# Software Verification using Predicate Abstraction and Iterative Refinement

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA  15213

Sagar Chaki
March 16, 2011

# Outline

Overview of Model Checking

Creating Models from C Code: Predicate Abstraction

Eliminating spurious behaviors from the model: Abstraction Refinement

Concluding remarks : research directions, tools etc.

# Model Checking

Algorithm for answering queries about behaviors of state machines

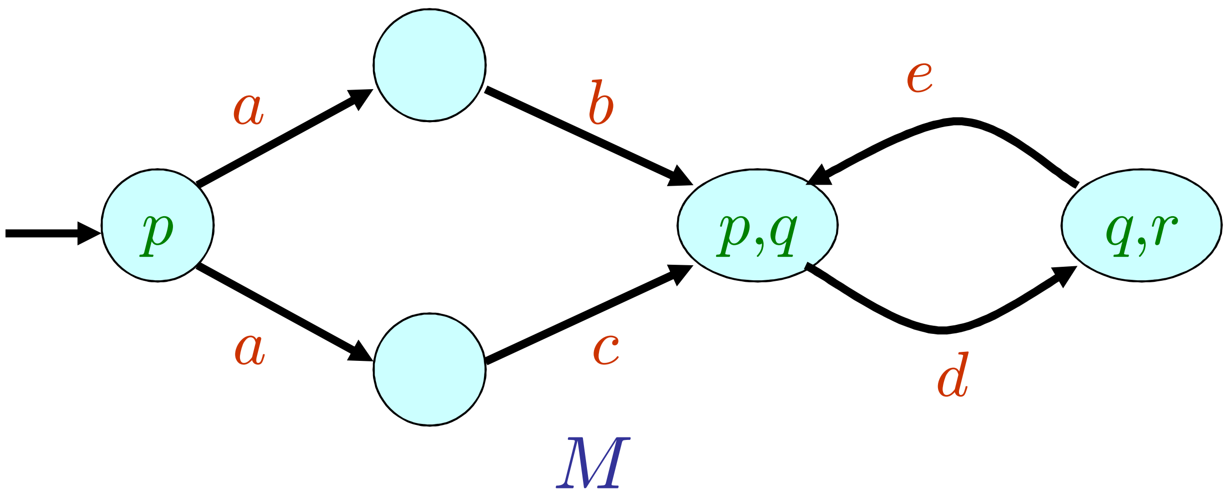
- Given a state machine $M$ and a query $\phi$ does $M \vDash \phi$ ?

Standard formulation:

- $M$ is a Kripke structure
- $\phi$ is a temporal logic formula
  - Computational Tree Logic (CTL)
  - Linear Temporal Logic (LTL)
- We'll use (slight) variants

Discovered independently by Clarke & Emerson and Queille & Sifakis in the early 1980's

# Models: Doubly Labeled State Machines



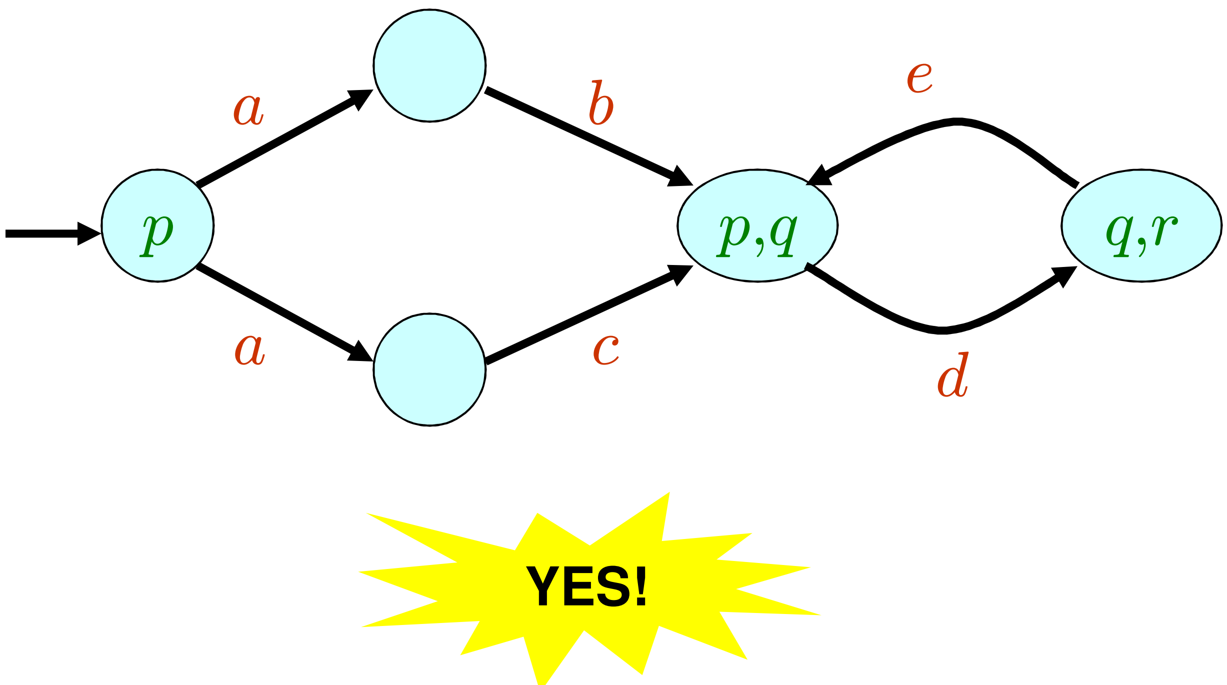

$\Sigma(M) = \{ a,b,c,d,e,f \}$ $\quad AP = \{ p,q,r \}$

alphabet

propositions

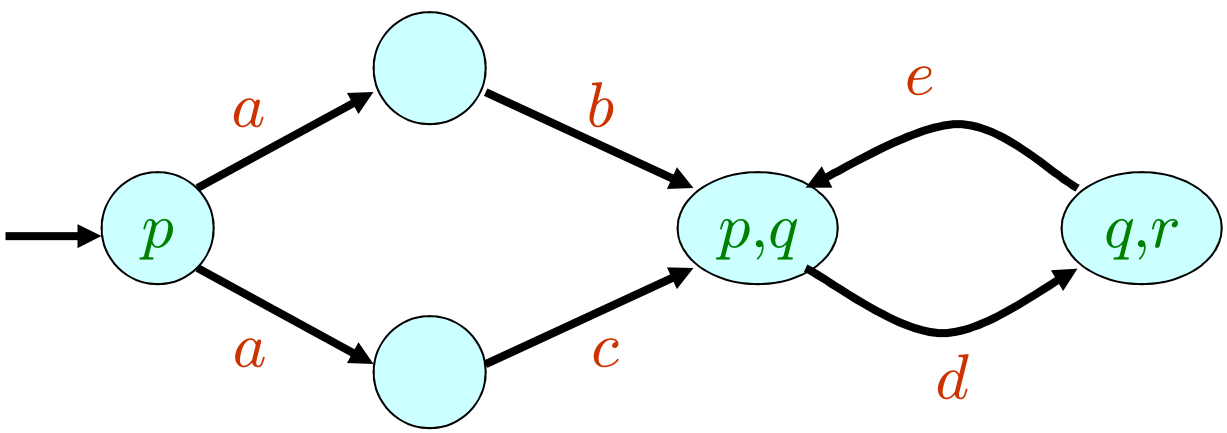
# Query 1: State-Event LTL Formulas

Whenever $p$ holds, $e$ happens some time in the future: $G\ (\ p \Rightarrow F\ \ e\ )$



