

18732: Secure Software Systems

Software Model Checking for Security II

Anupam Datta

CMU
Fall 2010

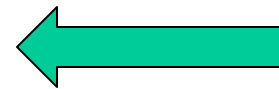
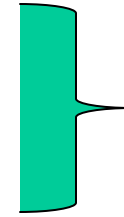
Announcement

- In class test: Monday, Sept 27
 - 10:30AM-12 noon
 - 1 letter size page filled with notes is permitted
 - no other source of information can be accessed: books, notes, laptops, smartphones....

Outline

- Security Protocols
- Overview of Model Checking
- Model Checking Code
- Model Checking Security Protocol Code
- Software Model Checking for Specific Security Vulnerabilities

More in Project 2



Verifying Security of OpenSSL

Client Code

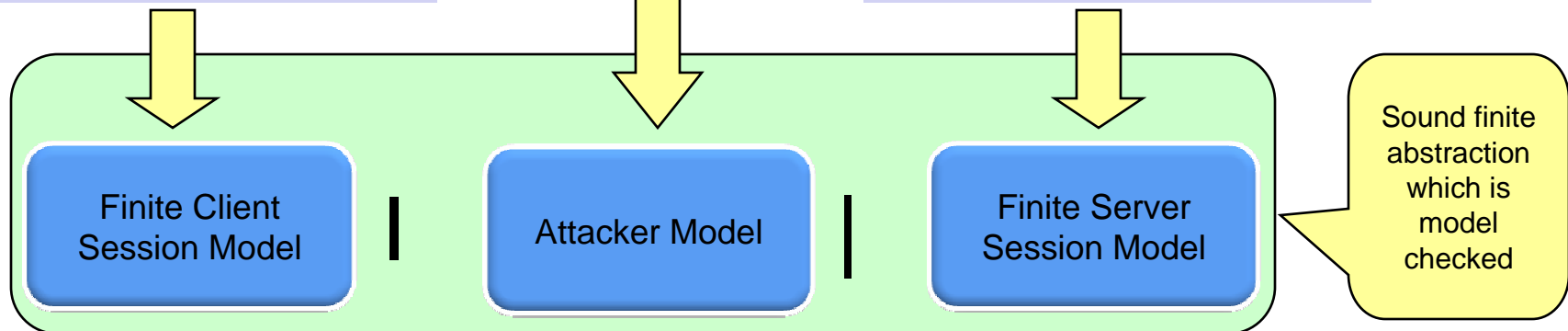
```
int state = 0;
while(1) {
  if(state==0) {
    send_hello();
    state++;
  } else if(state==1) {
    ver = recv_hello();
    state++;
  } else ...
}
```

Server Code

```
int state = 0;
while(1) {
  if(state==0) {
    recv_hello();
    state++;
  } else if(state==1) {
    send_hello(ver);
    state++;
  } else ...
}
```



Infinite state system

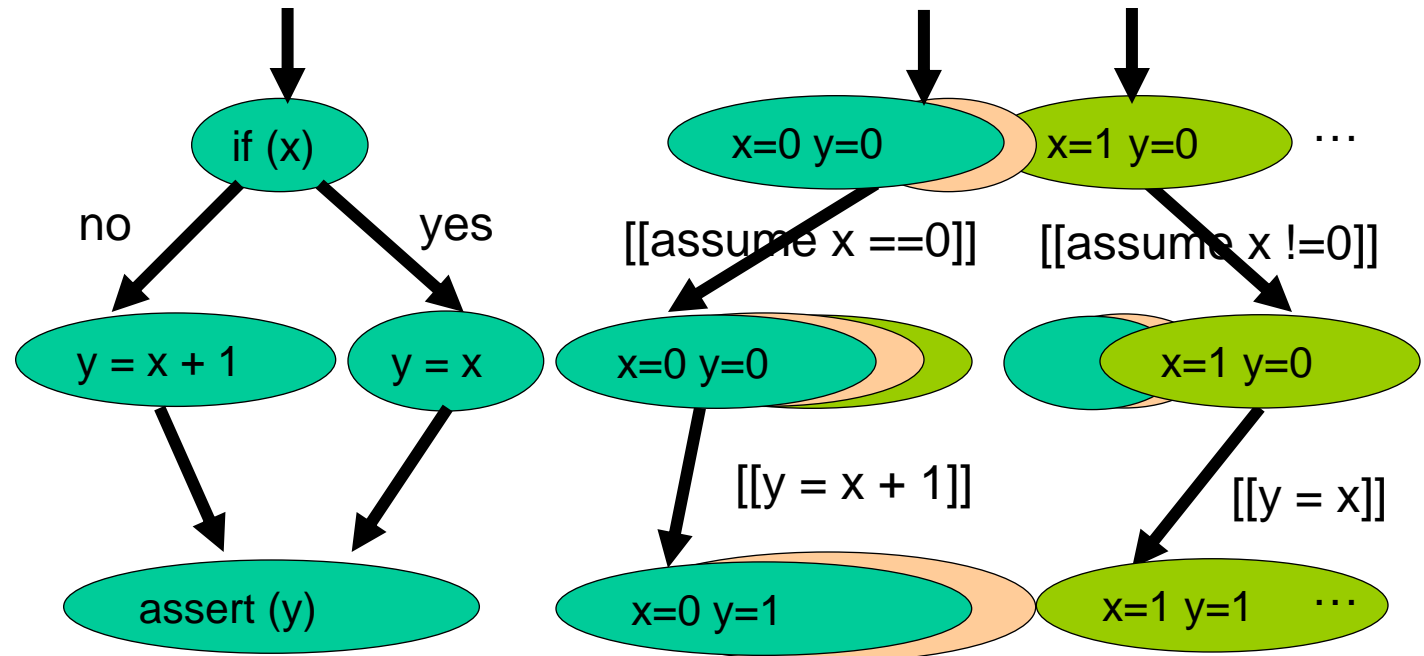


Models of C Code

```

if (x){
y = x;}
else {
y = x + 1;}
assert (y);

```



Program: Syntax

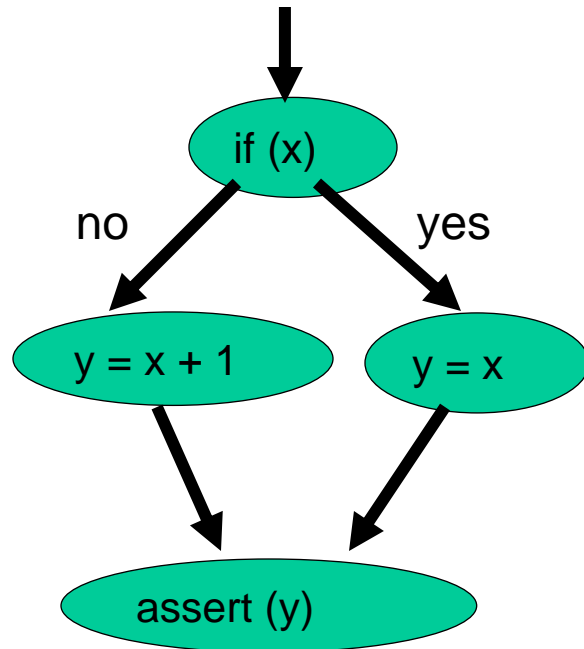
Control Flow Graph

Model: States and transitions

Goal: Extract *finite* state model *automatically* from C code

Infinite State

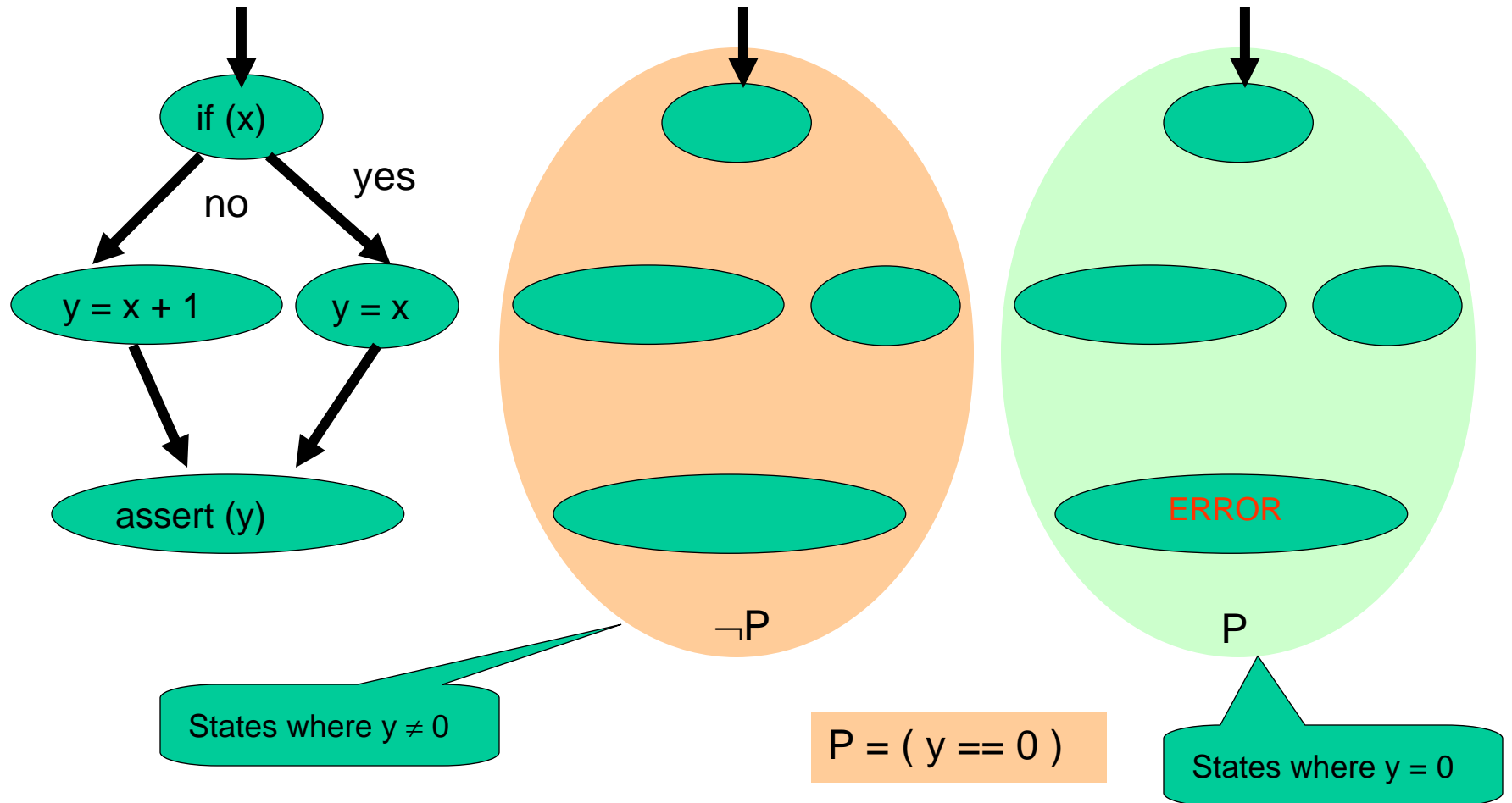
Predicate Abstraction



Extract finite model:
Partition the state space
based on values of a
finite set of **predicates**
on program variables

