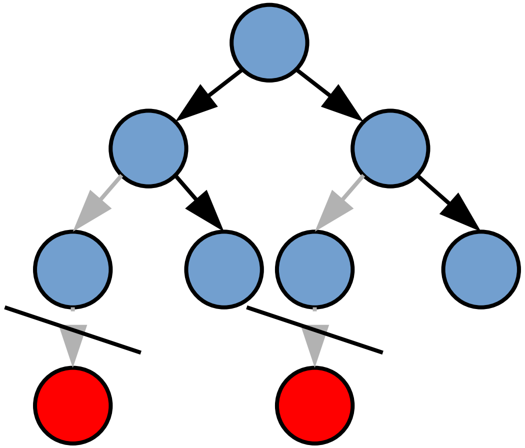


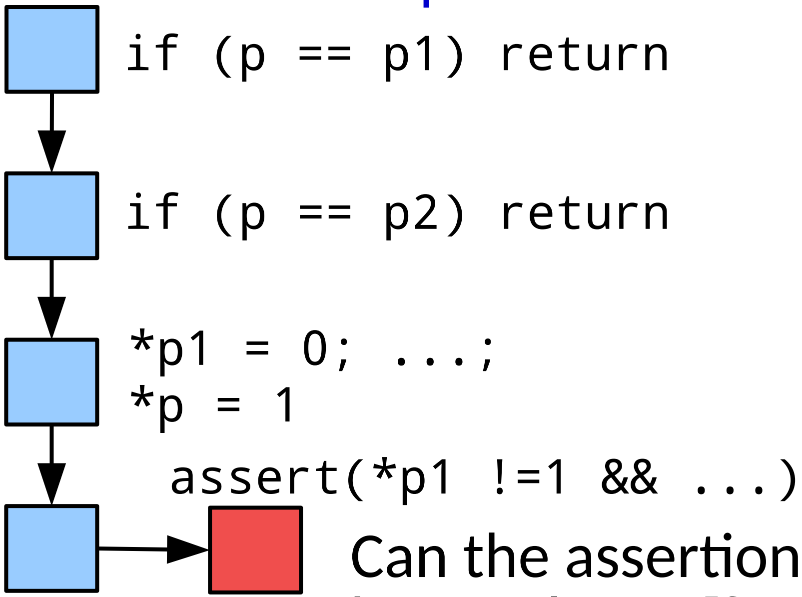
Can the assertion here fail?

Application: Test Driven Model Checking

- While traditional testing is sampling,
  SymEx enables targeted tests that answer questions about a program
- Carefully choosing which questions to ask can allow us to prove properties of programs!

Execution Tree



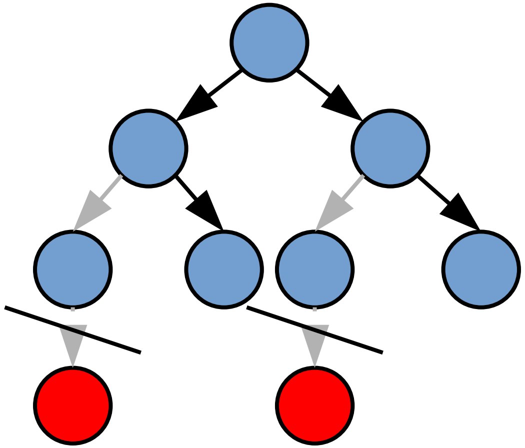


Can the assertion here fail?