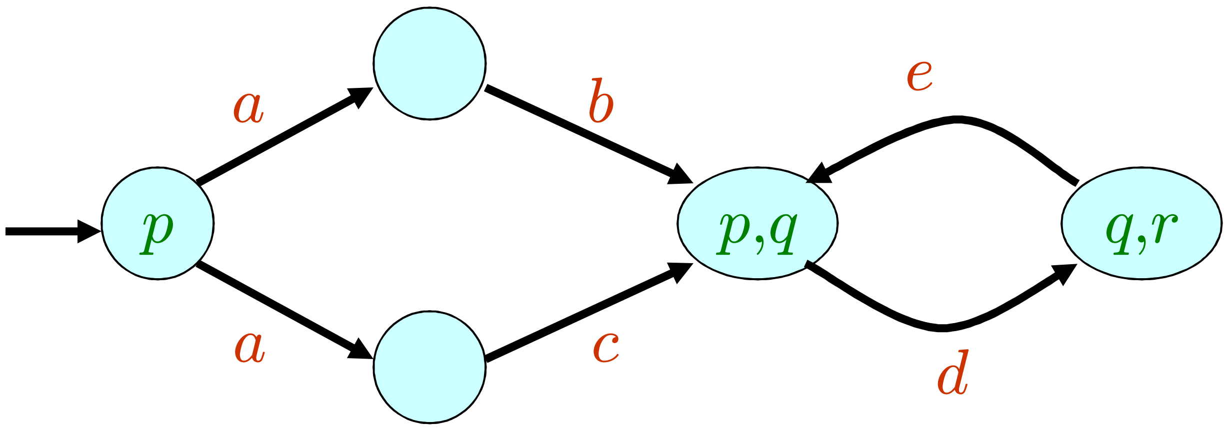
# Query 2: State-Event LTL Formulas

$p$ and $r$ are never true at the same time: $G(\neg p \vee \neg r)$



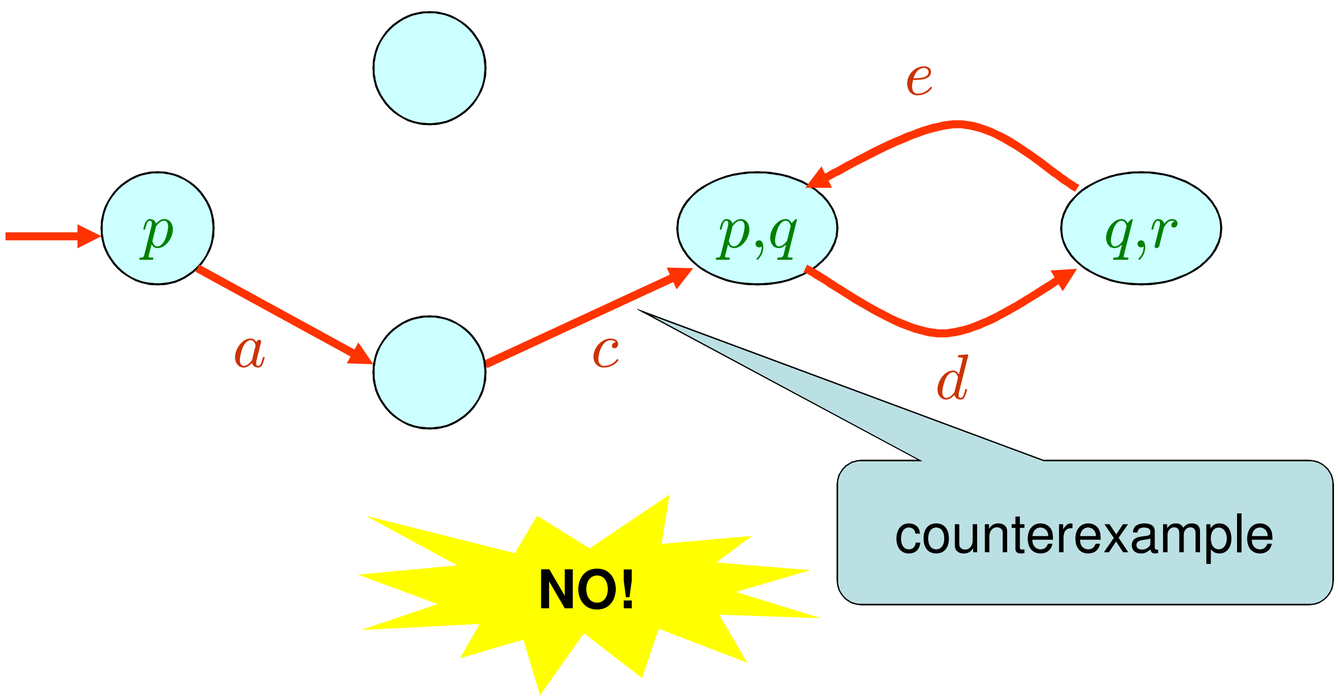

# Query 3: State-Event LTL Formulas

Whenever $p$ holds, $a$ happens some time in the future: $G\ (\ p \Rightarrow F\ \ a\ )$



# Query 3: State-Event LTL Formulas

Whenever $p$ holds, $a$ happens some time in the future: $G\ (\ p \Rightarrow F\ \ a\ )$



# Scalability of Model Checking

Explicit statespace exploration: early 1980s

- Tens of thousands of states

Symbolic statespace exploration: millions of states

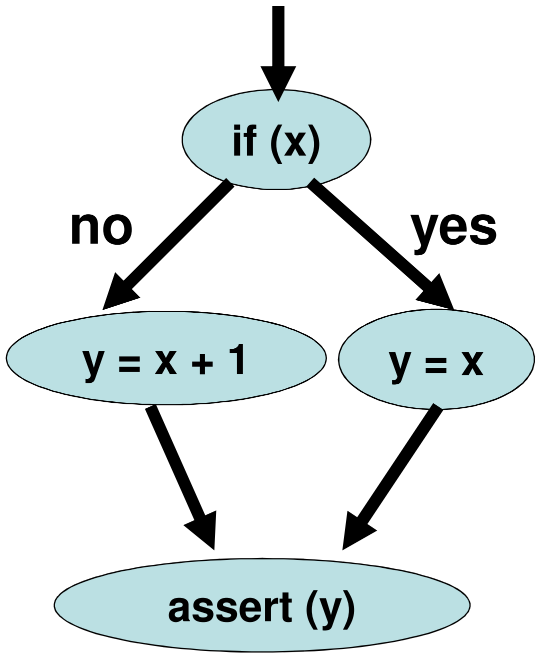
- Binary Decision Diagrams (BDD) : early 1990's
- Bounded Model Checking: late 1990's
  - Based on propositional satisfiability (SAT) technology

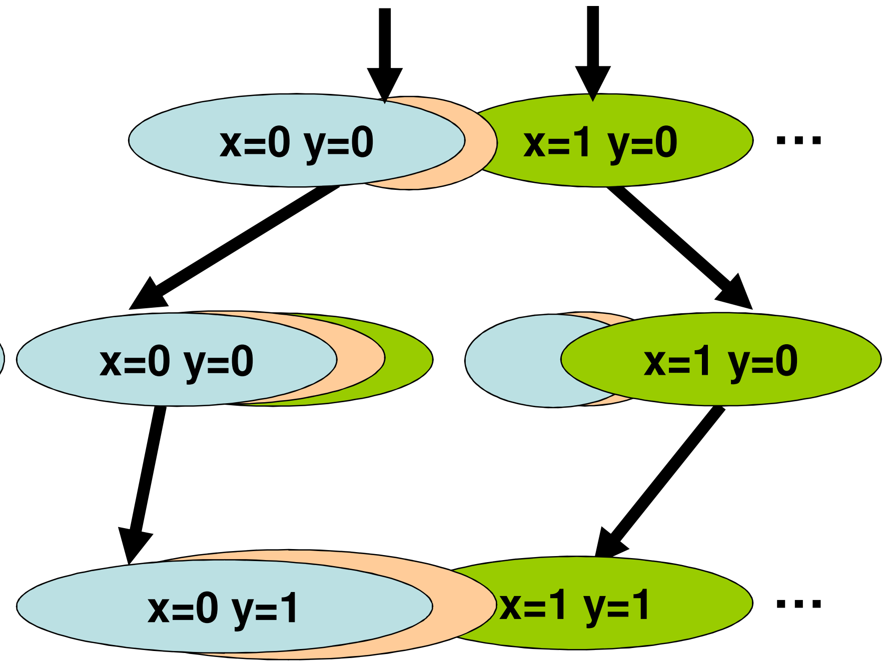
Abstraction and compositional reasoning

- $10^{120}$ to effectively infinite statespaces (particularly for software)

# Models of C Code

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




**Program: Syntax** **Control Flow Graph** **Model: Sem**

**Infinite State**

# Abstraction

Partition concrete statespace into abstract states

- Each abstract state $S$ corresponds to a set of concrete states $s$
- We write $\alpha(s)$ to mean the abstract state corresponding to $s$
- We define $\gamma(S) = \{ s \mid S = \alpha(s) \}$

Fix the transitions existentially

- $S \rightarrow S' \quad \Leftrightarrow \quad \exists\, s \in \gamma(S) \,.\, \exists\, s' \in \gamma(S') \,.\, s \rightarrow s'$
- $S \rightarrow S' \quad \Leftarrow \quad \exists\, s \in \gamma(S) \,.\, \exists\, s' \in \gamma(S') \,.\, s \rightarrow s'$

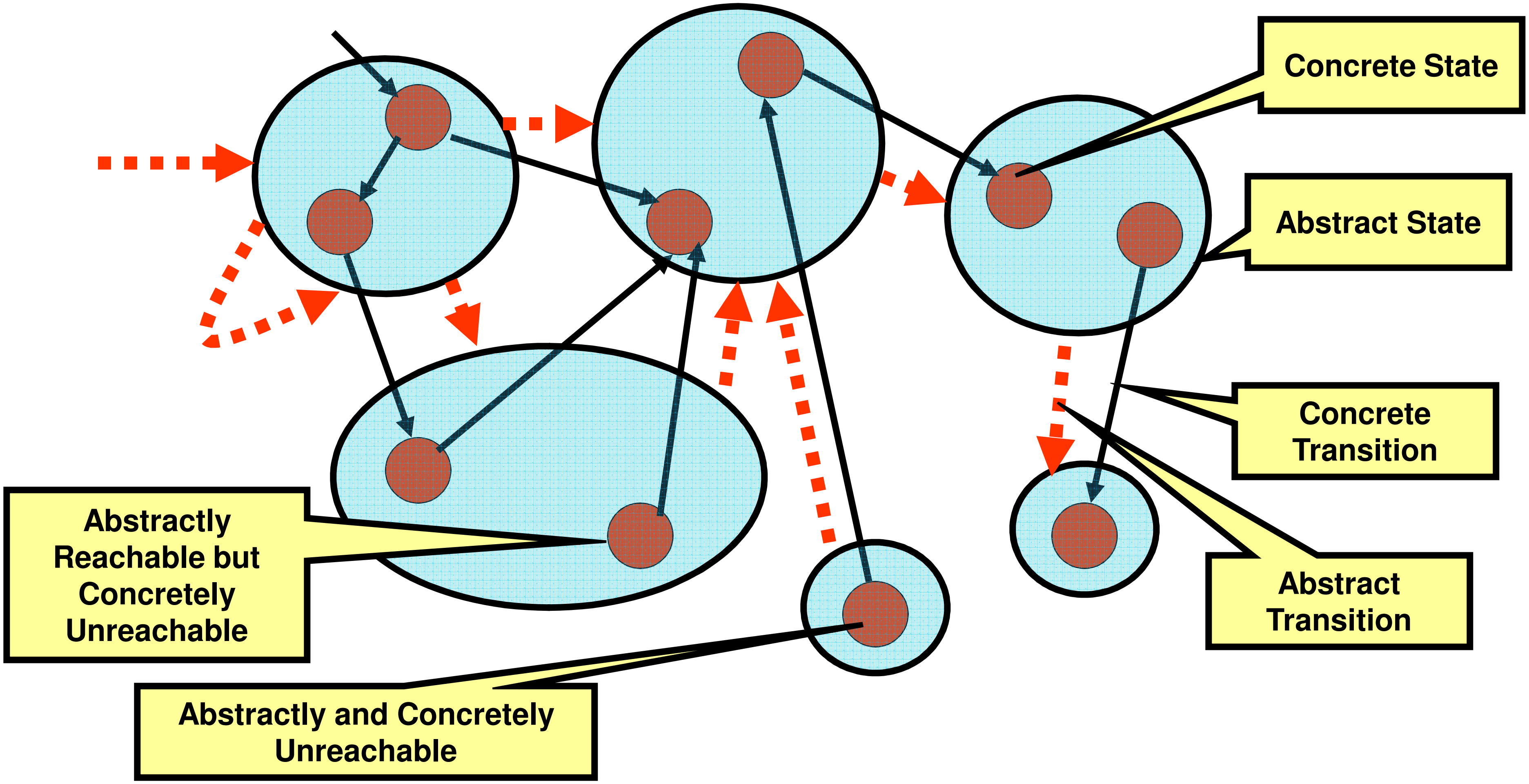
**Strong & sometimes not computable**

**Weak: computable**

Abstraction is conservative

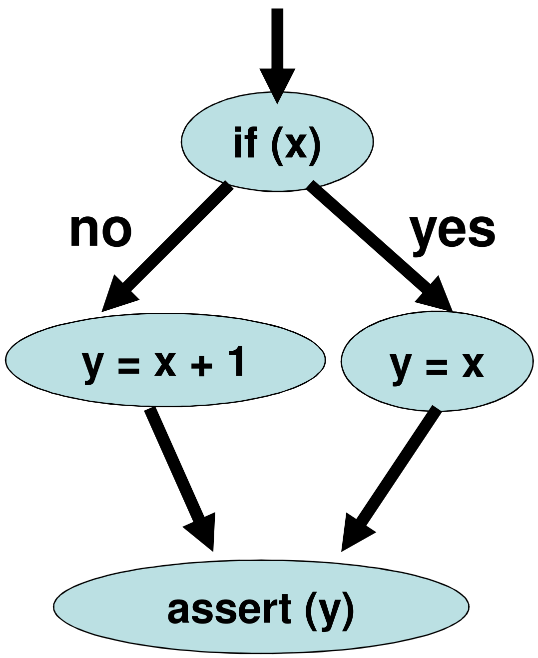
- If a State/Event-LTL property holds on the abstraction, it also holds on the program
- However, the converse is not true: a property that fails on the abstraction may still hold on the program