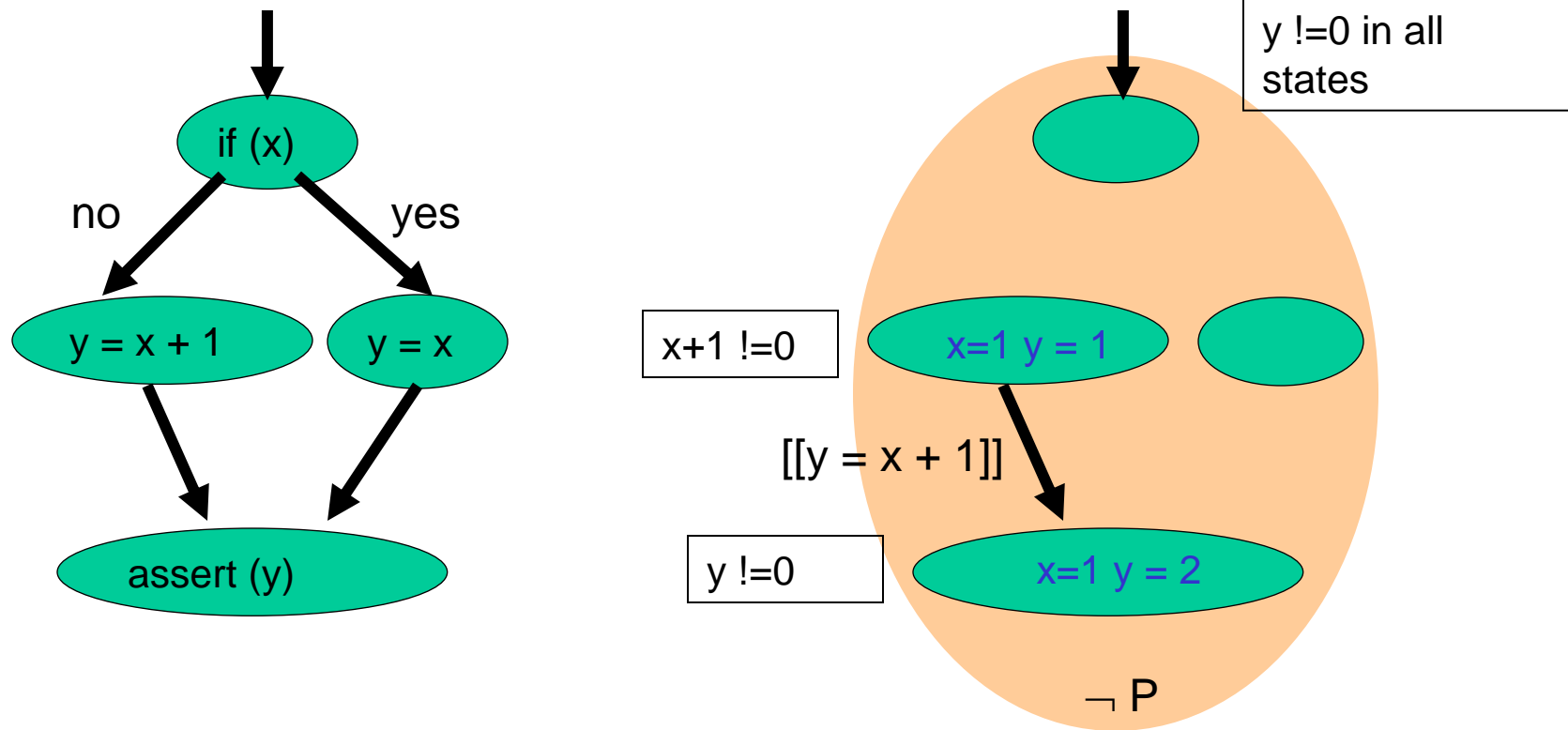
Typically, predicates will involve variables that appear in the property and control flow conditions

Predicate Abstraction: States



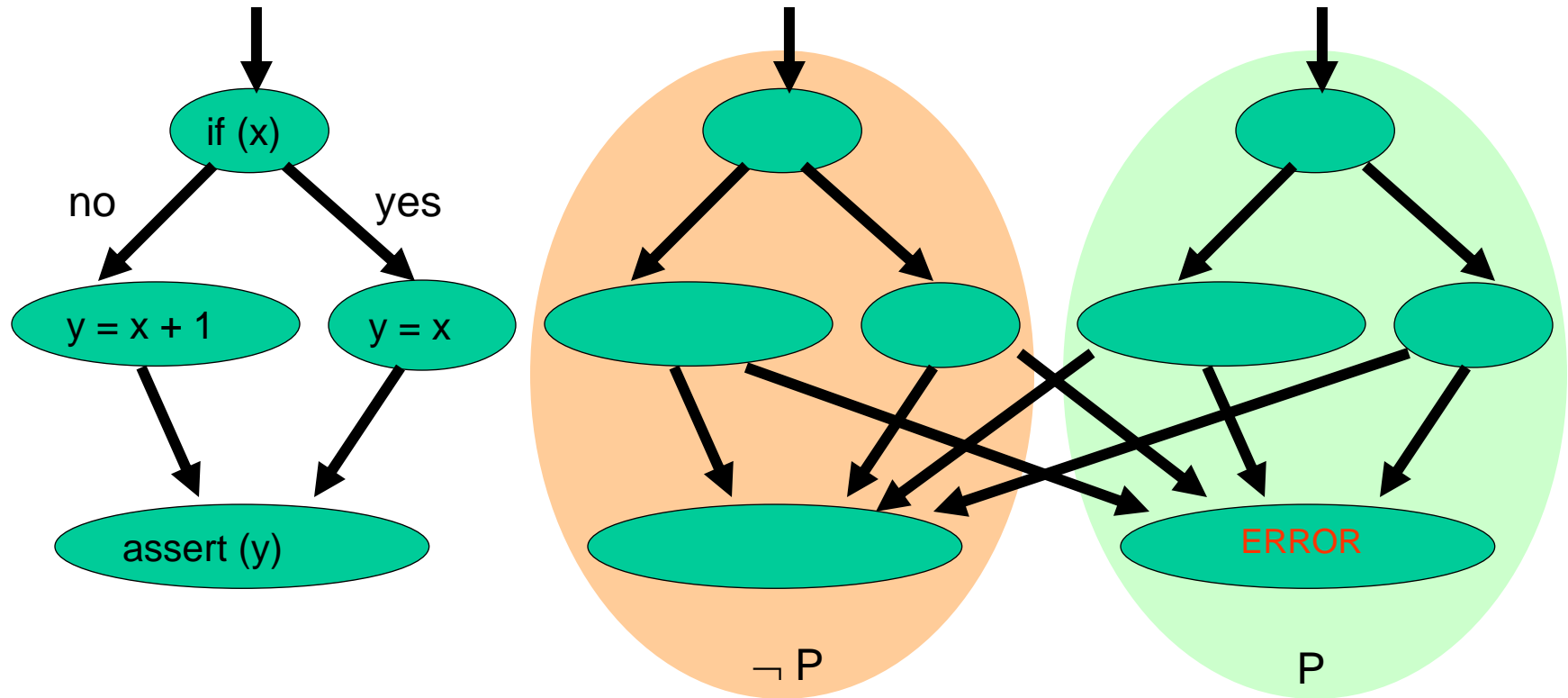
States: One copy of CFG per valuation of predicates

Predicate Abstraction: Transitions



Transition exists because the following weakest precondition is satisfiable:
 $(x + 1 \neq 0)$ and $(y \neq 0)$

Verification via Model Checking

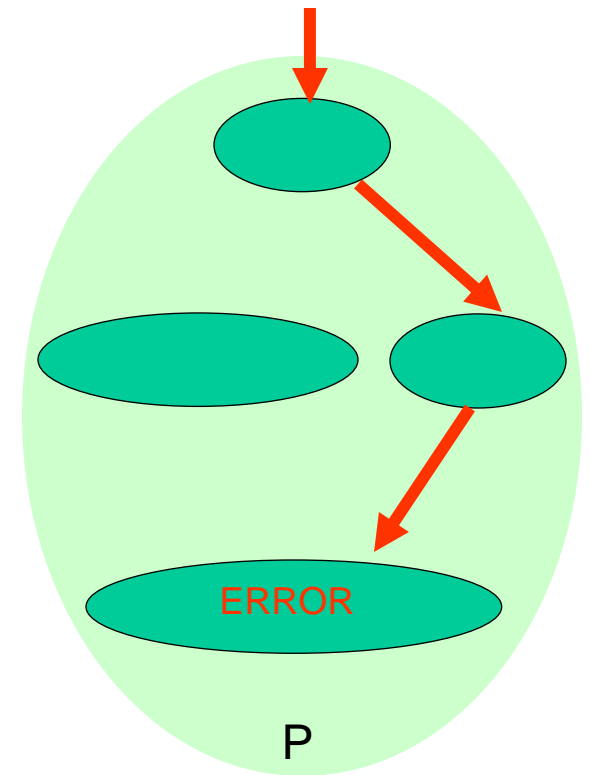
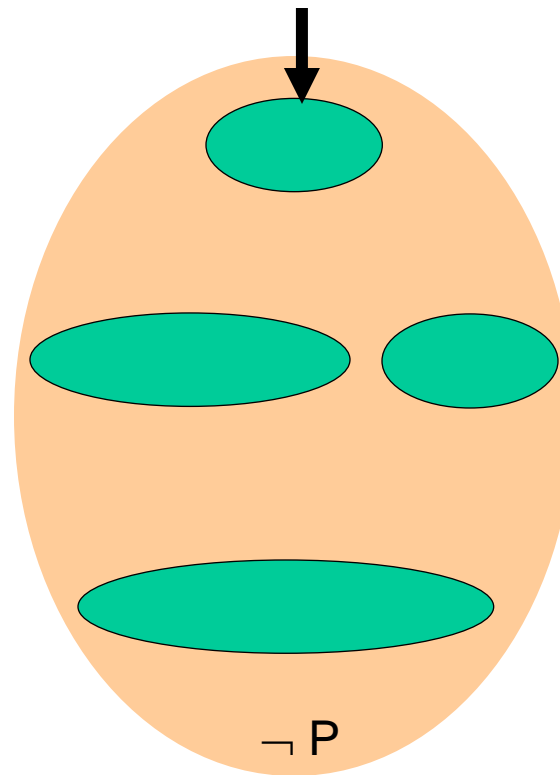
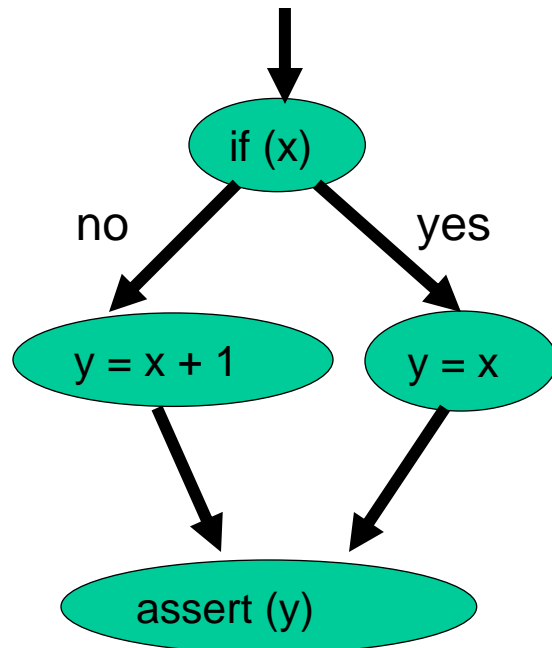