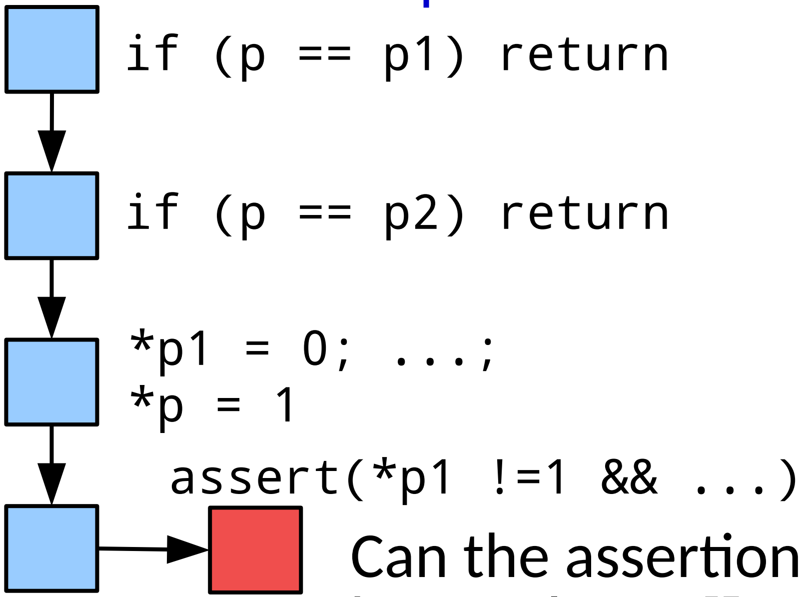
Application: Test Driven Model Checking

- While traditional testing is sampling,
  SymEx enables targeted tests that answer questions about a program
- Carefully choosing which questions to ask can allow us to prove properties of programs!

Execution Tree



Do you see any potential problems with this approach as given?



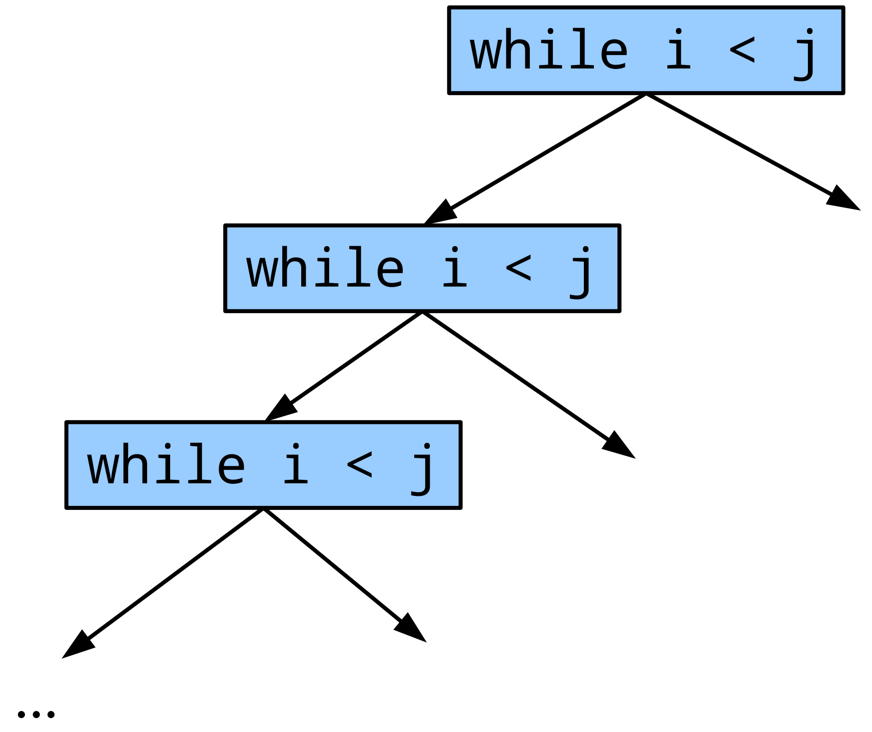
Can the assertion here fail?

Challenges

- Path Explosion
- Challenging constraints
- Constraint representations & domain knowledge

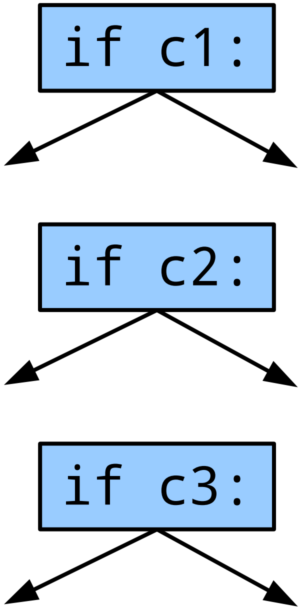
Path Explosion

- Loops



Path Explosion

- Loops
- Combinatorial Explosion



Path Explosion

- Loops
- Combinatorial Explosion
- Strategies

Path Explosion

- Loops
- Combinatorial Explosion
- Strategies
  - Search heuristics (DFS, BFS, Targeted, Merged, ...)

Path Explosion

- Loops
- Combinatorial Explosion
- Strategies
  - Search heuristics
  - Memoization (Have we already analyzed this?)

Challenging Constraints

- Intuitively, we cannot solve all constraints

```
if hash(password) == y:
  print(“how odd”)
```

What would it imply if we could?

Challenging Constraints

- Intuitively, we cannot solve all constraints
- How can we address this?