# Example



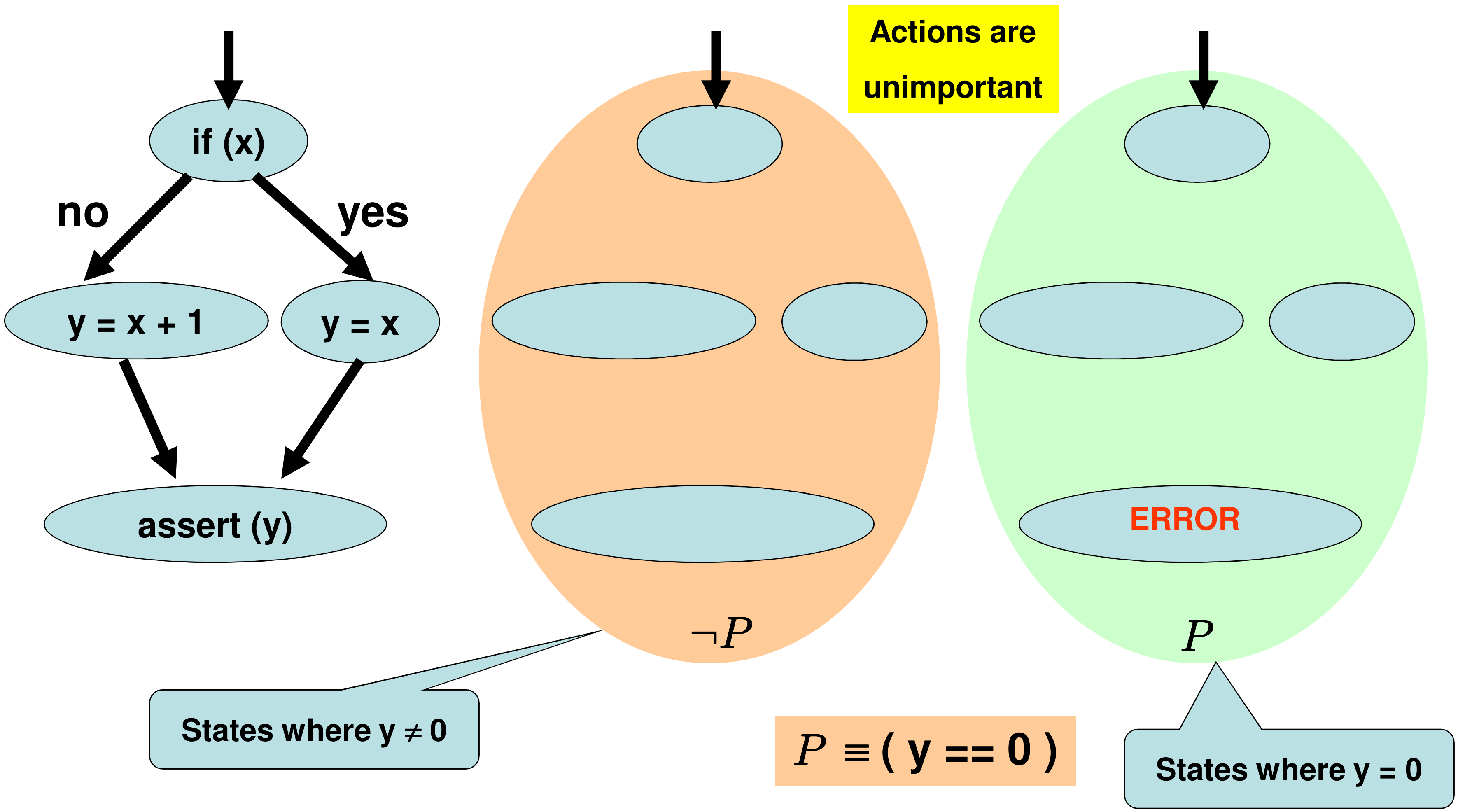
Concrete State
Abstract State
Concrete Transition
Abstract Transition
Abstractly Reachable but Concretely Unreachable
Abstractly and Concretely Unreachable

# Predicate Abstraction

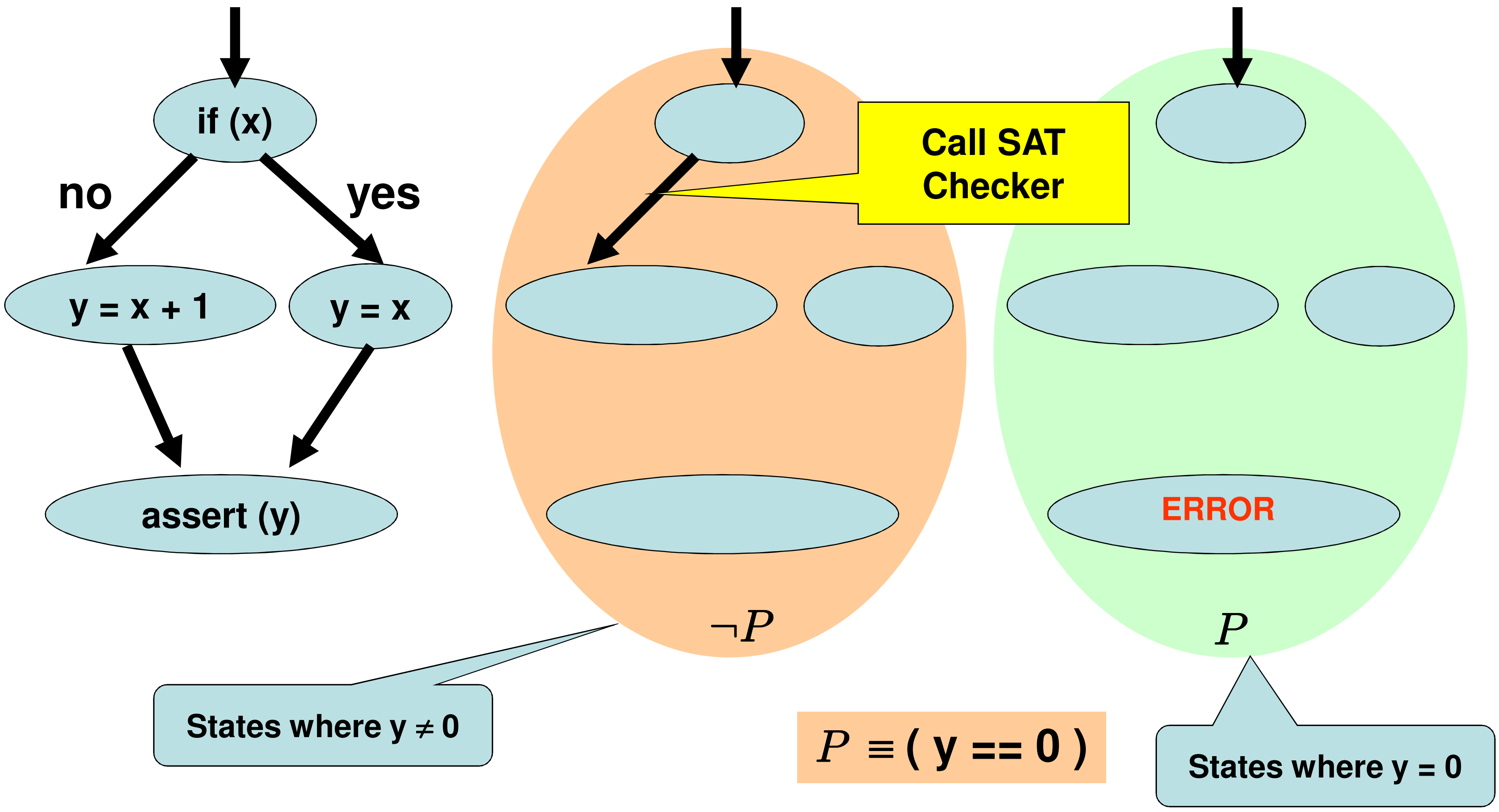



Partition the statespace based on values of a finite set of predicates on program variables