$$\phi = G (\neg \text{ERROR})$$

$$P = (y == 0)$$

The model is now finite

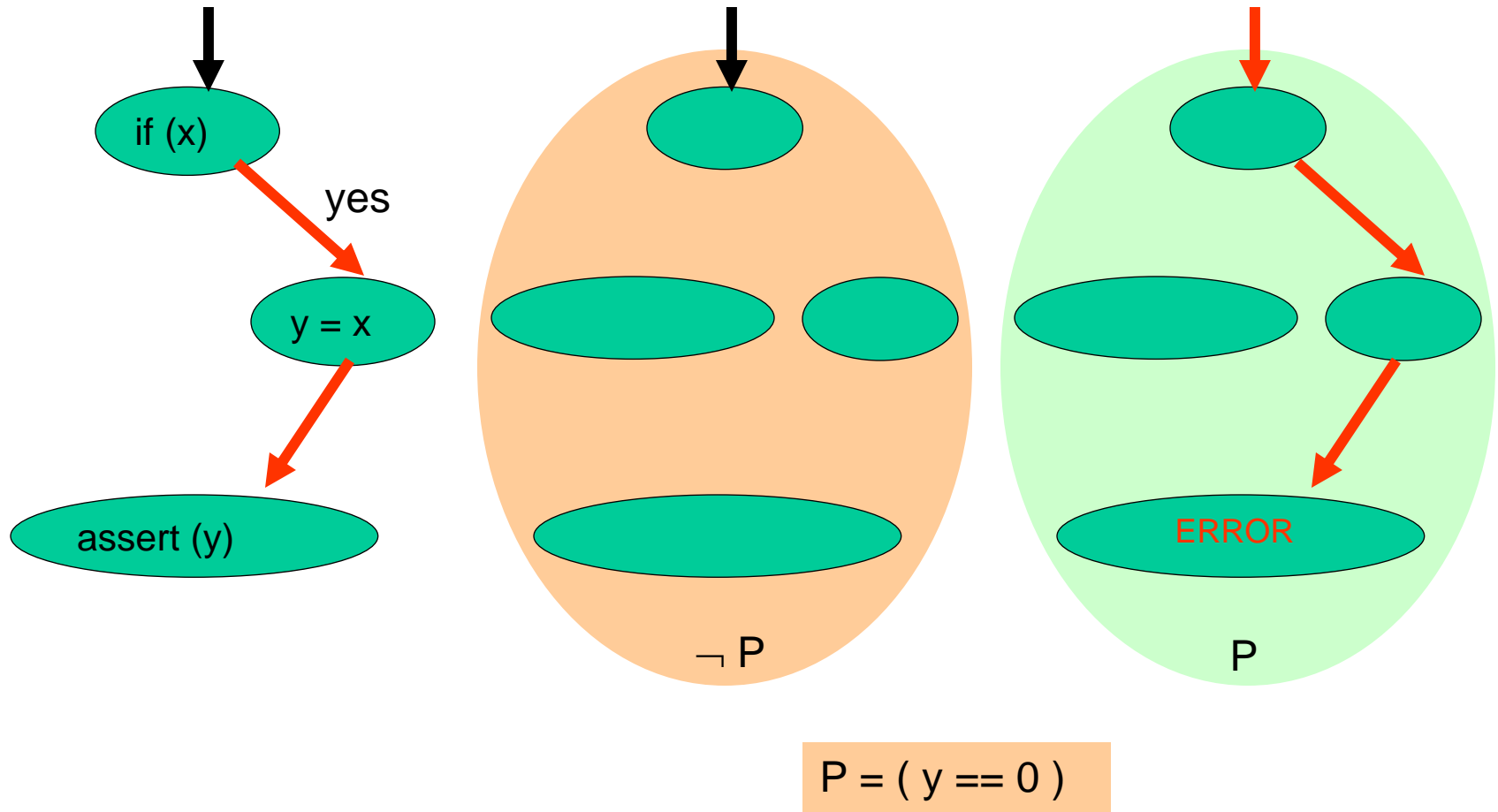
Model Checking



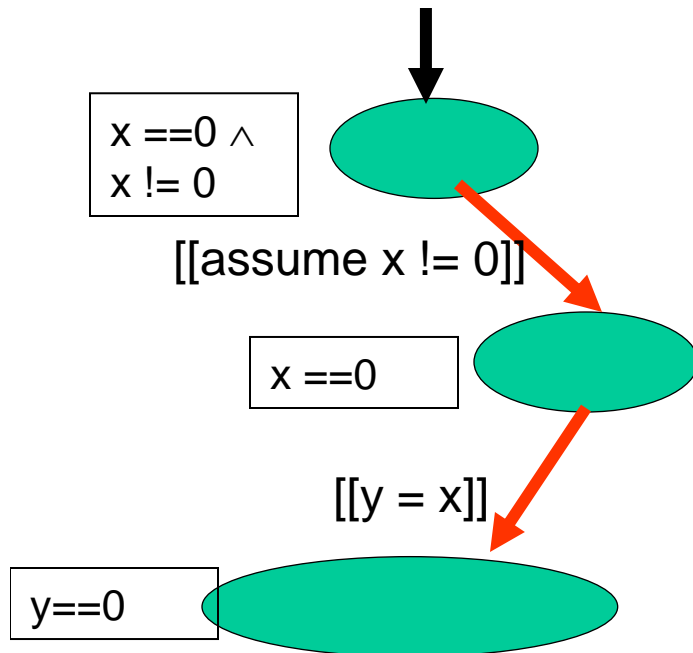
$$\phi = G (\neg \text{ERROR})$$

$$P = (y == 0)$$

Model Checking



Counterexample Validation & Refinement



Validation

- Simulate counterexample symbolically
- Call a theorem prover to determine if the precondition is satisfiable

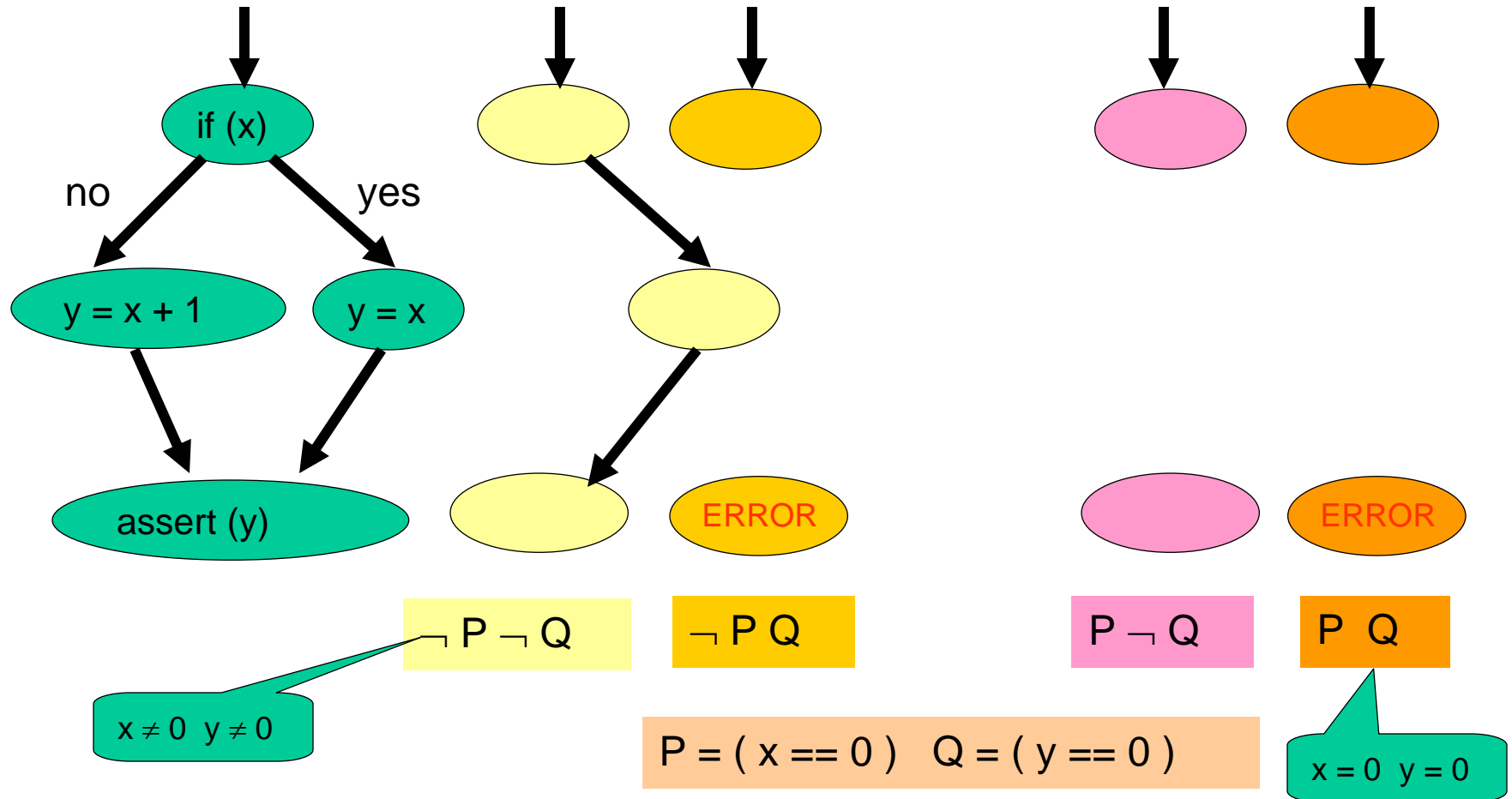
Refinement

- New set of predicates $\{x == 0, y == 0\}$

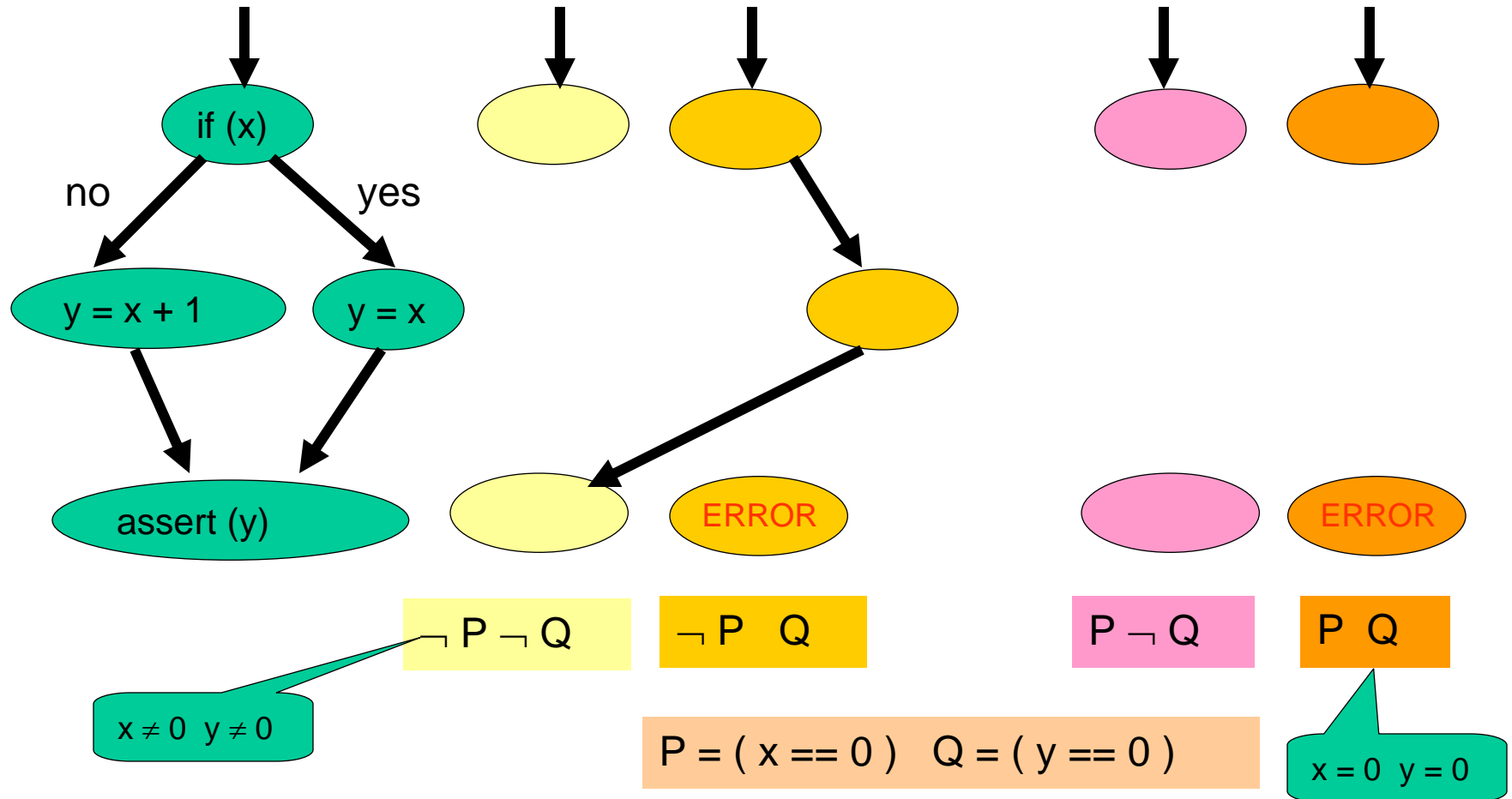
Weakest precondition by example

Spurious counterexample because $x == 0$ from assignment and $x \neq 0$ from conditional, i.e. unsatisfiable precondition