Challenging Constraints

- Intuitively, we cannot solve all constraints
- How can we address this?
  - IDEA: Observe the actual values of variables in runs we have

password = fritter
hash(password) = HJdjdskS&8sdh

```
if hash(password) == y:
  print("how odd")
```

Challenging Constraints

- Intuitively, we cannot solve all constraints
- How can we address this?
  - IDEA: Observe the actual values of variables in runs we have
    Substitute those observed values in challenging runs in the future

password = fritter
hash(password) = HjdjdskS&8sdh
y = HjdjdskS&8sdh

```
if hash(password) == y:
  print(“how odd”)
```

Challenging Constraints

- Intuitively, we cannot solve all constraints
- How can we address this?
  - IDEA: Observe the actual values of variables in runs we have
    Substitute those observed values in challenging runs in the future
  - Build a library of (input,output) pairs for challenging expressions
    (Use the theory of uninterpreted functions!)

Challenging Constraints

- Intuitively, we cannot solve all constraints
- How can we address this?
  - IDEA: Observe the actual values of variables in runs we have
    Substitute those observed values in challenging runs in the future
  - Build a library of (input,output) pairs for challenging expressions
    (Use the theory of uninterpreted functions!)

How do these affect our ability to explore the execution tree?

Domain Knowledge

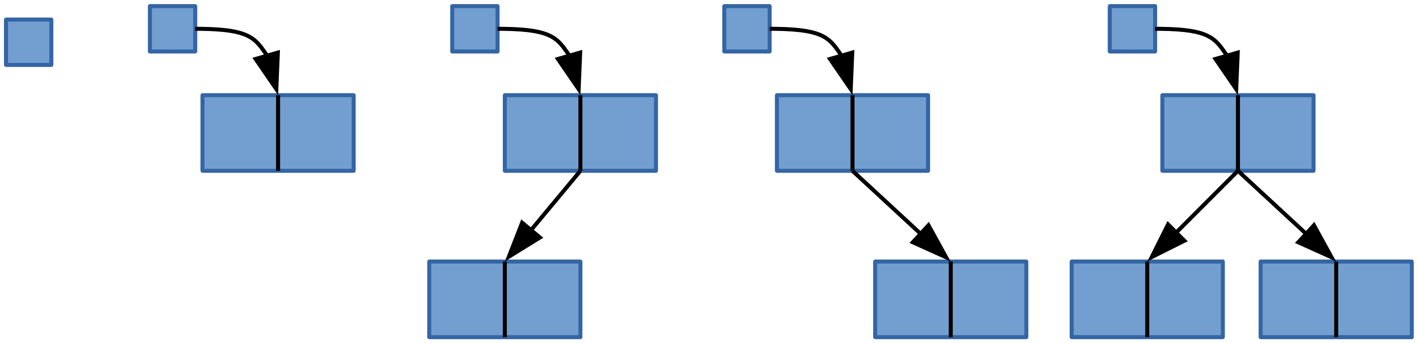
- How should we represent memory?

Domain Knowledge

- How should we represent memory?
  - A linear arrangement of memory?
  - Combinatorial aliasing relation pairs?

Domain Knowledge

- How should we represent memory?
  - A linear arrangement of memory?
  - Combinatorial aliasing relation pairs?
- Can we carefully explore interesting structures?
  - Korat style enumeration



Domain Knowledge

- How should we represent memory?
  - A linear arrangement of memory?
  - Combinatorial aliasing relation pairs?
- Can we carefully explore interesting structures?
  - Korat style enumeration
- Can we use more constrained problems than SAT/SMT?
  - Many problems can use simpler *constrained Horn clauses*

Interesting Directions

- How can we speed up or remove symbolic operations?

Interesting Directions

- How can we speed up or remove symbolic operations?
  - Neural strategies

Interesting Directions

- How can we speed up or remove symbolic operations?
  - Neural strategies
  - Indexing & memoization

Interesting Directions

- How can we speed up or remove symbolic operations?
  - Neural strategies
  - Indexing & memoization
  - Aggressively using theory of uninterpreted functions