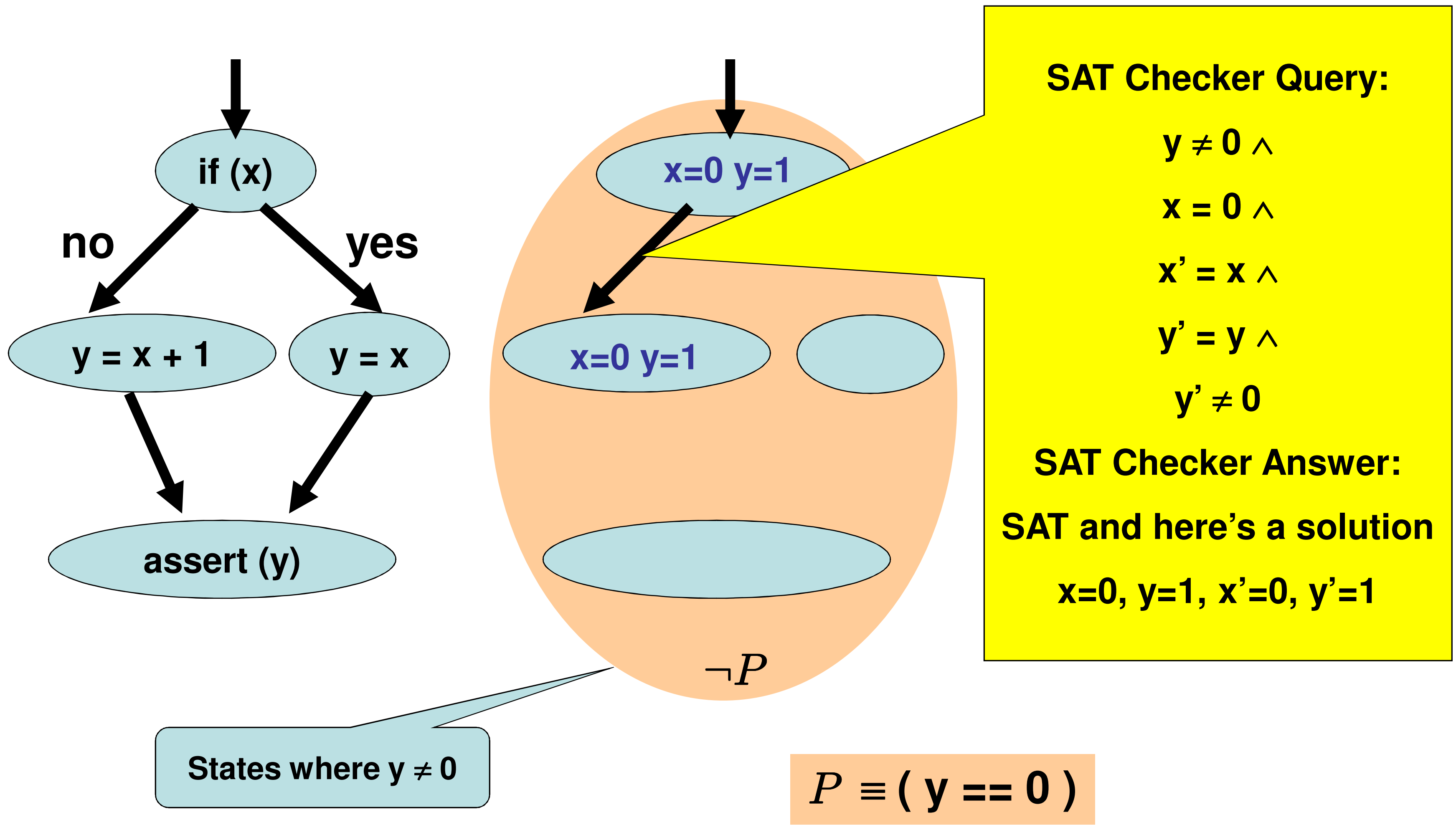
# Predicate Abstraction



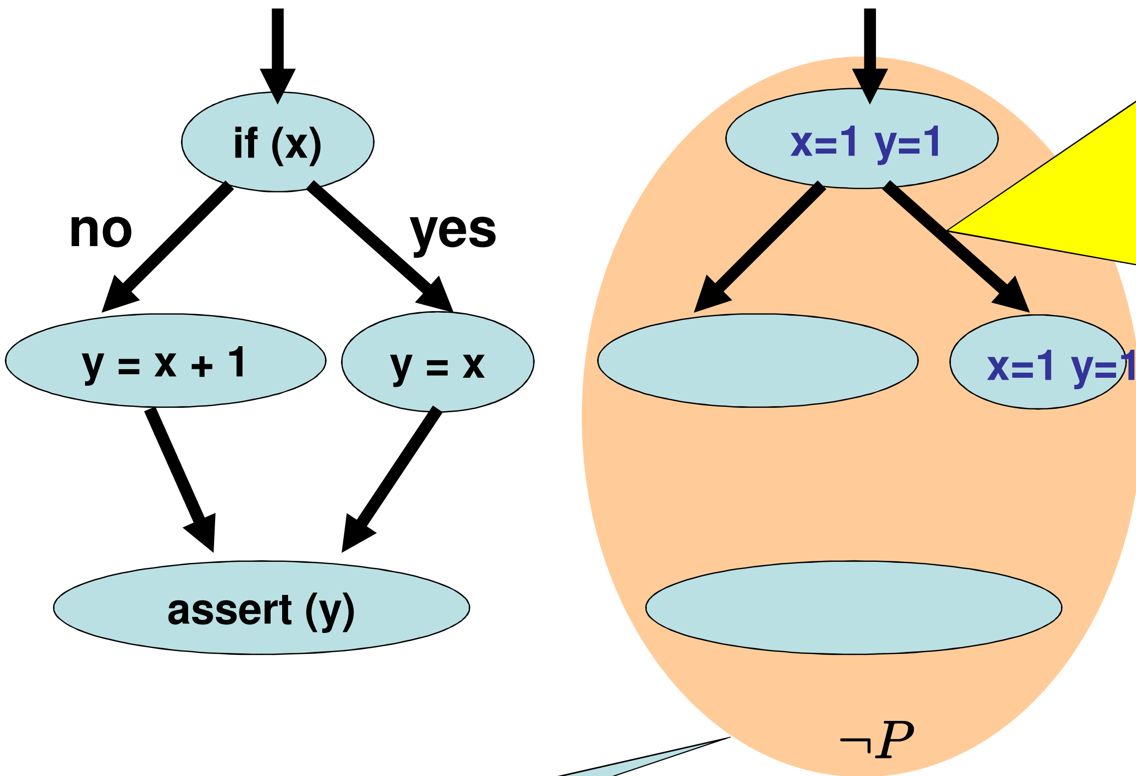
# Predicate Abstraction



# Predicate Abstraction



# Predicate Abstraction




States where $y \neq 0$

**SAT Checker Query:**

$y \neq 0 \wedge$

$x \neq 0 \wedge$

$x' = x \wedge$

$y' = y \wedge$

$y' \neq 0$

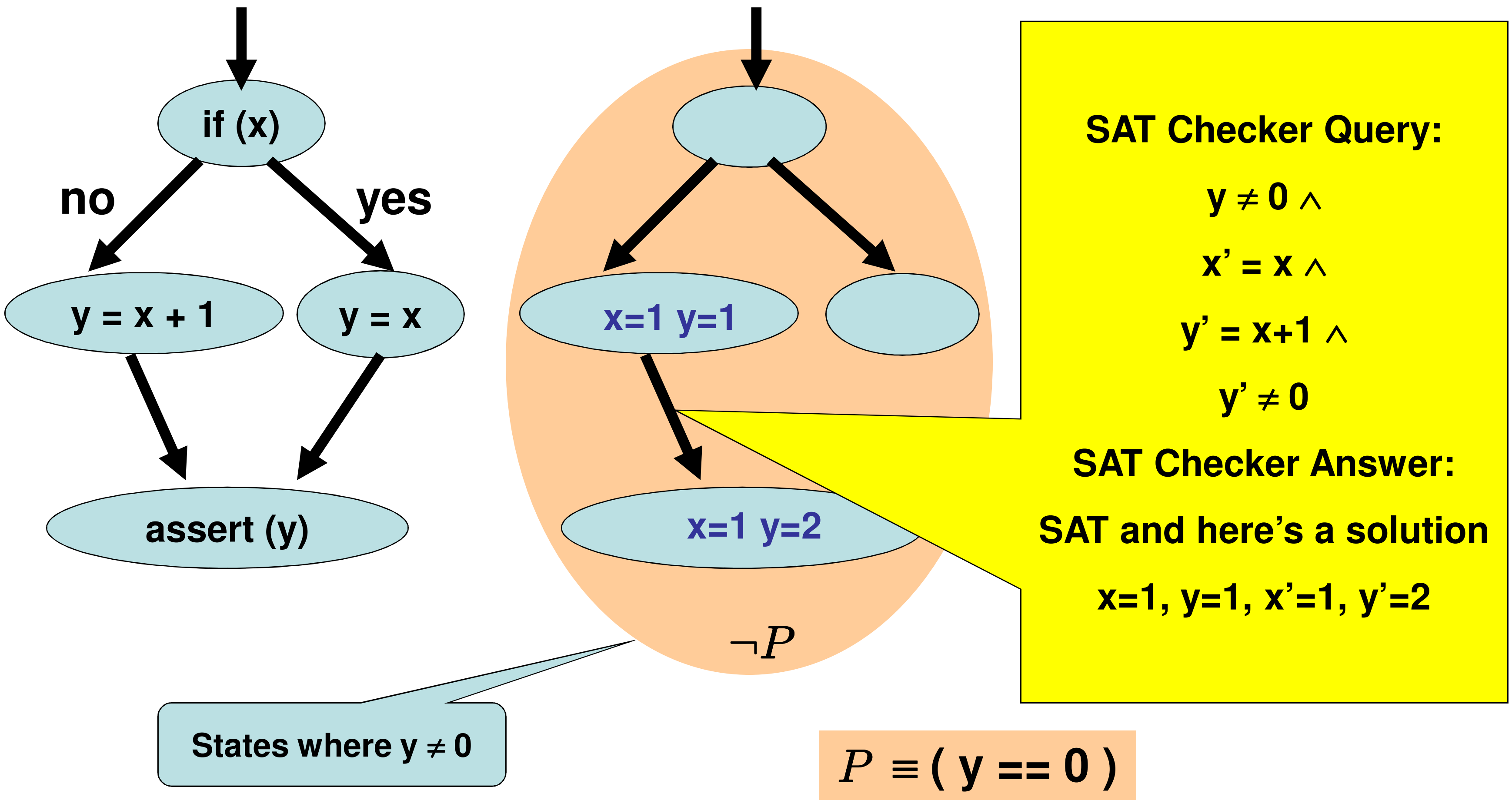
**SAT Checker Answer:**

**SAT and here's a solution**

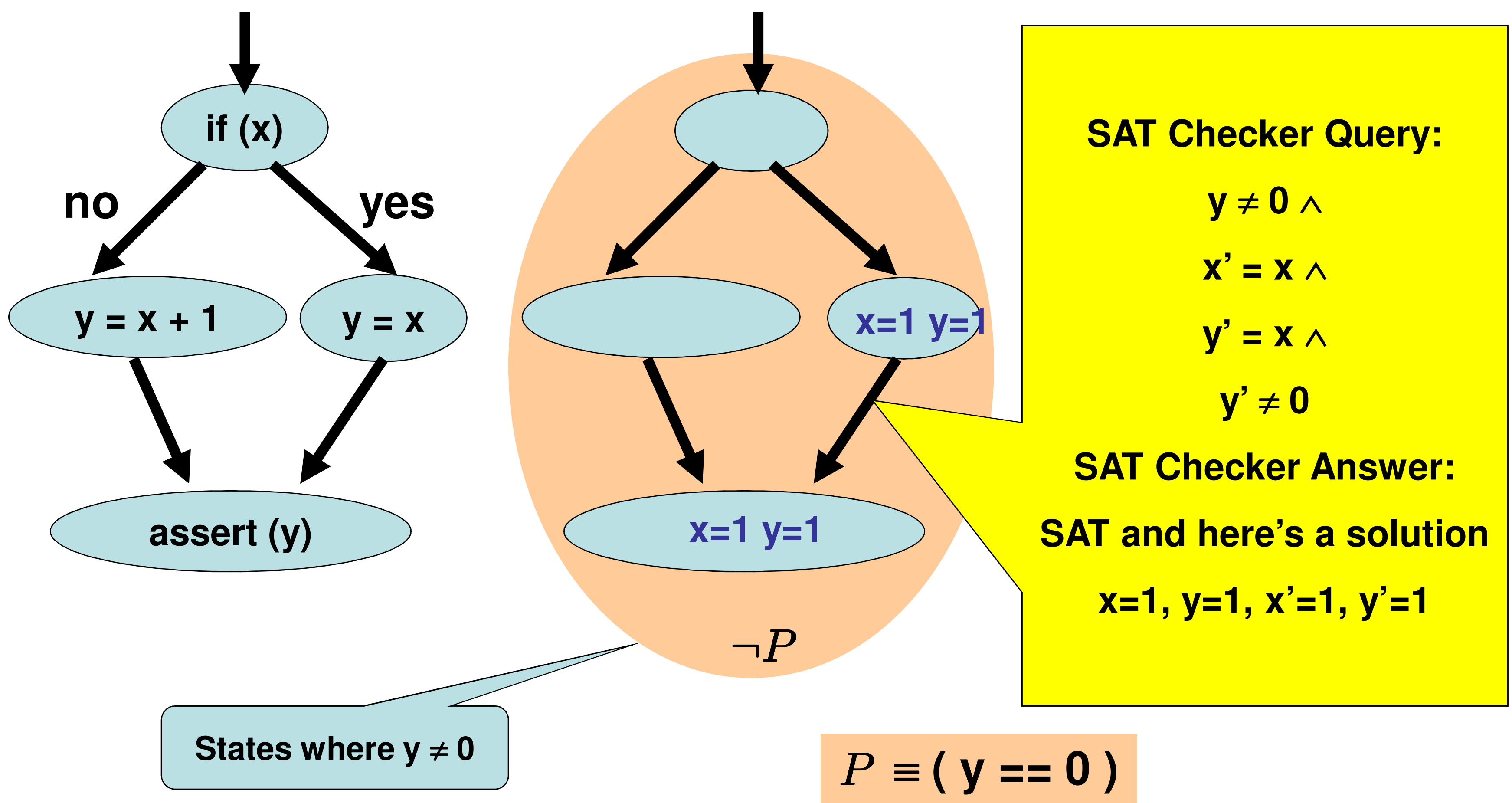
**x=1, y=1, x'=1, y'=1**

$P \equiv ( y == 0 )$

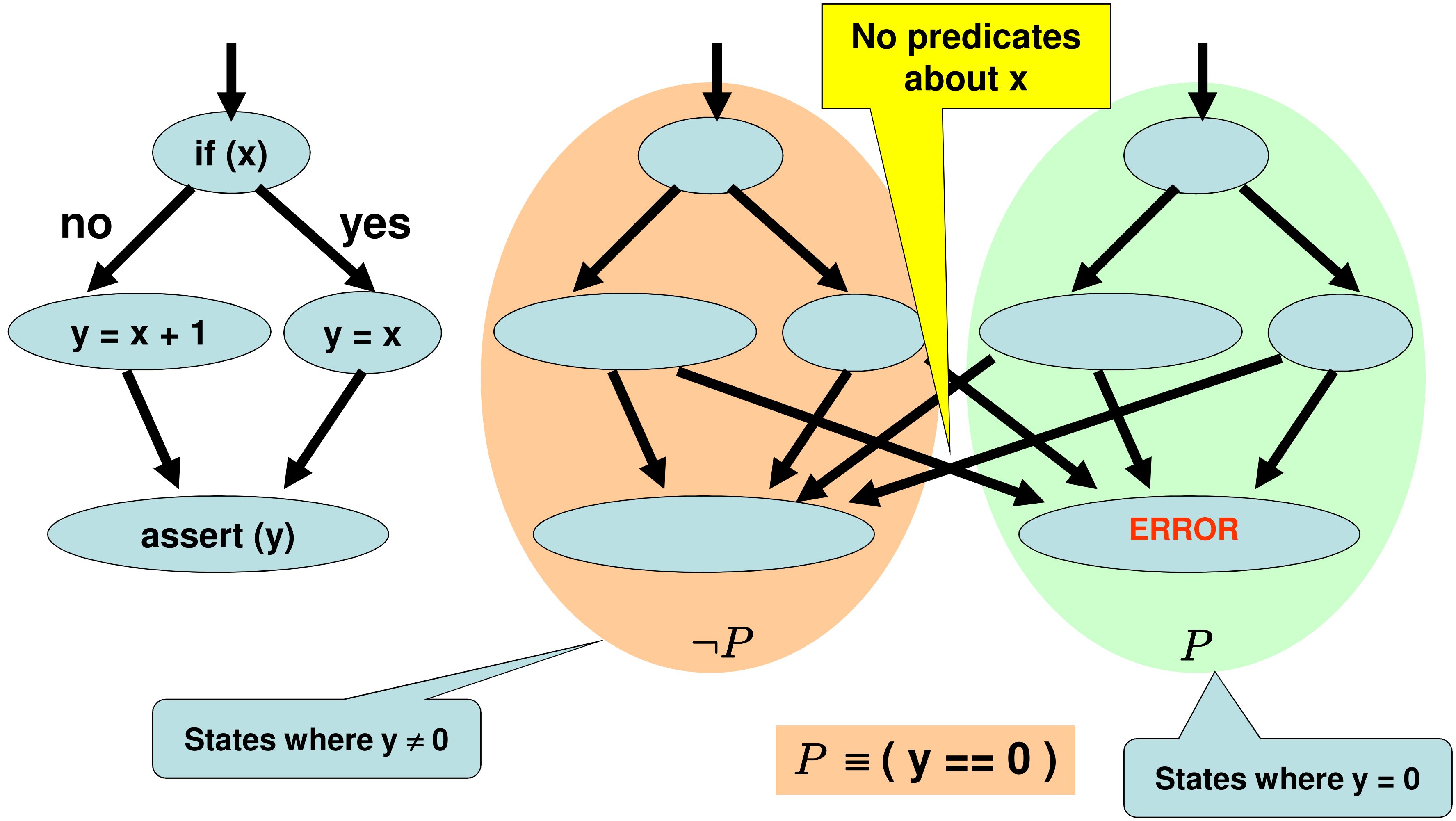