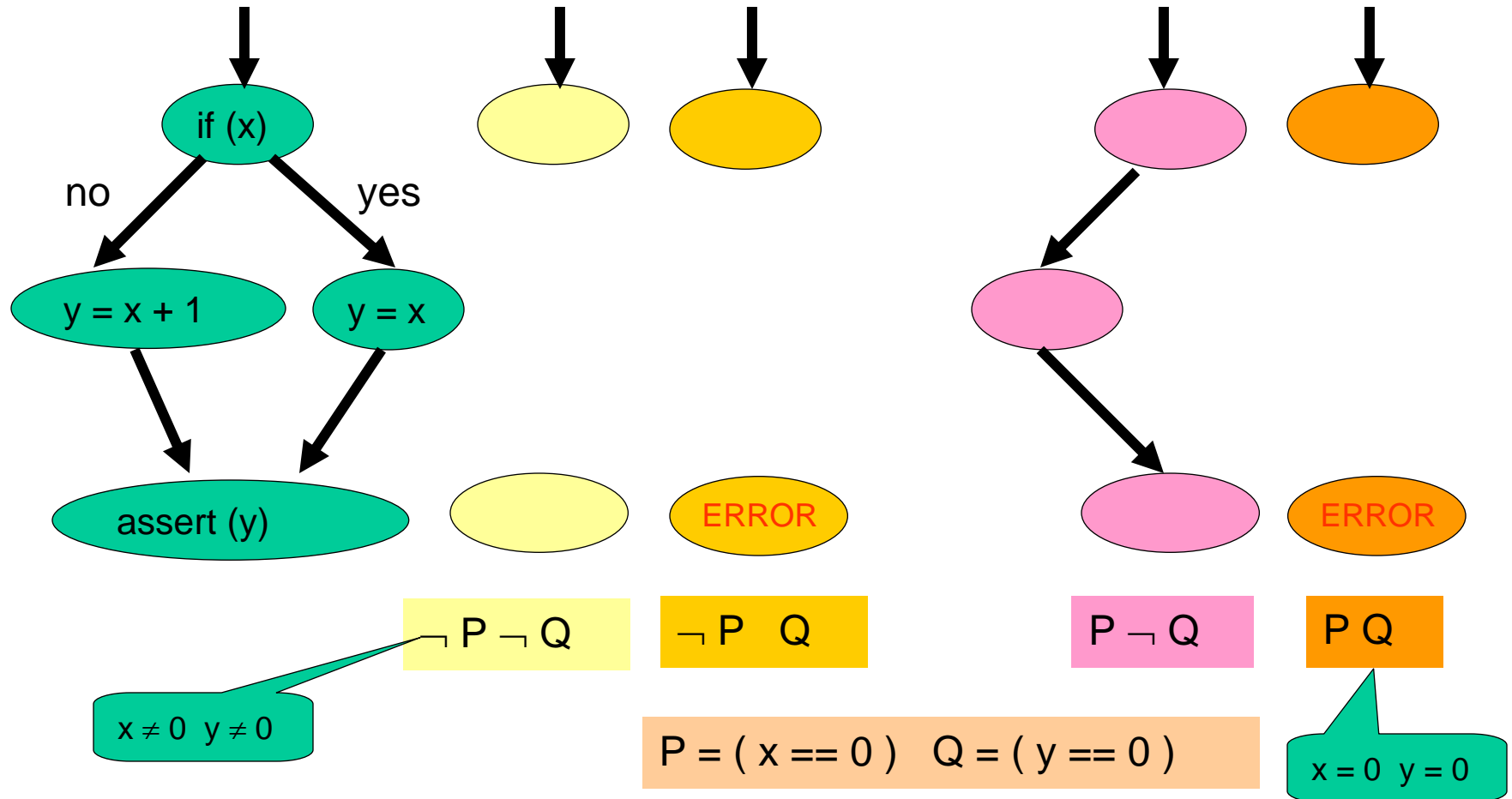
Predicate Abstraction: 2nd Iteration



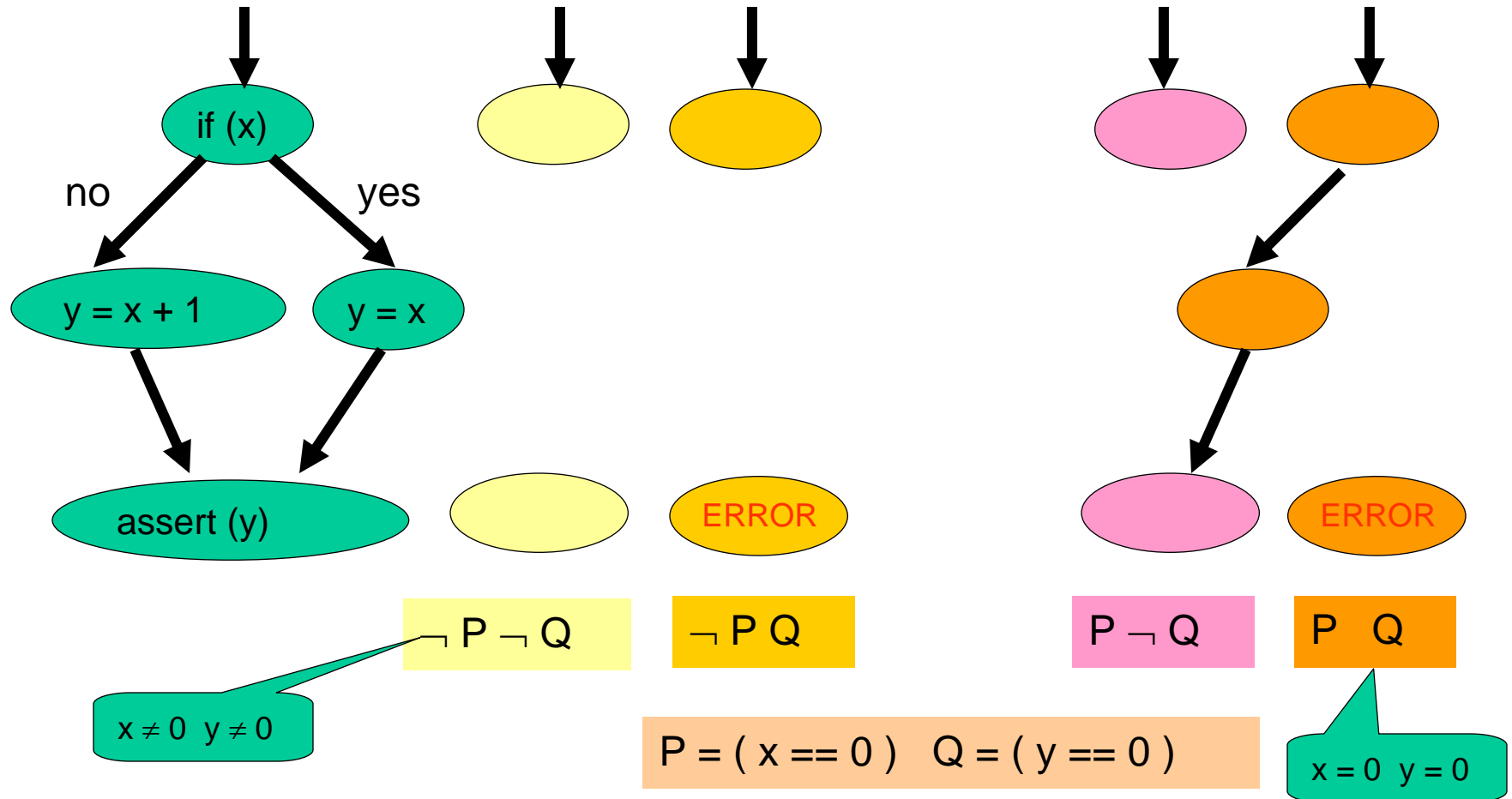
Predicate Abstraction: 2nd Iteration



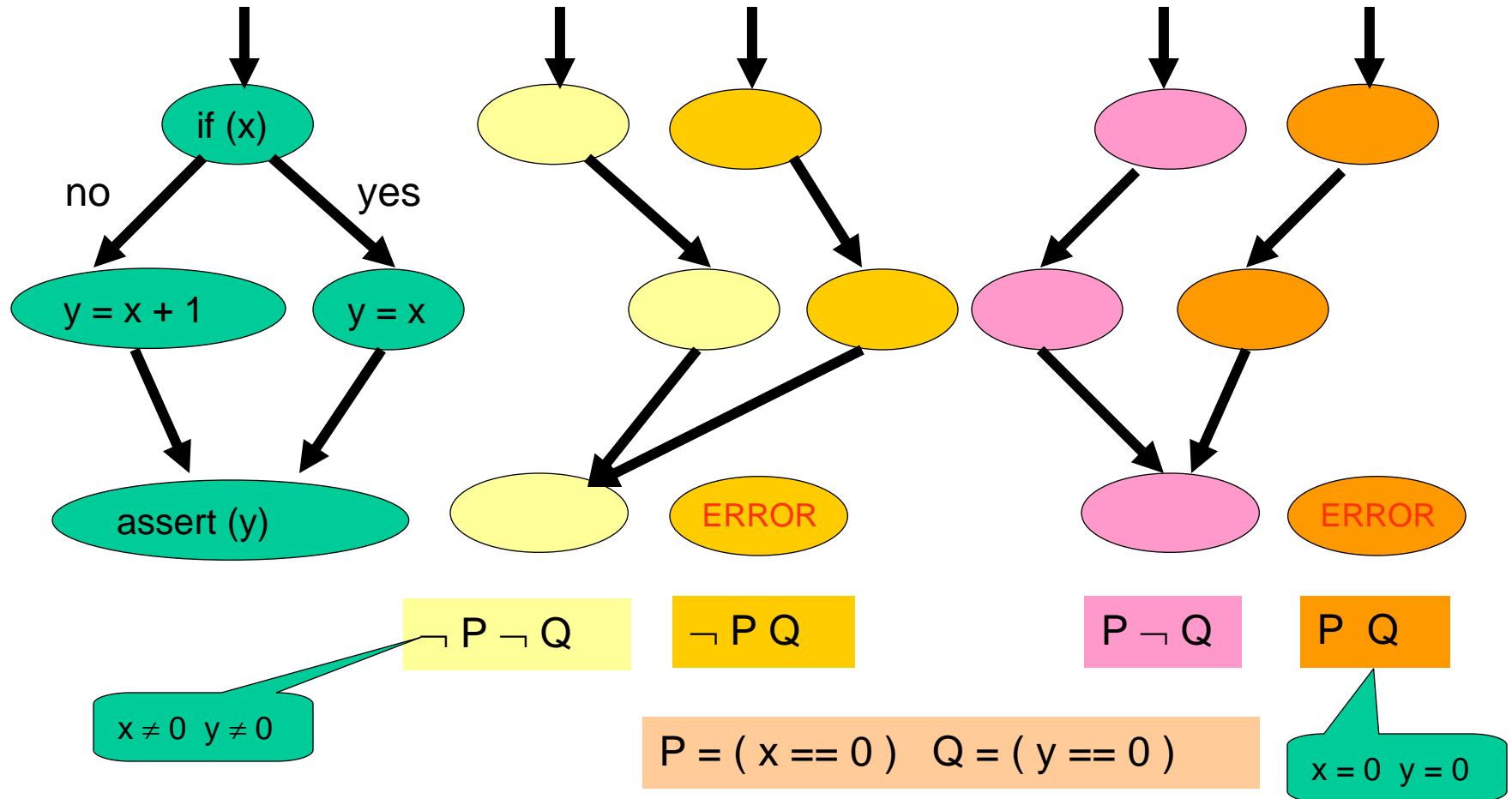
Predicate Abstraction: 2nd Iteration



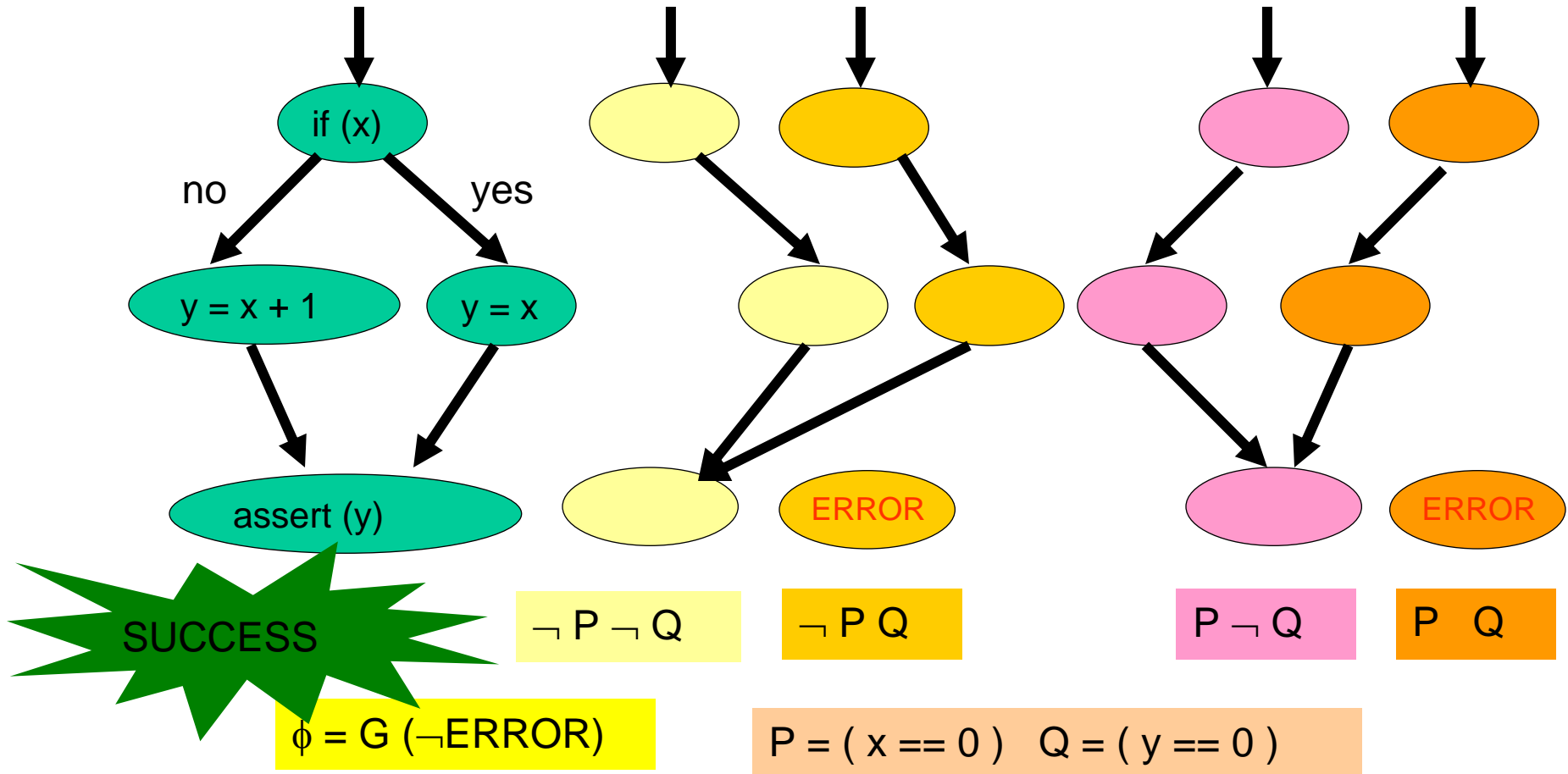
Predicate Abstraction: 2nd Iteration



Predicate Abstraction: 2nd Iteration



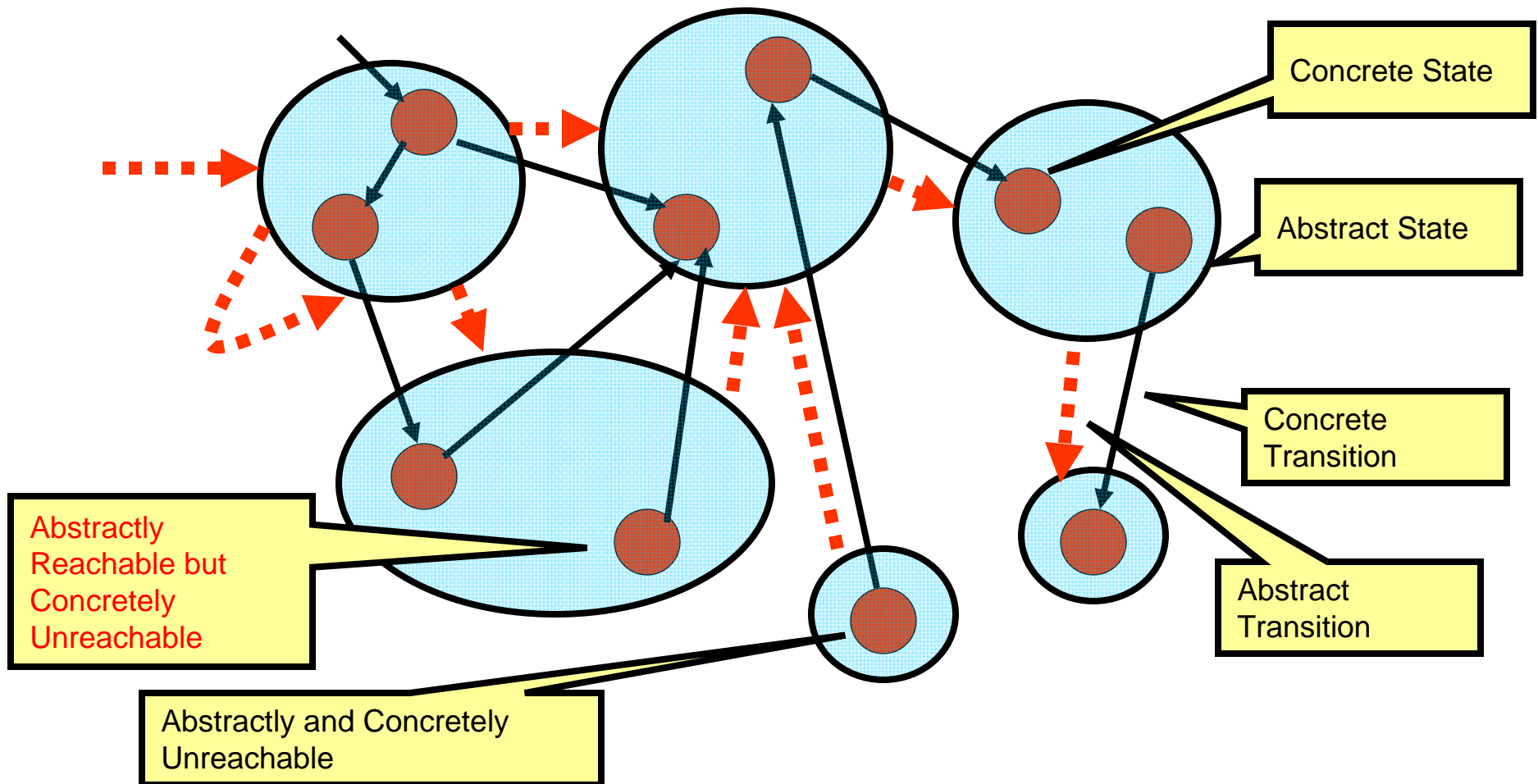
Model Checking: 2nd Iteration



Iterative Refinement: Summary

- Choose an initial set of predicates, and proceed iteratively as follows:
 1. **Abstraction:** Construct an abstract model M of the program using predicate abstraction
 2. **Verification:** Model check M . If model checking succeeds, exit with **success**. Otherwise, get counterexample CE .
 3. **Validation:** Check CE for validity. If CE is valid, exit with **failure**.
 4. **Refinement:** Otherwise, update the set of predicates and repeat from Step 1.

Soundness of Abstraction



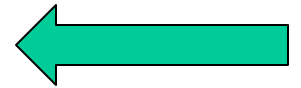
Soundness: If property holds on abstract finite model, then it also holds in the concrete infinite model

Software Model Checking Tools

- Iterative Refinement
 - SLAM, BLAST, MAGIC, Copper, ...
 - SLAM successfully applied to device driver verification at Microsoft
- Bounded Model Checking
 - CBMC, ...
- Others
 - Engines: MOPED, BEBOP, BOPPO, ...
 - Java: Java PathFinder, Bandera, BOGOR, ...
 - C: CMC, ...

Outline

- Security Protocols