Interesting Directions

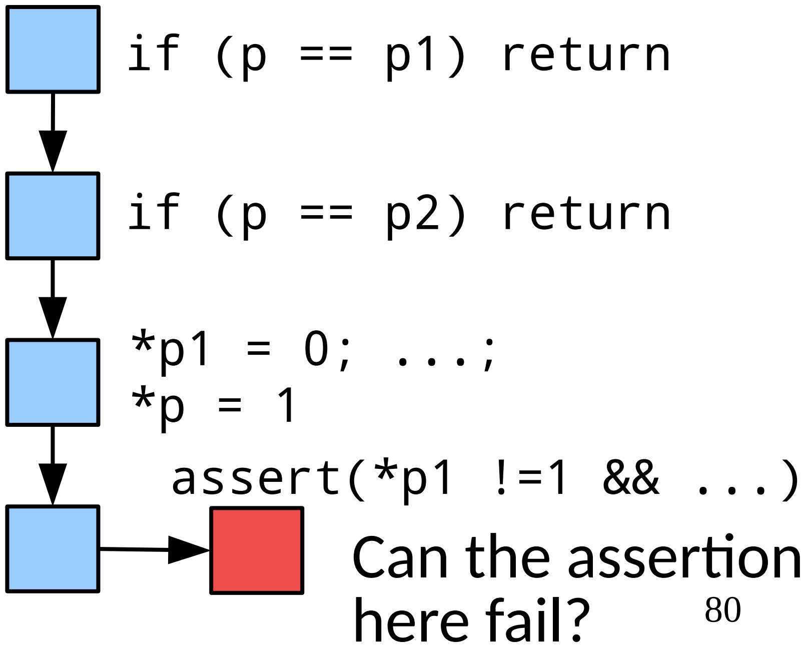
- How can we speed up or remove symbolic operations?
  - Neural strategies
  - Indexing & memoization
  - Aggressively using theory of uninterpreted functions
- Probabilistic Symbolic Execution
  - How *likely* is one path vs another?
  - Can that direct our search toward more interesting areas?

Interesting Directions

- How can we speed up or remove symbolic operations?
  - Neural strategies
  - Indexing & memoization
  - Aggressively using theory of uninterpreted functions
- Probabilistic Symbolic Execution
  - How *likely* is one path vs another?
  - Can that direct our search toward more interesting areas?
- Decomposing goals into smaller problems
  - How can we analyze systems like Linux, Chrome, & Firefox well?
    [Brown 2020]

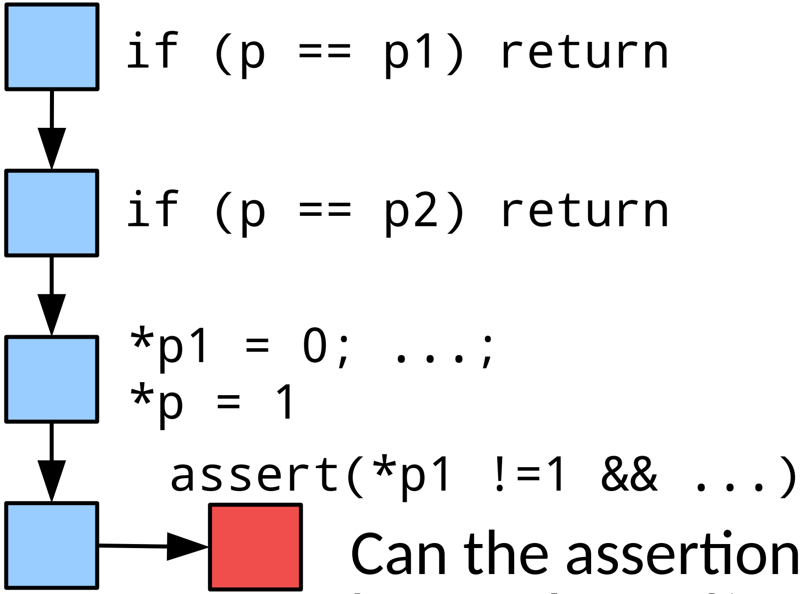
Revisit: Test Driven Model Checking

- Carefully choosing which questions to ask can allow us to prove properties of programs!