# Predicate Abstraction



# Predicate Abstraction



# Predicate Abstraction




$P \equiv$ ( y == 0 )

# Imprecision due to Predicate Abstraction

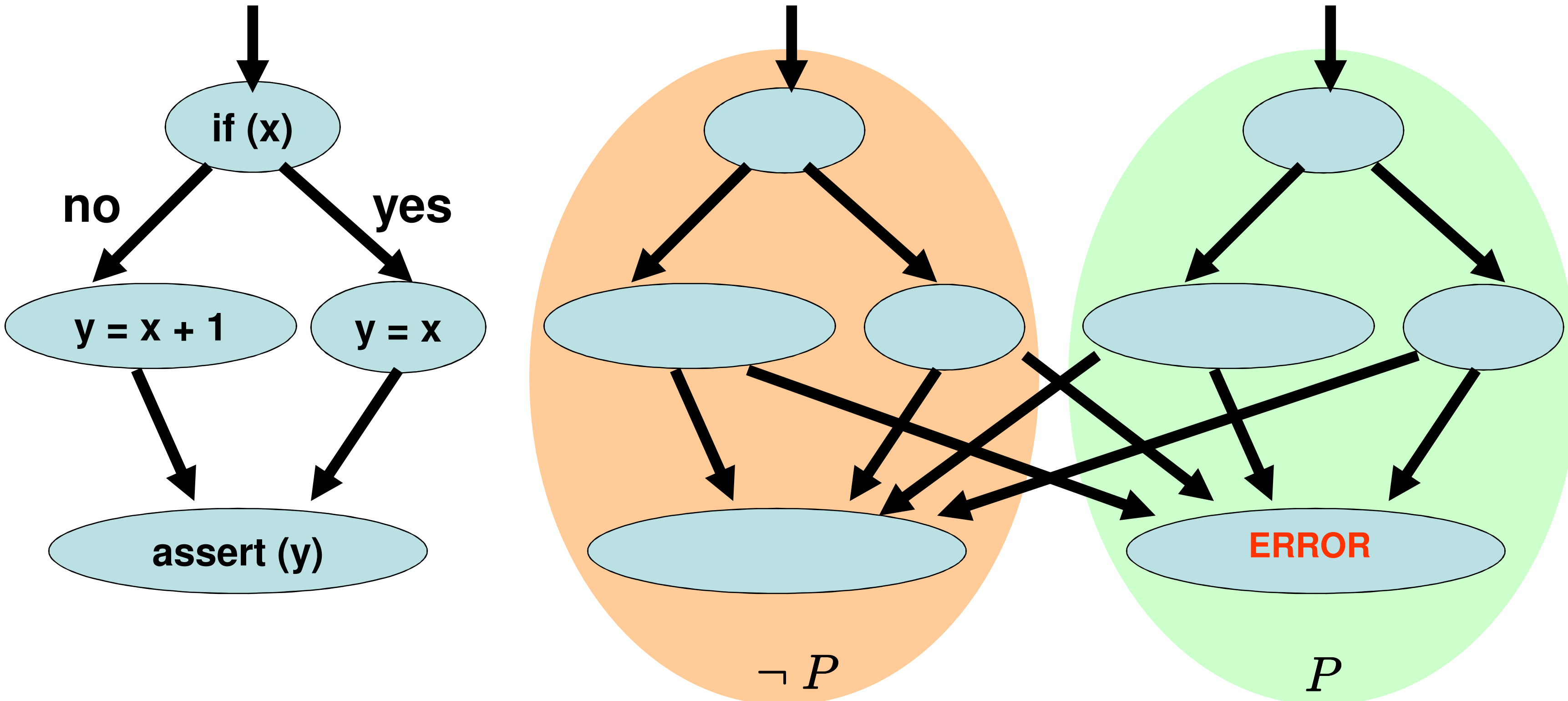
Counterexamples generated by model checking the abstract model may be spurious, i.e., not concretely realizable

Need to refine the abstraction iteratively by changing the set of predicates

Can infer new set of predicates by analyzing the spurious counterexample

- Lot of research in doing this effectively
- Counterexample Guided Abstraction Refinement (CEGAR)
- A.K.A. Iterative Abstraction Refinement
- A.K.A. Iterative Refinement