- Overview of Model Checking
- Model Checking Code
- Model Checking Security Protocol Code
- Software Model Checking for Specific Security Vulnerabilities



Verifying Security of OpenSSL

Client Code

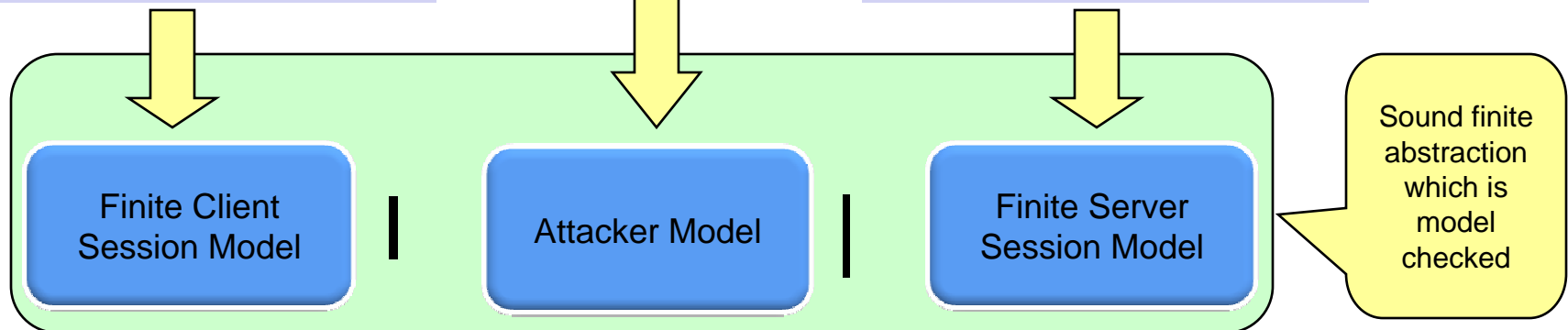
```
int state = 0;
while(1) {
  if(state==0) {
    send_hello();
    state++;
  } else if(state==1) {
    ver = recv_hello();
    state++;
  } else ...
}
```

Server Code

```
int state = 0;
while(1) {
  if(state==0) {
    recv_hello();
    state++;
  } else if(state==1) {
    send_hello(ver);
    state++;
  } else ...
}
```



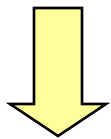
Infinite state system



Structure of Protocol Implementations

Client Code

```
int state = 0;
while(1) {
  if(state==0) {
    send_hello();
    state++;
  } else if(state==1) {
    ver = rcv_hello();
    state++;
  } else ...
}
```



Finite Abstract
Client Session
Model

How do we extract a finite model from such an infinite state system?

Observations:

- Two disjoint types of operations: numeric and cryptographic
- Control flow depends on predicates on numeric variables, e.g., state
- Cryptographic operations are implemented by library routines, e.g., send_hello()

ASPIER's model extraction:

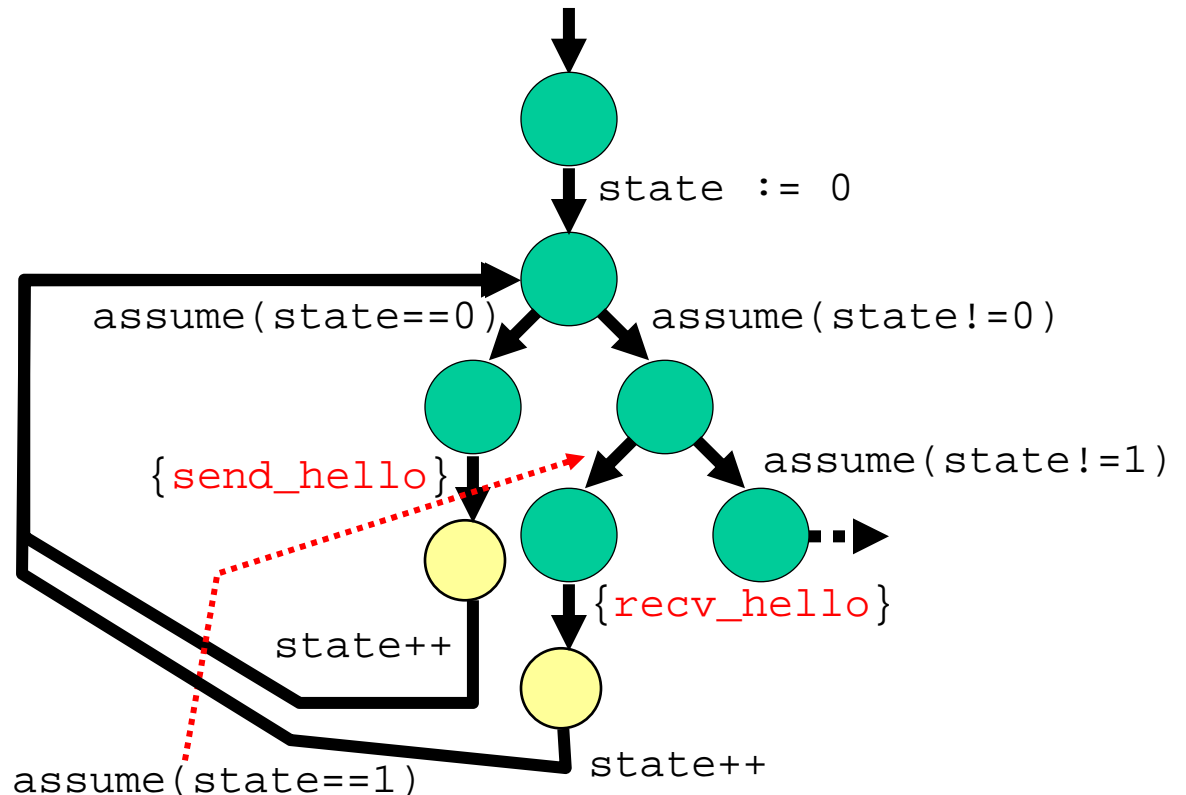
- Predicate abstraction over numeric variables assuming cryptographic operations do not update numeric variables
- Cryptographic operations in program determines messages available to adversary