Revisit: Test Driven Model Checking

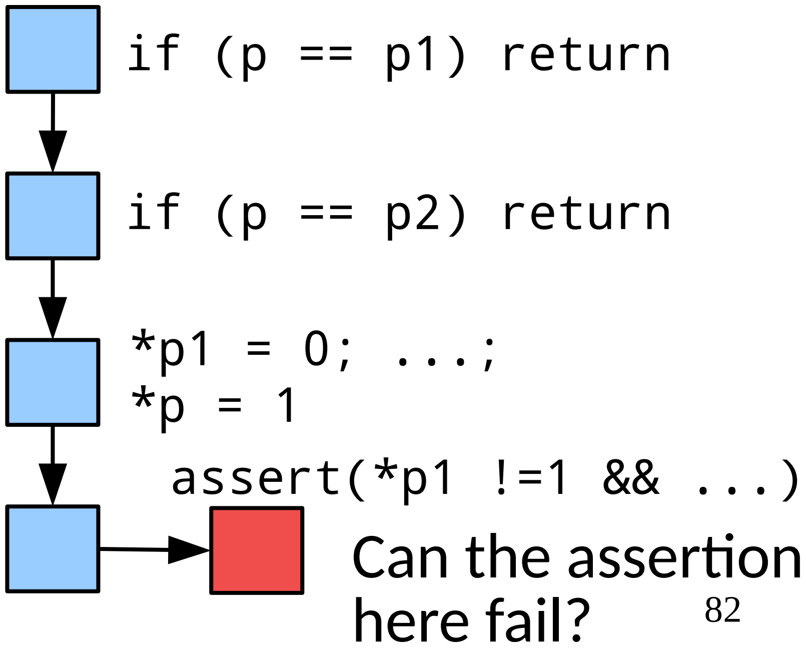
- Carefully choosing which questions to ask can allow us to prove properties of programs!
  - Some relationships may be hard or missing



Can the assertion here fail?

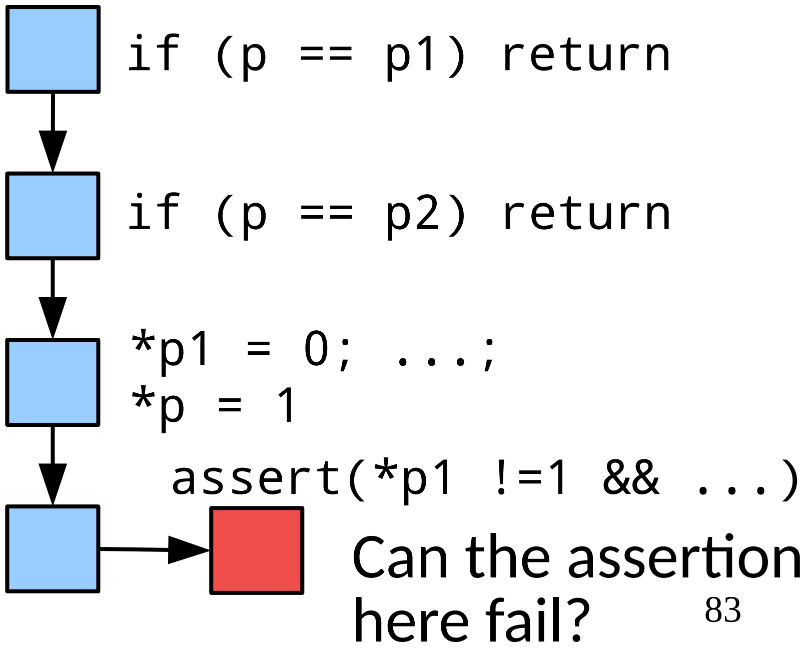
Revisit: Test Driven Model Checking

- Carefully choosing which questions to ask can allow us to prove properties of programs!
  - Some relationships may be hard or missing
  - Full combinatorial search will not scale