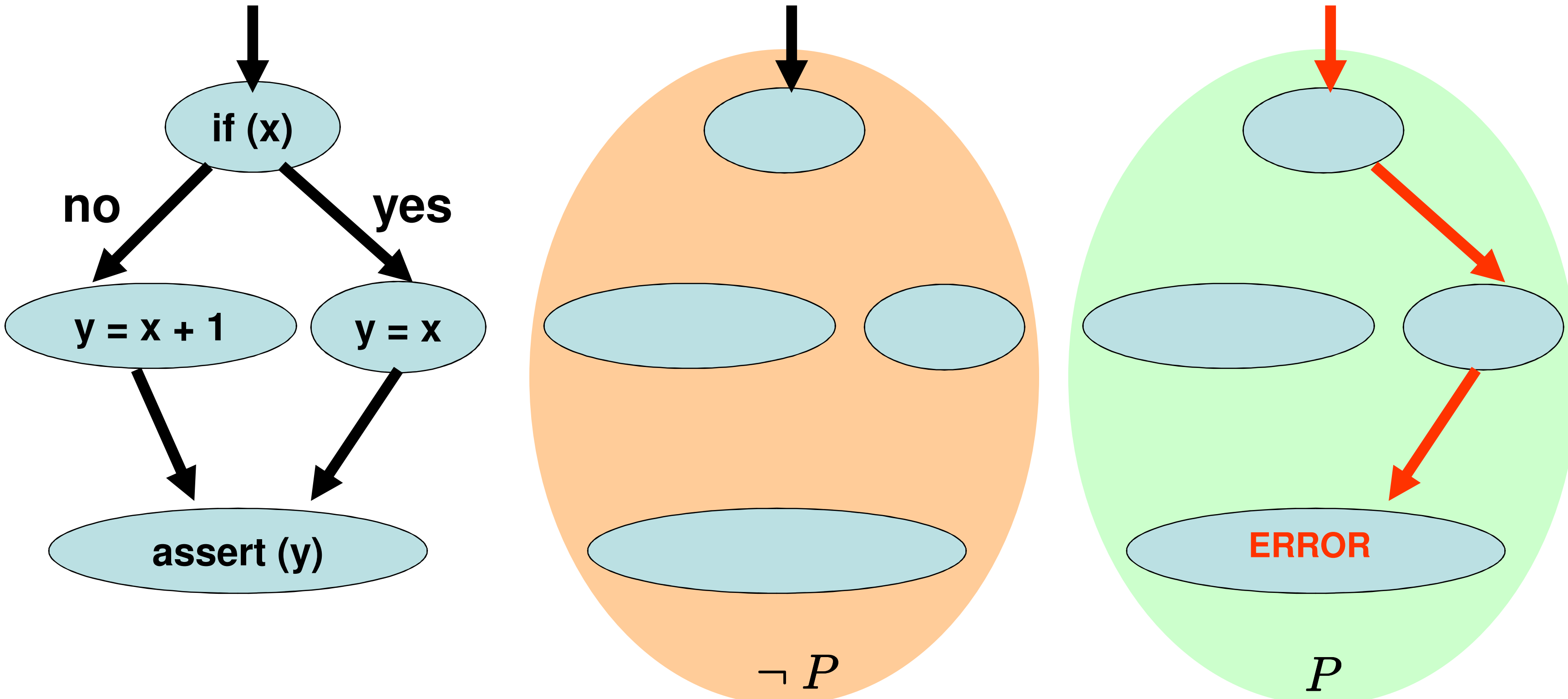
# Model Checking




$\phi = G(\neg$ ERROR)

$P \equiv$ ( y == 0 )

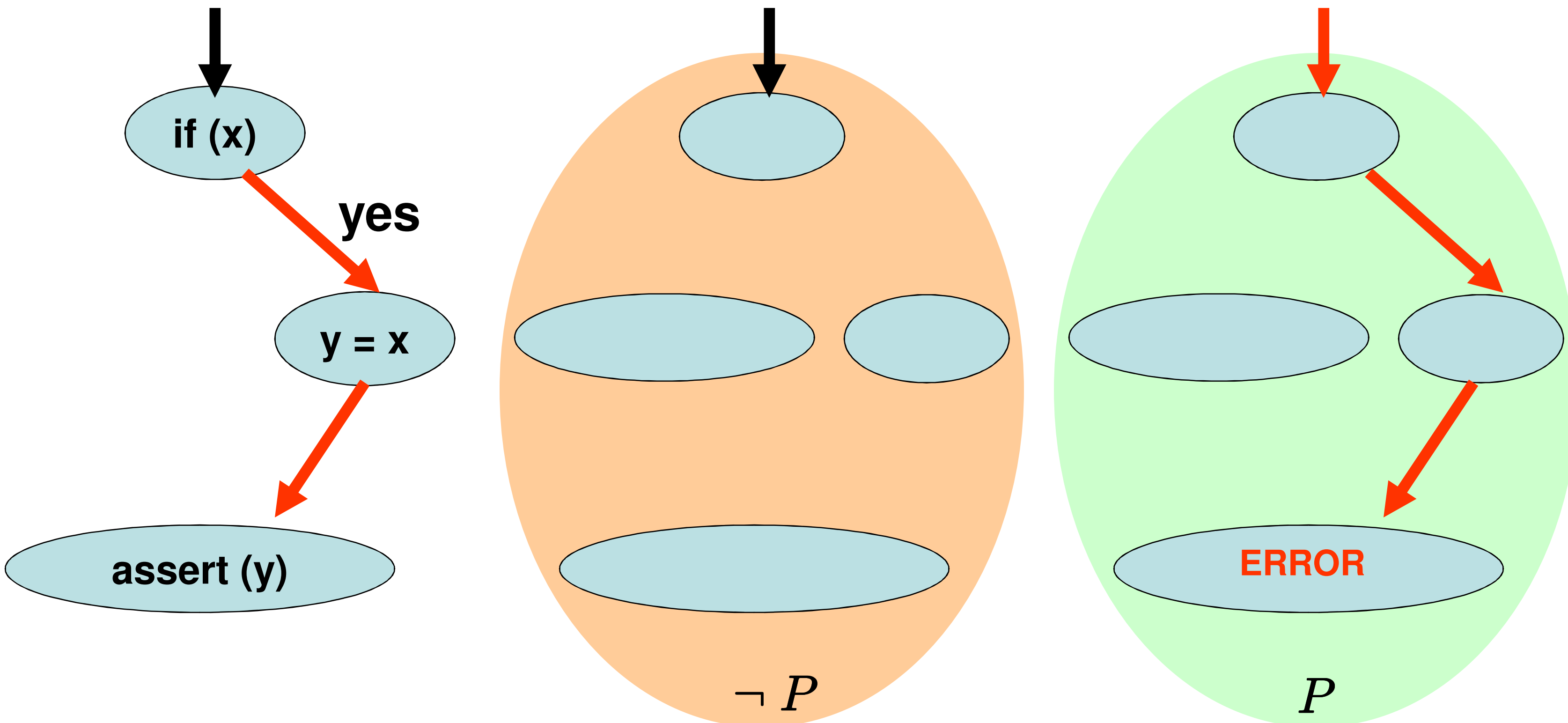
# Model Checking




$\phi$ = $G$(¬ ERROR)

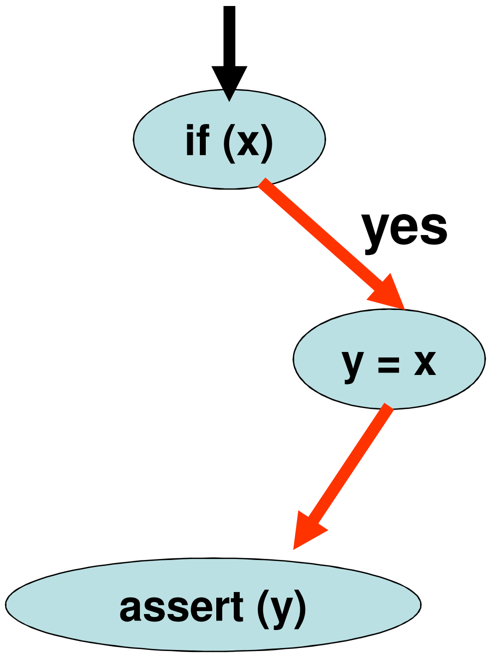
$P$ ≡ ( y == 0 )

# Model Checking



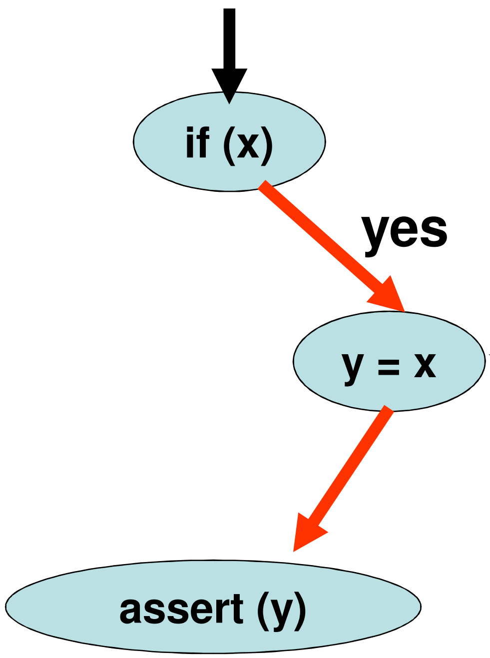

$P \equiv$ ( y == 0 )

# Counterexample Validation




- Simulate counterexample symbolically
- Call SAT Checker to determine if the post-condition is satisfiable
- In our case, Counterexample is spurious
- New set of predicates {x==0,y==0}