ASPIER Model of OpenSSL

```

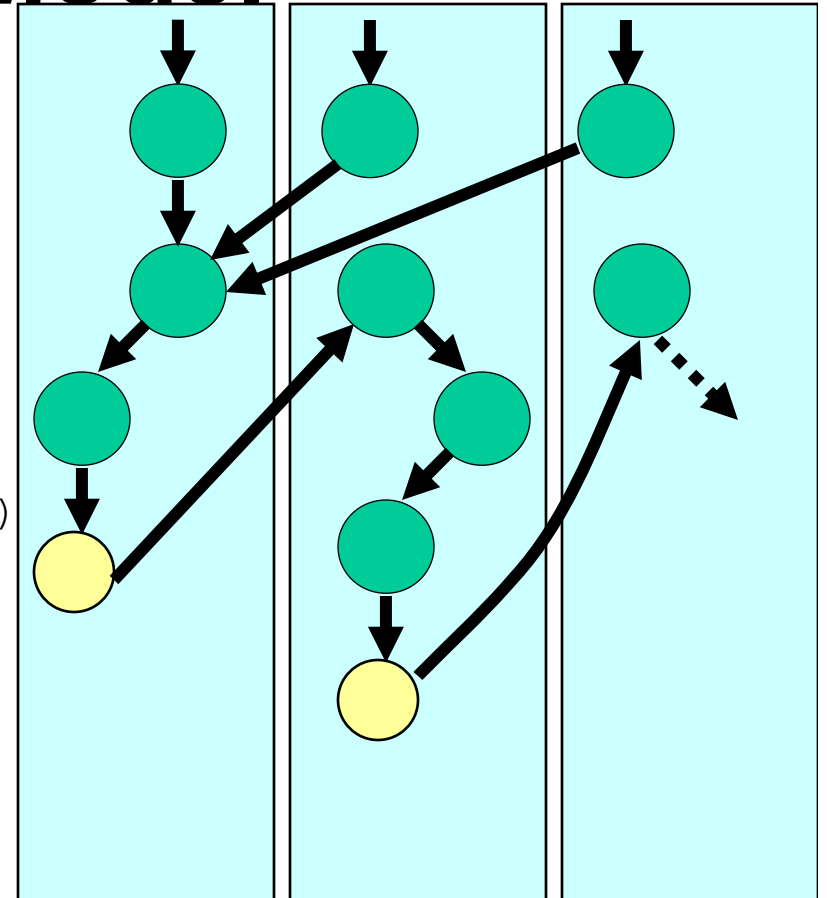
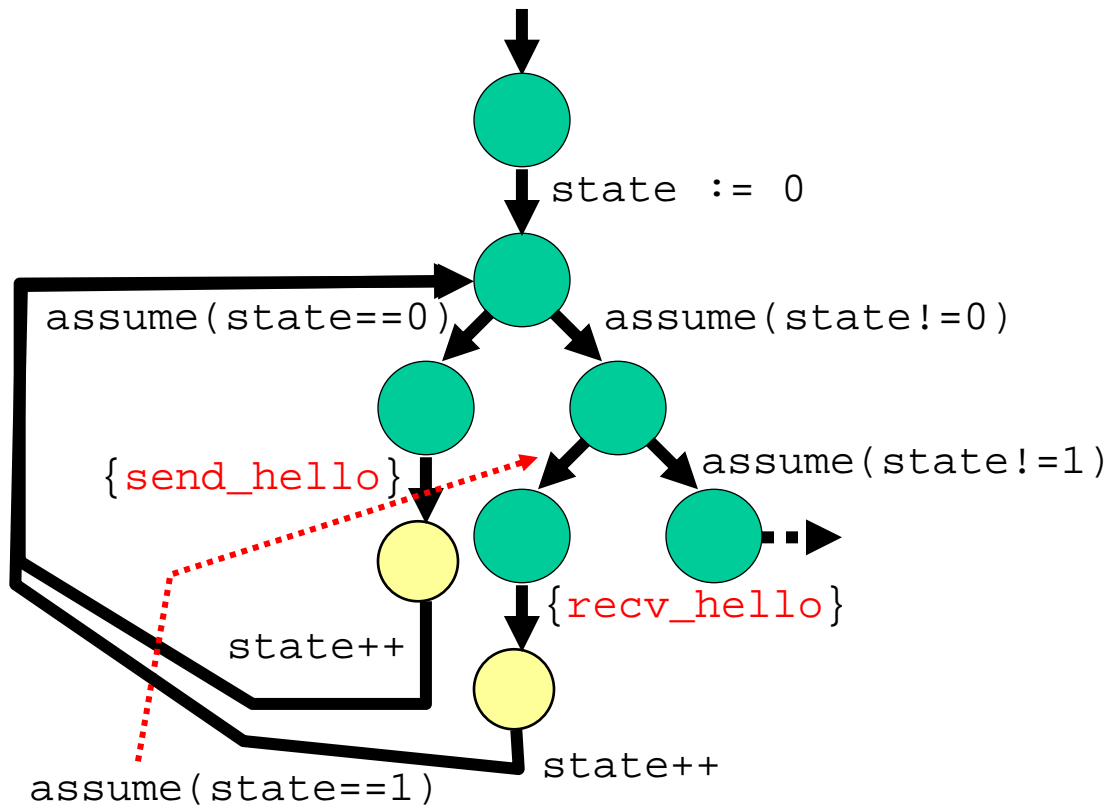
int state = 0;
while(1) {
  if(state==0) {
    send_hello();
    state++;
  } else if(state==1) {
    ver = recv_hello();
    state++;
  } else ...
}

```



`send_hello` sends first msg of SSL out on network; does not update state

Predicate Abstraction over OpenSSL Model



$P = (state == 0) \quad Q = (state == 1)$

$P, \neg Q$

$\neg P, Q$

$\neg P, \neg Q$

OpenSSL Verification Results with ASPIER

- Verified OpenSSL implementation of SSL 3.0 handshake
 - Authentication and secrecy properties (up to 3 clients and 3 servers)
 - Client and server implementation each consists of about 1200 LOC
- Confirmed the presence of the version-rollback attack in OpenSSL when clients and servers implement both SSL 2.0 and SSL 3.0

Outline

- Security Protocols
- Overview of Model Checking
- Model Checking Code
- Model Checking Security Protocol Code
- Software Model Checking for Specific Security Vulnerabilities



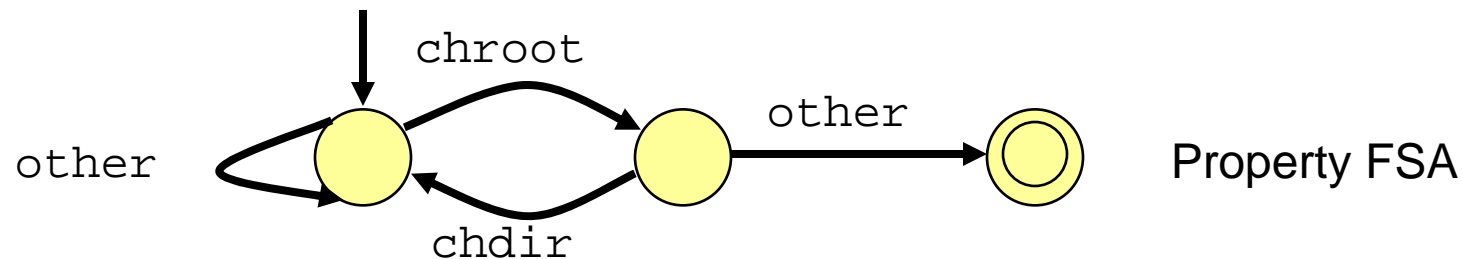
Example: chroot and chdir

```
//current directory is "/var/ftp"
chroot("/var/ftp/pub");
filename = read_from_network();
fd = open(filename, O_RDONLY);
```

Missing call to
chdir("/")

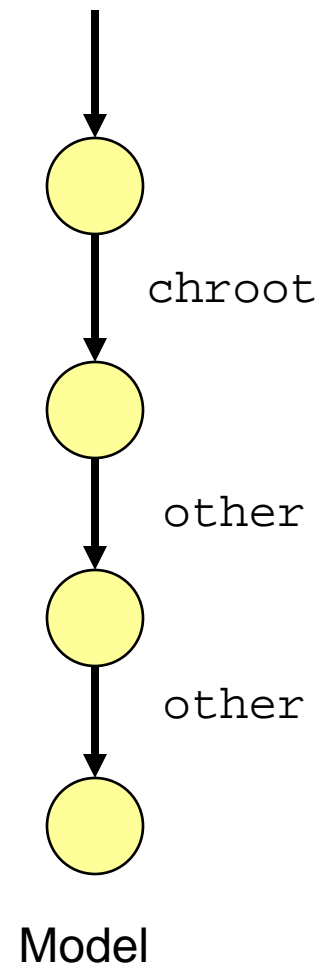
Attacker can read "/etc/passwd" by supplying
"./../etc/passwd" as "filename"

In general, we want every call to "chroot" to be followed
immediately by a call to "chdir("/")"

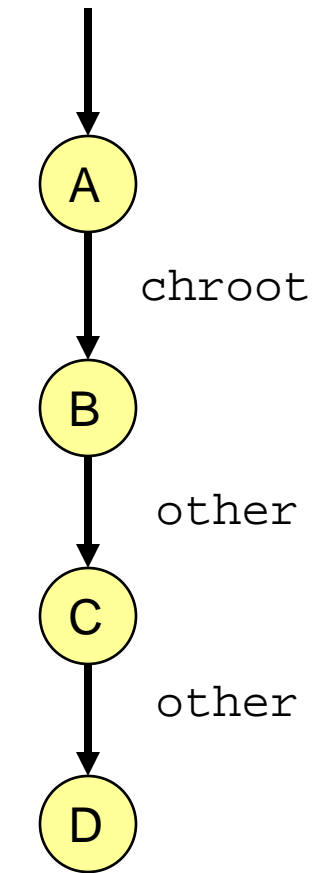


MOPS: Model extraction

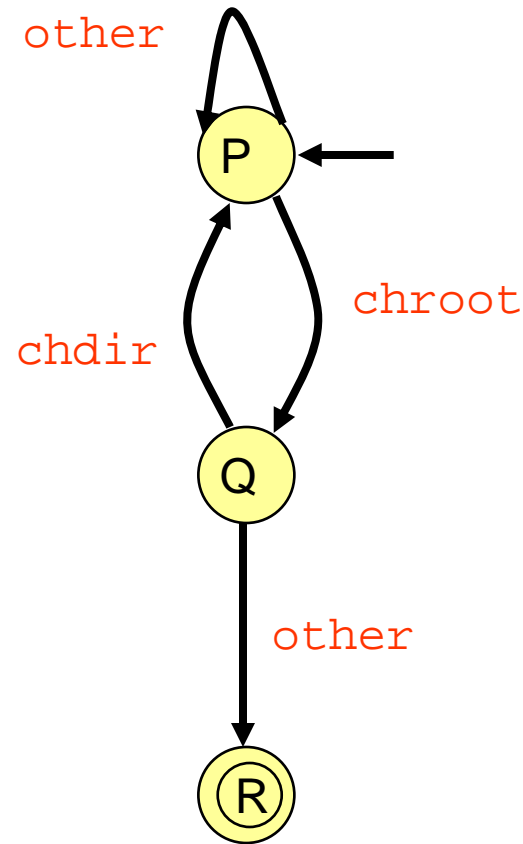
```
//current directory is "/var/ftp"  
chroot("/var/ftp/pub");  
  
filename = read_from_network();  
  
fd = open(filename,O_RDONLY);
```