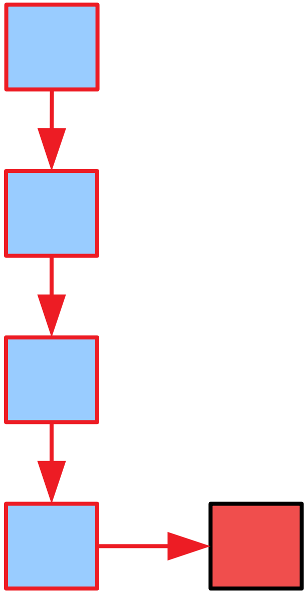
Revisit: Test Driven Model Checking

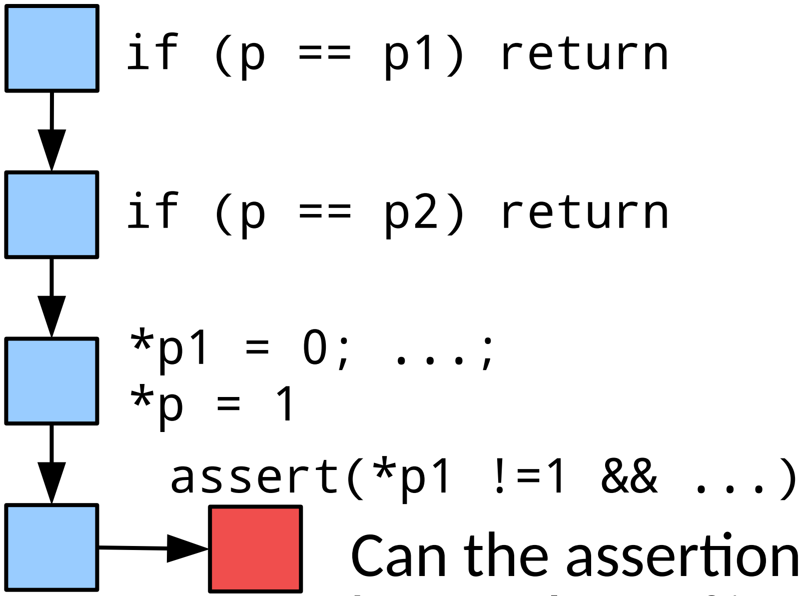
- Carefully choosing which questions to ask can allow us to prove properties of programs!
  - Some relationships may be hard or missing
  - Full combinatorial search will not scale
  - We still want a proof of correctness



Revisit: Test Driven Model Checking

- Carefully choosing which questions to ask can allow us to prove properties of programs!



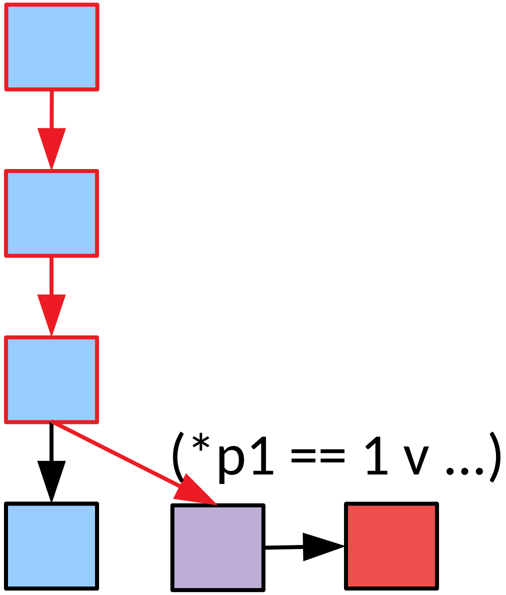


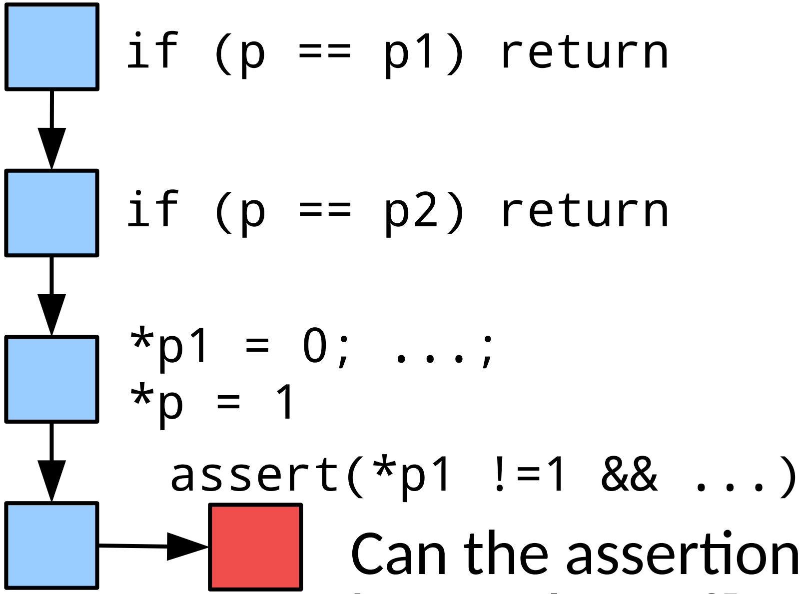
Can the assertion here fail?

[Beckman, TSE 2016]

Revisit: Test Driven Model Checking

- Carefully choosing which questions to ask can allow us to prove properties of programs!