# Counterexample Validation




**SAT Checker Query:**

$x \neq 0 \wedge$

$y' = x \wedge$

$y' = 0$

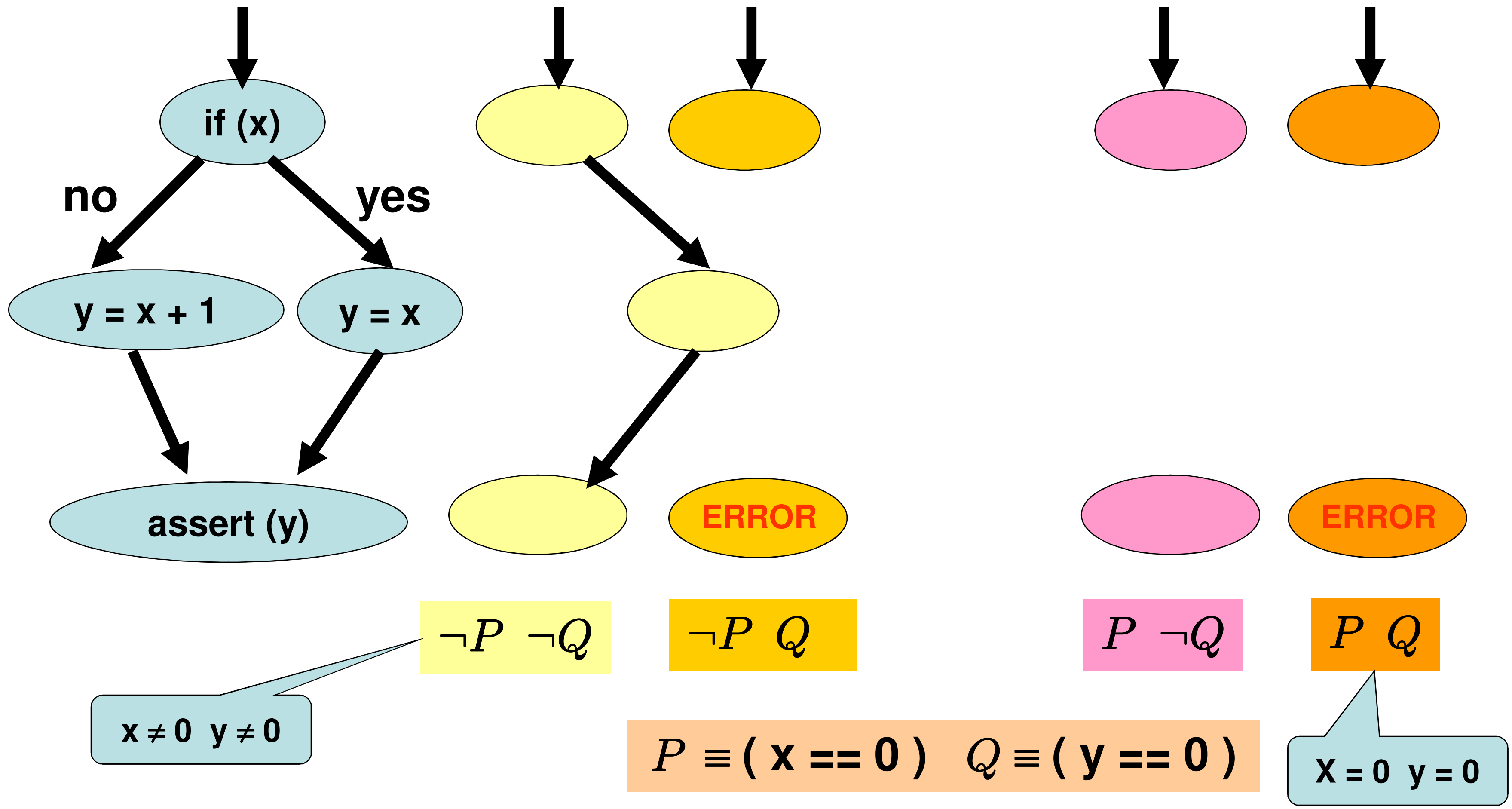
**SAT Checker Answer:**

**UNSAT and here's an UNSAT core**

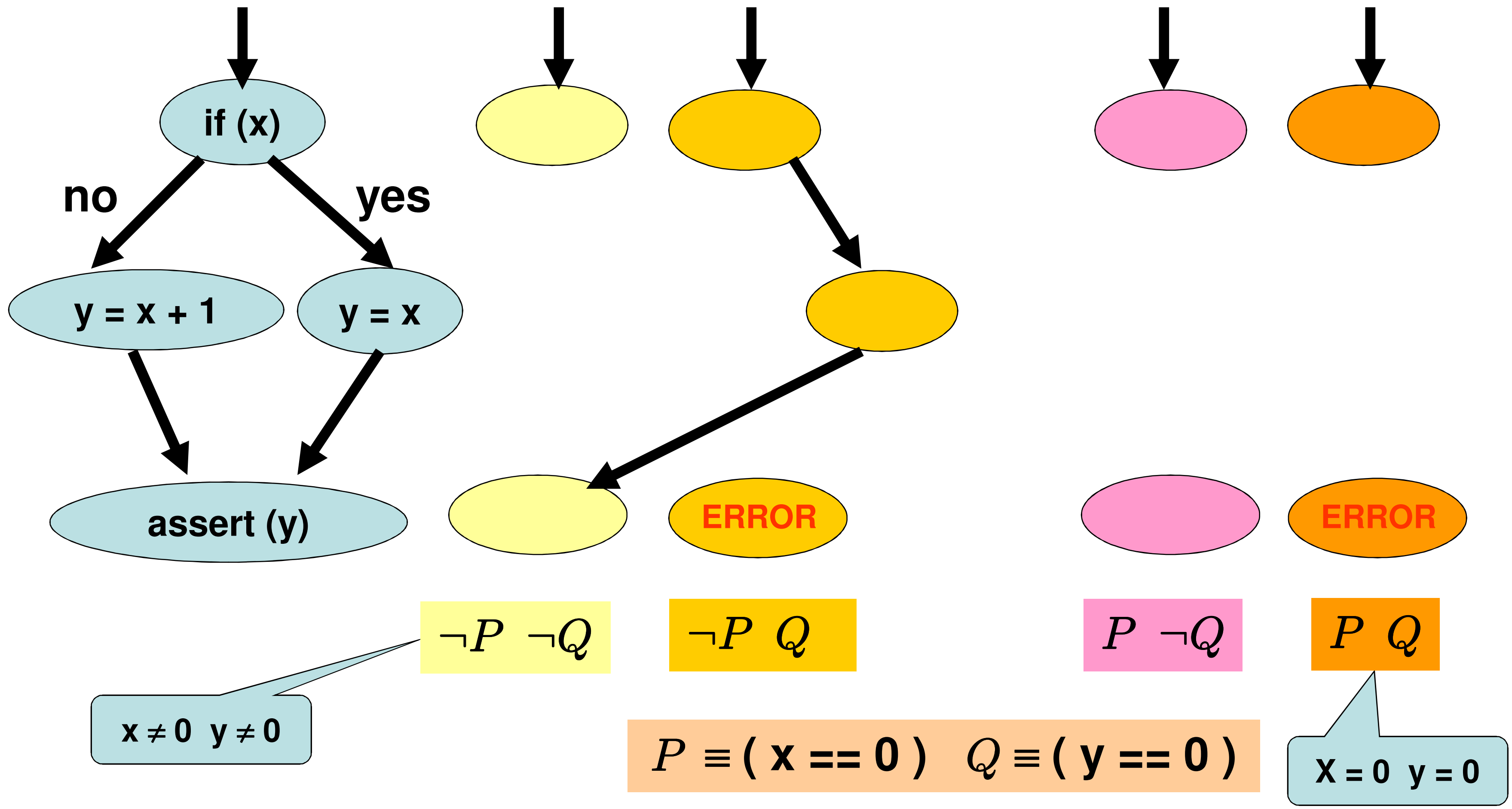
$\{x \neq 0 \,,\, y' = x \,,\, y' = 0\}$