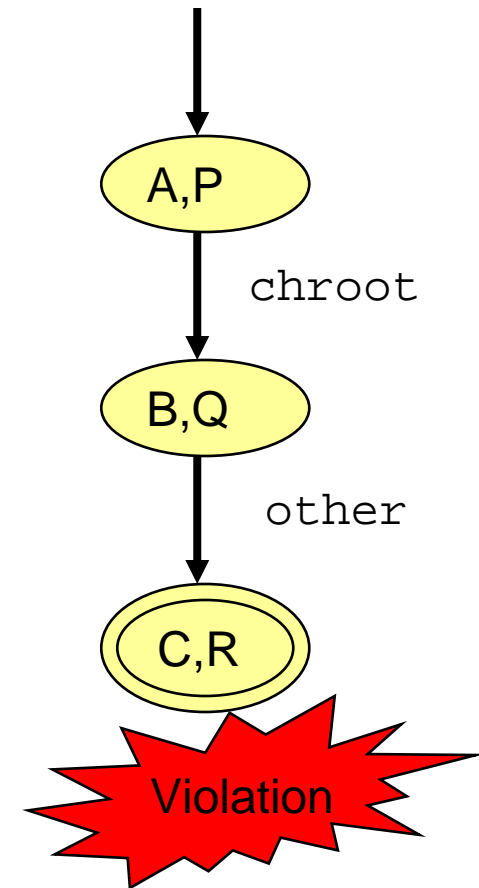
MOPS: Verification



Model



Property FSA



Model X Property FSA

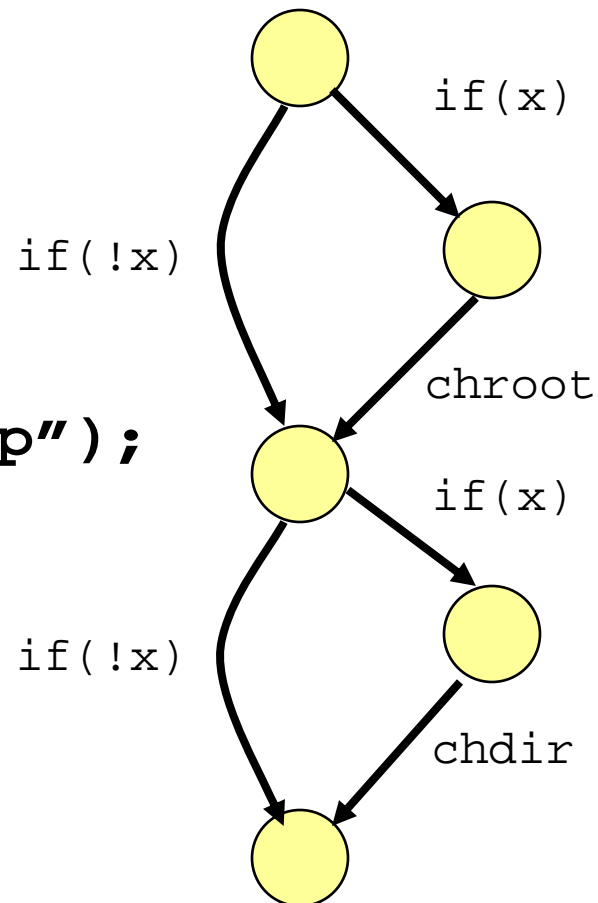
MOPS: Data Insensitive

if(x)

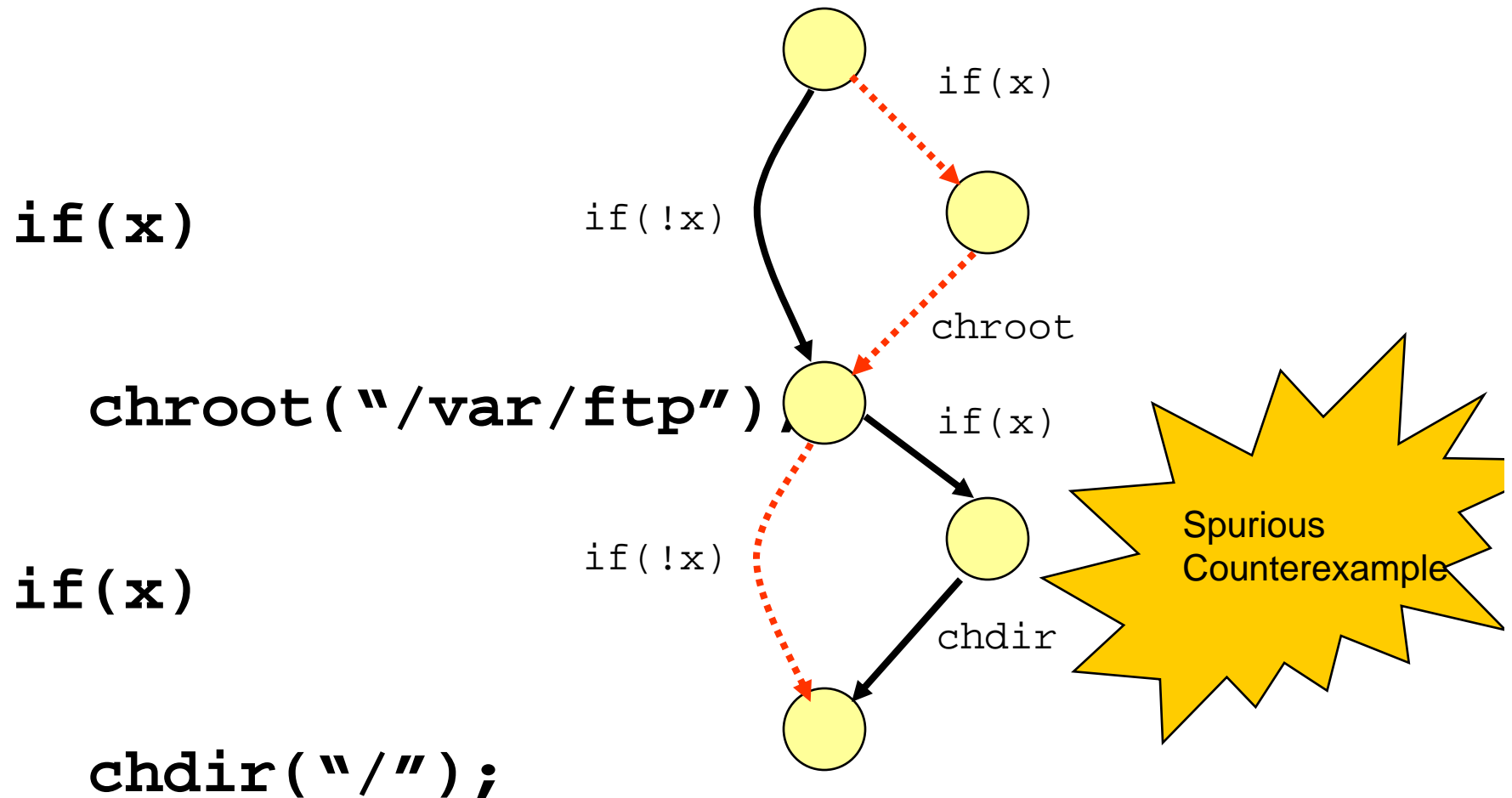
chroot("/var/ftp");

if(x)

chdir("/");



MOPS: Data Insensitive



Difference between ASPIER and MOPS

1. ASPIER aims to show absence of attacks in the presence of explicit adversary; MOPS focuses on finding specific vulnerabilities
2. ASPIER works with concurrent systems, MOPS sequential
3. MOPS scales to much larger programs because (a) it does not deal with concurrency; (b) of the abstractions it makes (e.g., data insensitive)
4. ASPIER uses iterative refinement, MOPS doesn't (implies spurious counterexamples)

Questions?

Acknowledgement

- A number of slides used in this lecture are based on slides provided by Sagar Chaki (CMU SEI)

MOPS: Overview

- Security analysis of sequential C programs
 - Temporal safety properties
- Uses program control flow graph (CFG) as model
 - Data-flow insensitive, e.g., assumes that all branches can be taken
 - No abstraction refinement
 - Can handle recursion: models extracted are push-down automata (PDA)
- Violations of properties expressed as finite state automata (FSA)
- Verifies if the CFG model has an execution that violates the property
 - Compose CFG model (PDA) with property FSA
 - Check emptiness of the resulting system (PDA)

MOPS: Handling Recursion

```
void do_foo() {  
    foo();  
}
```

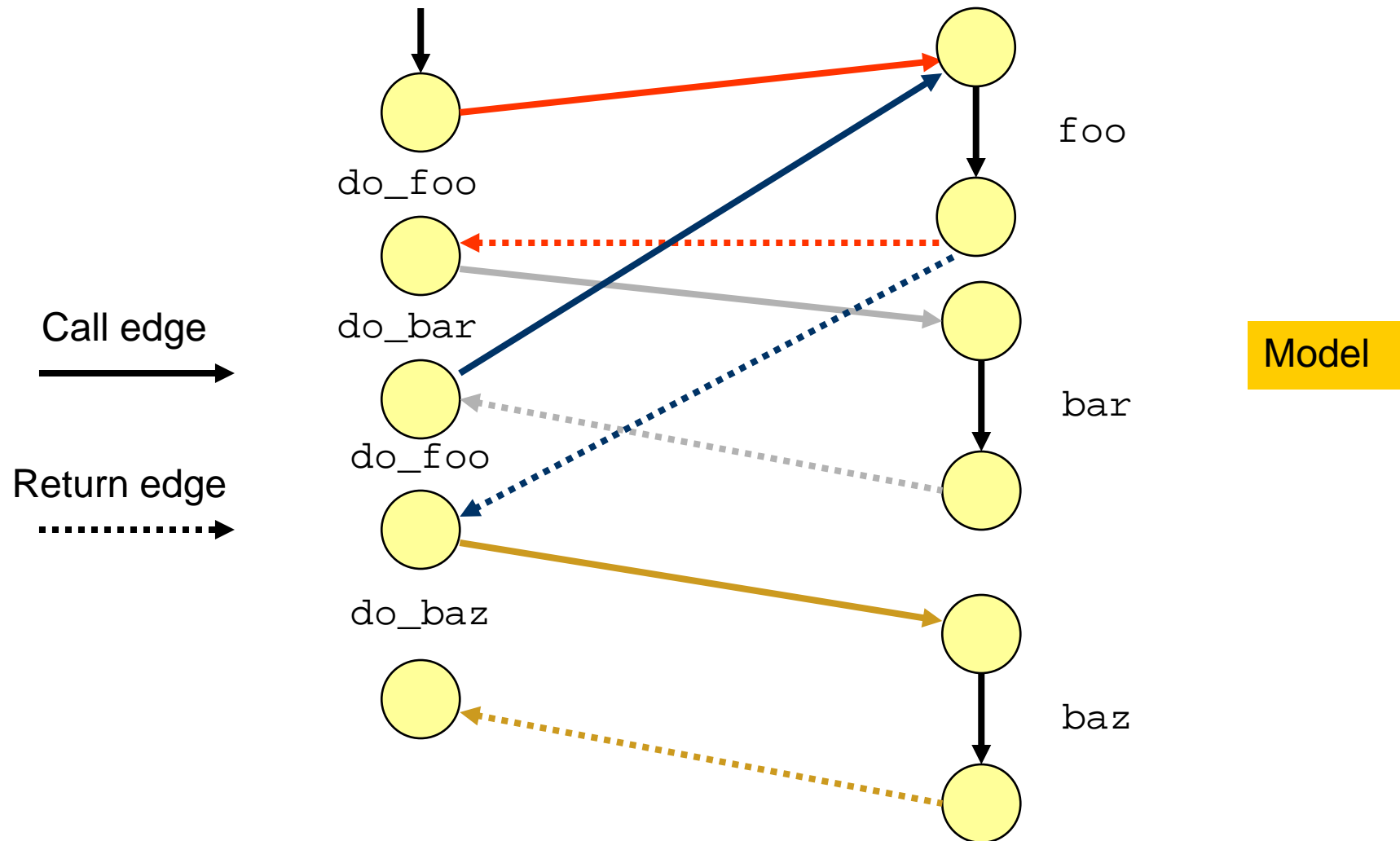
```
void do_bar() {  
    bar();  
}
```

```
void do_baz() {  
    baz();  
}
```