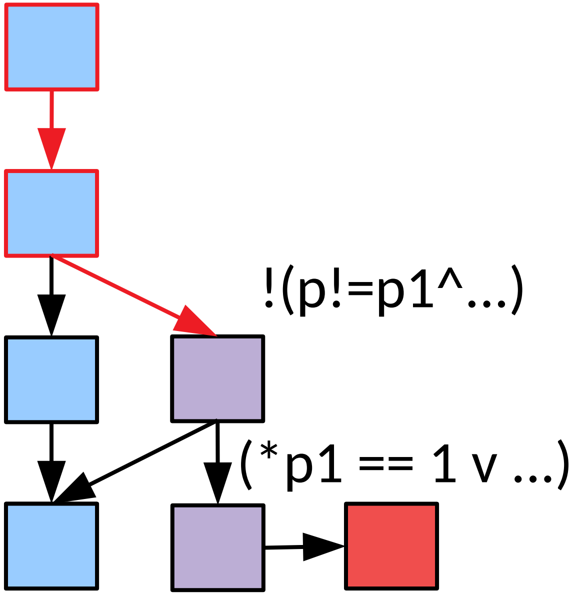


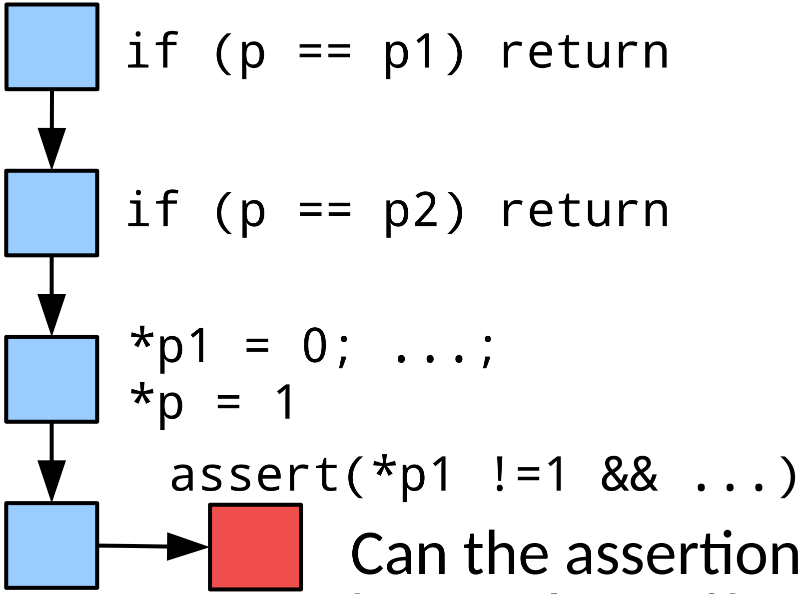
Can the assertion here fail?

[Beckman, TSE 2016]

Revisit: Test Driven Model Checking

- Carefully choosing which questions to ask can allow us to prove properties of programs!



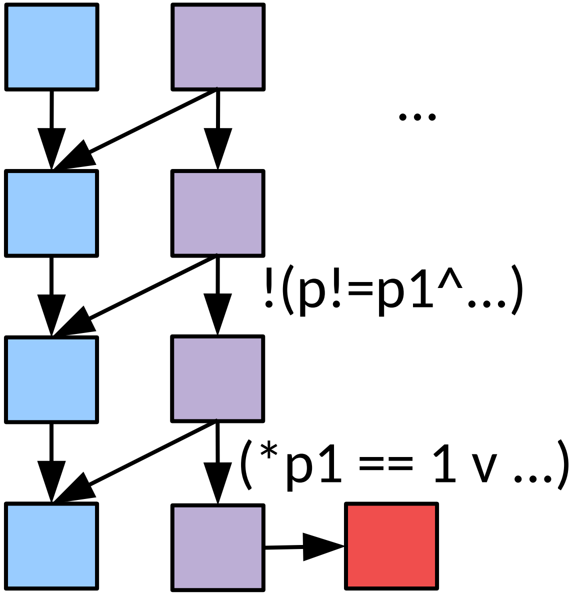


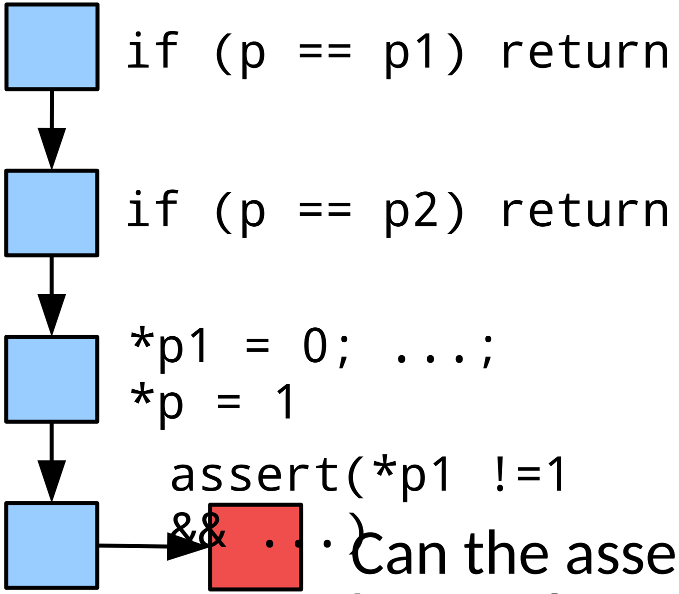
Can the assertion here fail?

[Beckman, TSE 2016]

Revisit: Test Driven Model Checking

- Carefully choosing which questions to ask can allow us to prove properties of programs!



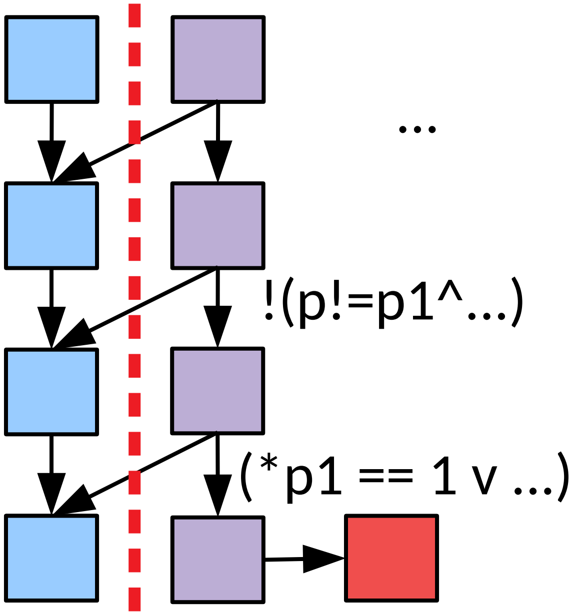


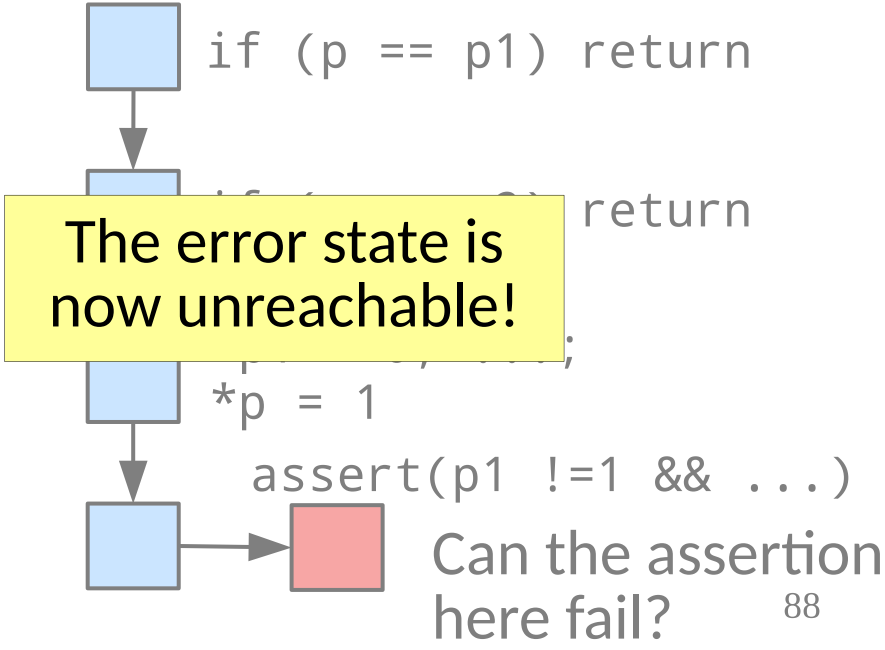
Can the assertion here fail?

[Beckman, TSE 2016]

Revisit: Test Driven Model Checking

- Carefully choosing which questions to ask can allow us to prove properties of programs!





[Beckman, TSE 2016]

Symbolic Execution

- Increasingly scalable every year

Symbolic Execution

- Increasingly scalable every year
- Can automatically generate test inputs from constraints

Symbolic Execution

- Increasingly scalable every year
- Can automatically generate test inputs from constraints
- The resulting symbolic formulae have many uses beyond just testing.

Symbolic Execution

- Increasingly scalable every year
- Can automatically generate test inputs from constraints
- The resulting symbolic formulae have many uses beyond just testing.

Try it out:
1) https://github.com/klee/klee
2) Symbolic PathFinder
3) http://research.microsoft.com/Pex/
4) http://angr.io/