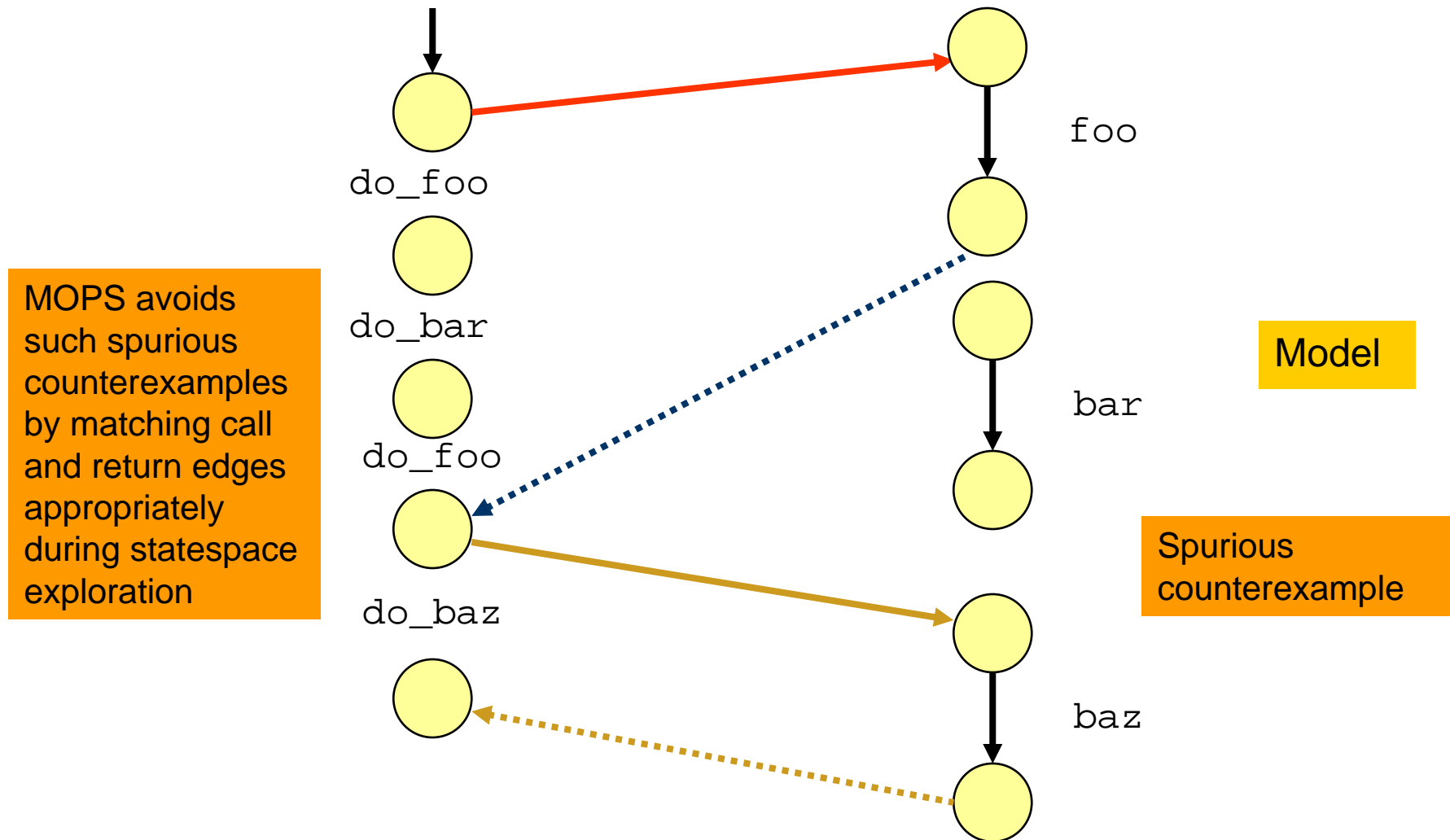
```
void main() {  
    do_foo();  
    do_bar();  
    do_foo();  
    do_baz();  
}
```

Property: Every call to `baz` is preceded by a call to `bar`

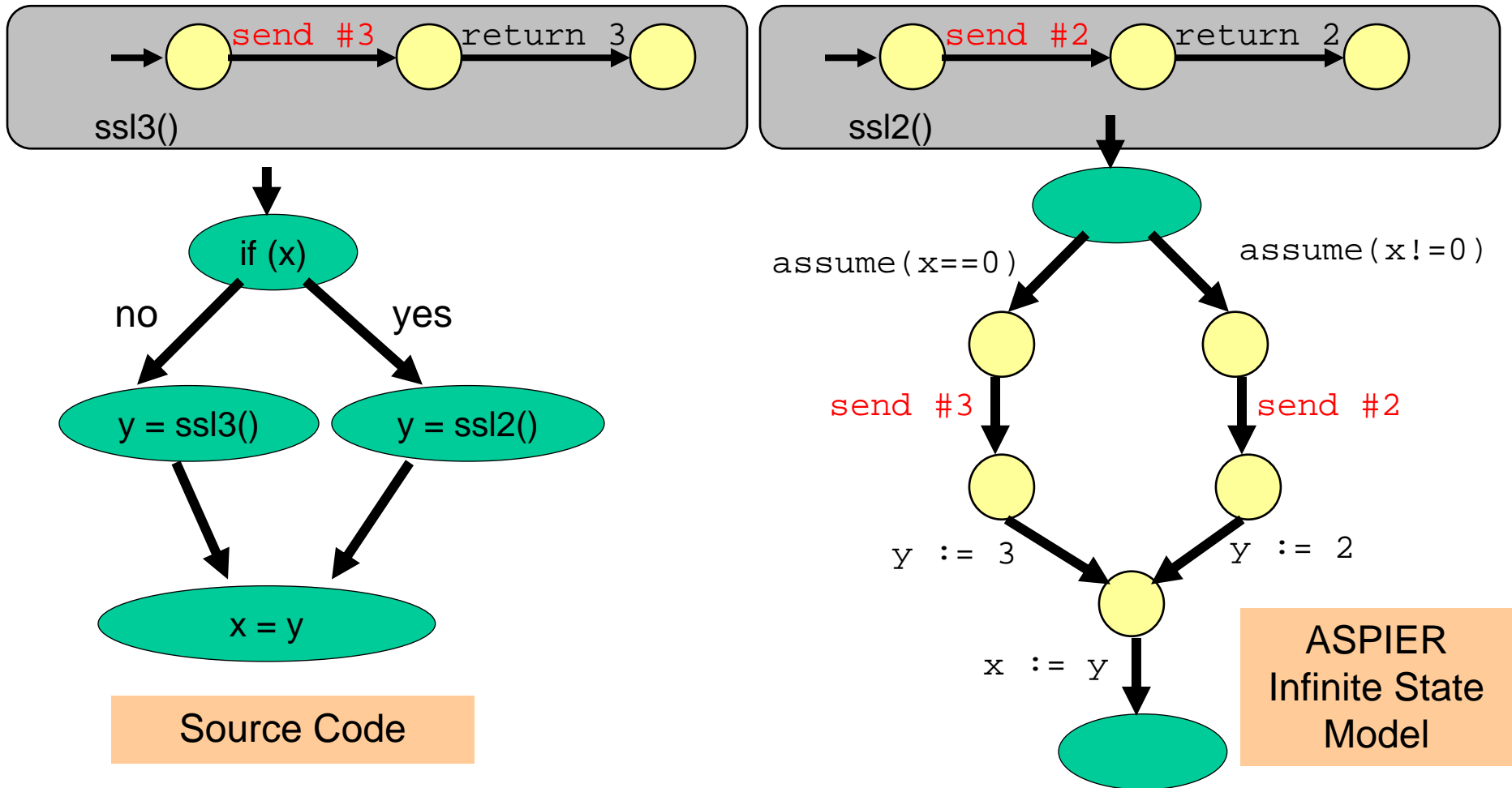
MOPS: Handling Recursion



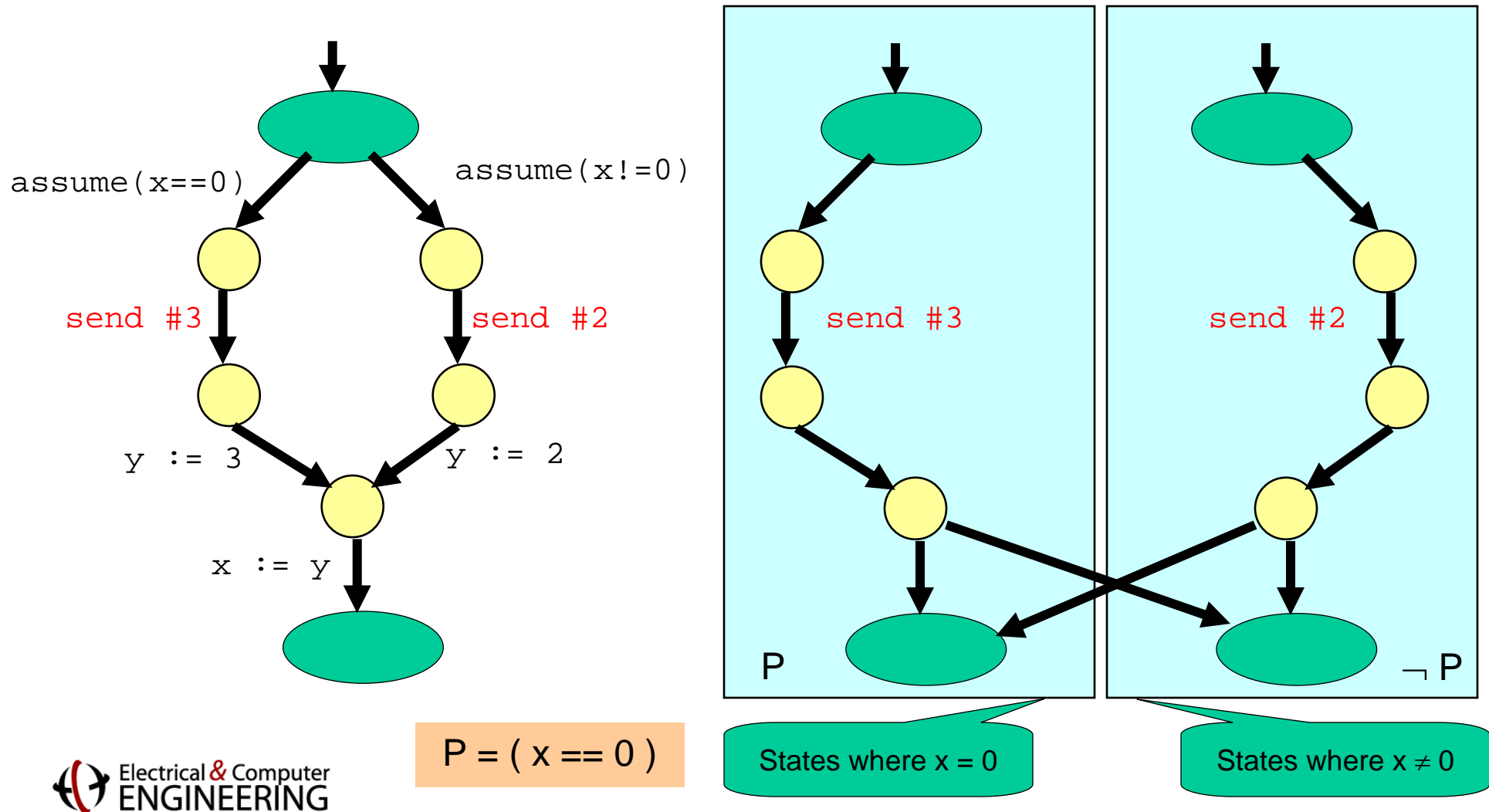
MOPS: Handling Recursion



Abstraction Example Motivated by OpenSSL23



ASPIER's Predicate Abstraction



Version Rollback Attack Results

Servers	Clients	Server-Rollback				
		Present	Inst	T1	T2	Mem
1-SSL3	1-SSL3	No	4	4.8	4.5	18.8
1-SSL3	1-SSL23	No	4	5.2	4.7	18.9
1-SSL23	1-SSL3	No	4	6.0	5.2	19.7
1-SSL23	1-SSL23	Yes	4	79.9	77.8	37.7
2-SSL3	2-SSL3	No	25	72749	35893	415
2-SSL3	2-SSL23	No	25	96740	49868	520
2-SSL23	2-SSL3	No	25	171308	86373	966
2-SSL23	2-SSL23	Yes	1	5292	2719	136
Servers	Clients	Client-Rollback				
1-SSL3	1-SSL3	No	4	5.8	5.1	18.8
1-SSL3	1-SSL23	No	4	6.2	5.3	19.2
1-SSL23	1-SSL3	No	4	7.3	6.0	19.7
1-SSL23	1-SSL23	Yes	4	79.3	78.7	34.0
2-SSL3	2-SSL3	No	25	99224	50165	437
2-SSL3	2-SSL23	No	25	134869	111530	632
2-SSL23	2-SSL3	No	25	258459	162185	1294
2-SSL23	2-SSL23	Yes	1	4249	2711	139

Results: Verification

Client#	Server#	AuthSrvr		
		Inst	Time	Mem
2	2	25	34987	410
3	3	225	226953	625

Client#	Server#	AuthClnt		
		Inst	Time	Mem
2	2	25	48539	434
3	3	225	499535	1583

Client#	Server#	Secrecy		
		Inst	Time	Mem
2	2	25	43936	424
3	3	225	399279	1132