- Used to derive new predicate (x=0)
- Different heuristics used in practice

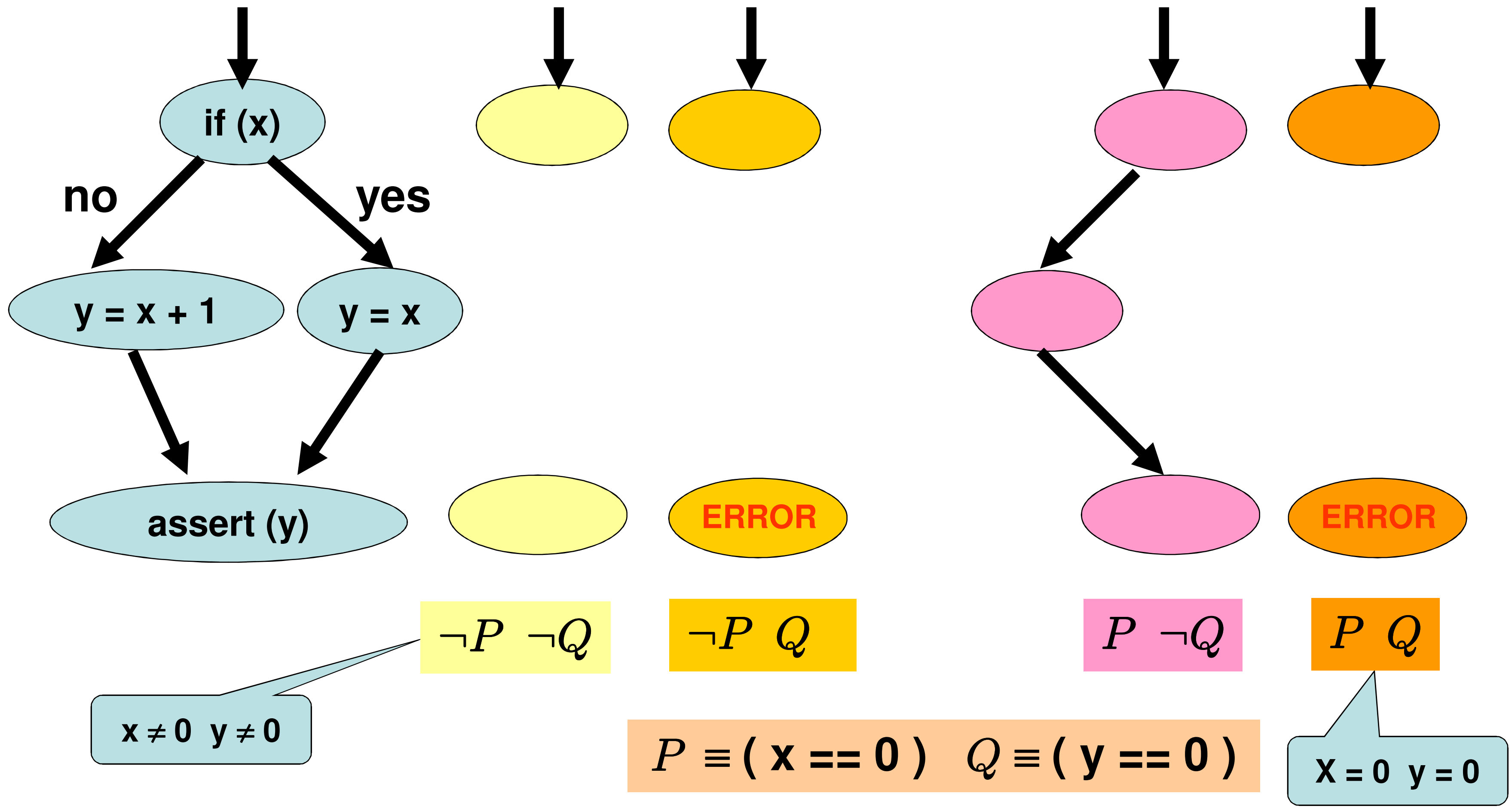
# Predicate Abstraction: 2nd Iteration



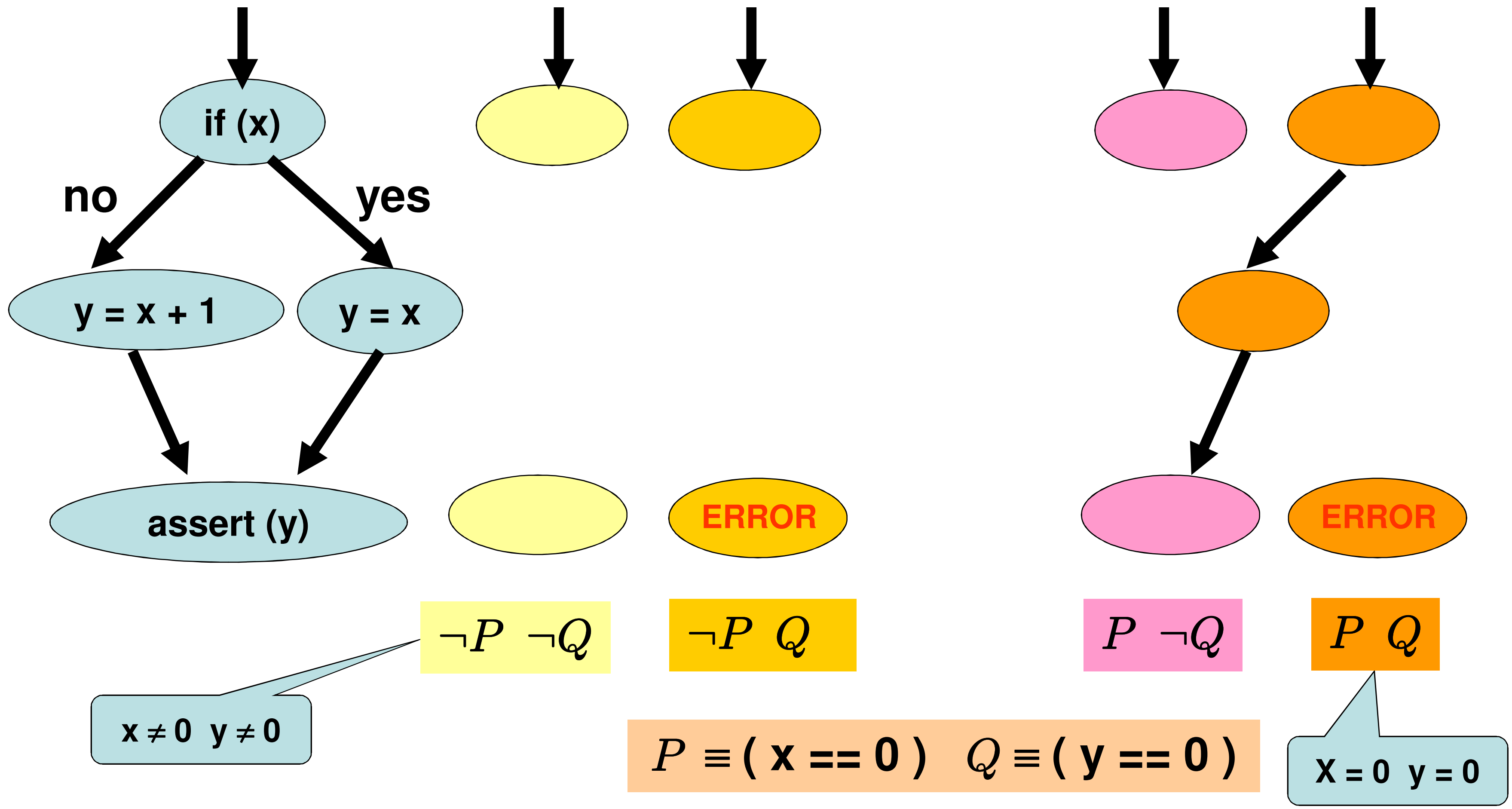
# Predicate Abstraction: 2nd Iteration



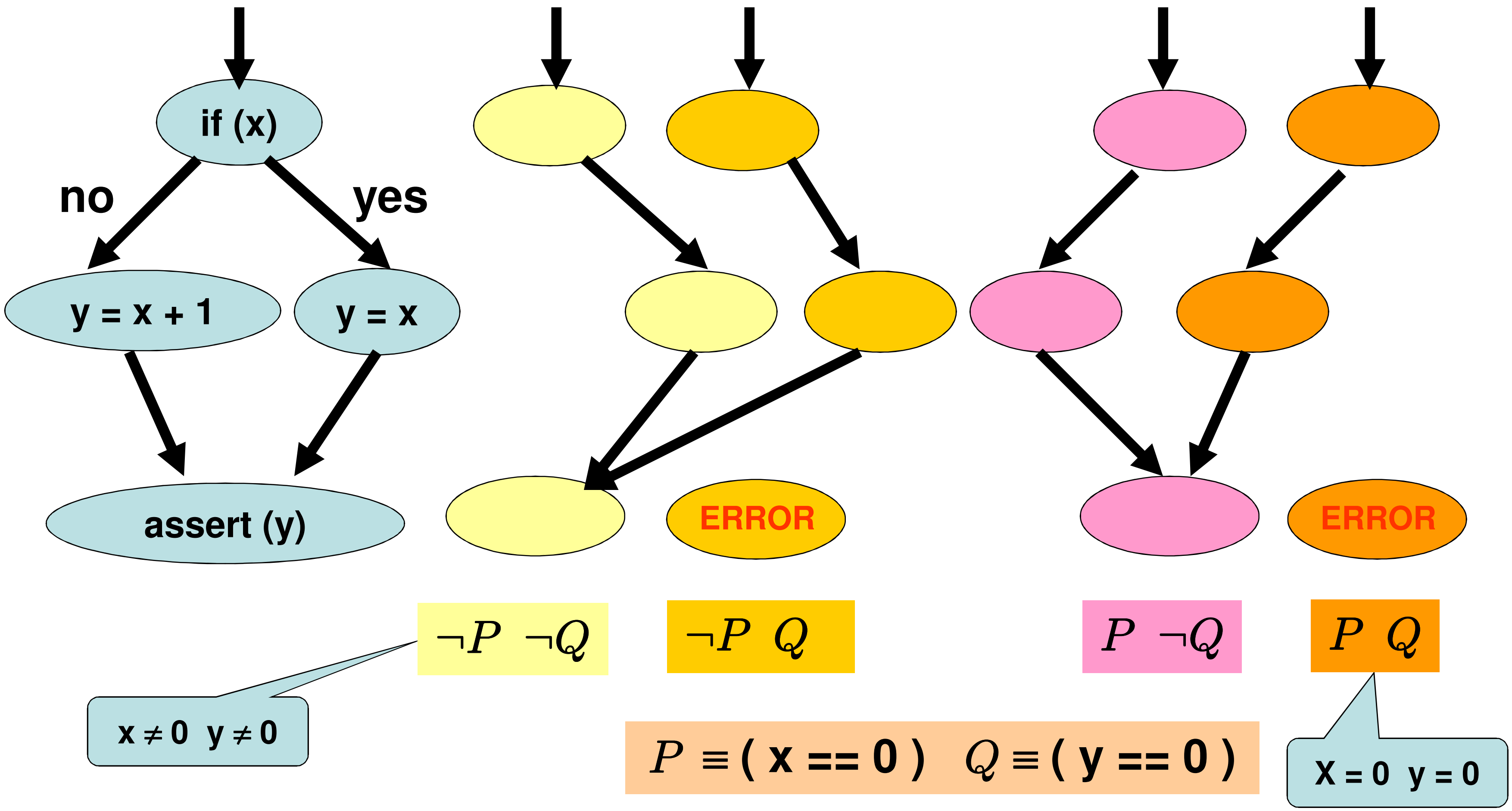
# Predicate Abstraction: 2nd Iteration



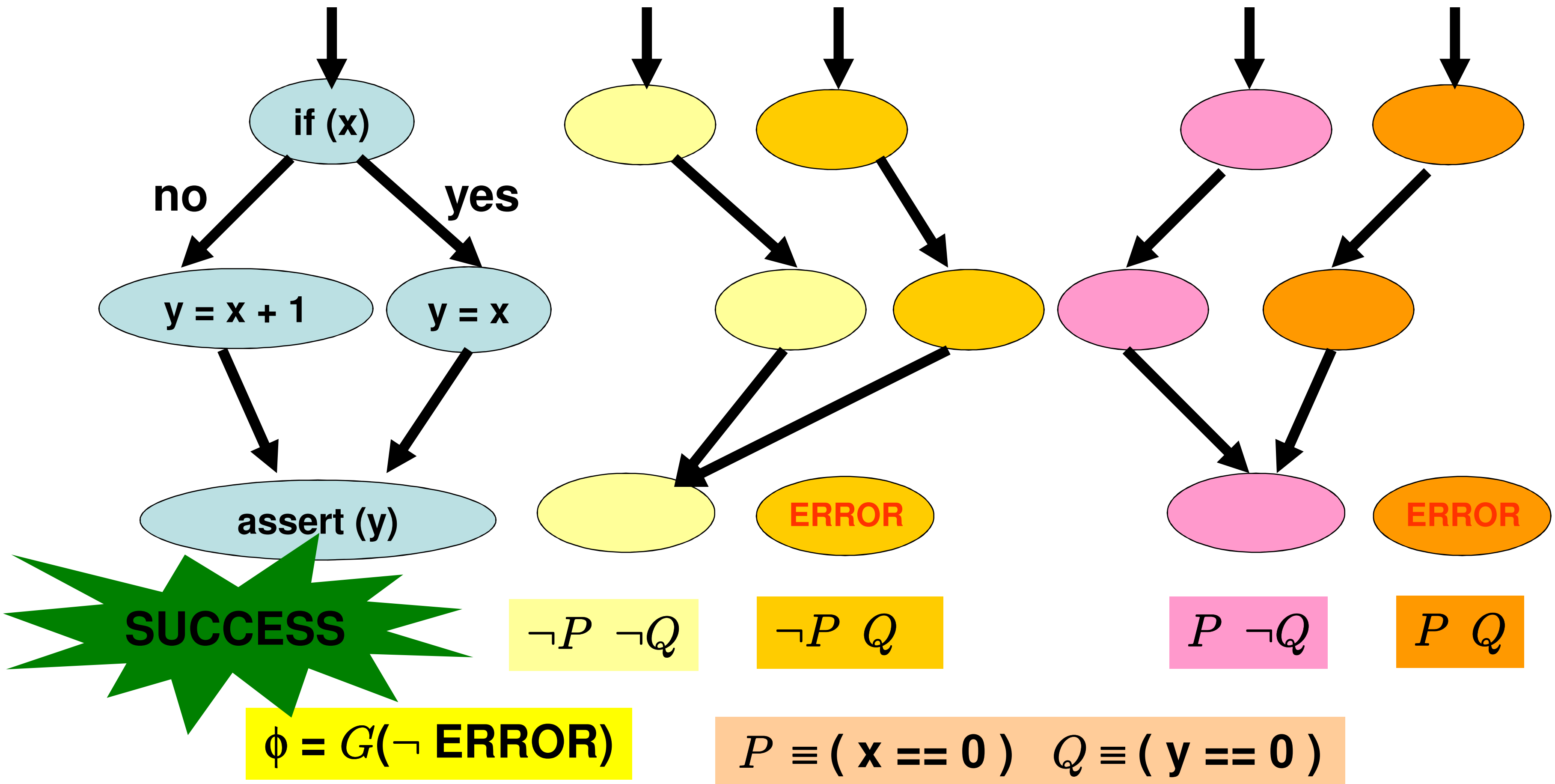
# Predicate Abstraction: 2nd Iteration



# Predicate Abstraction: 2nd Iteration



# Model Checking: 2nd Iteration

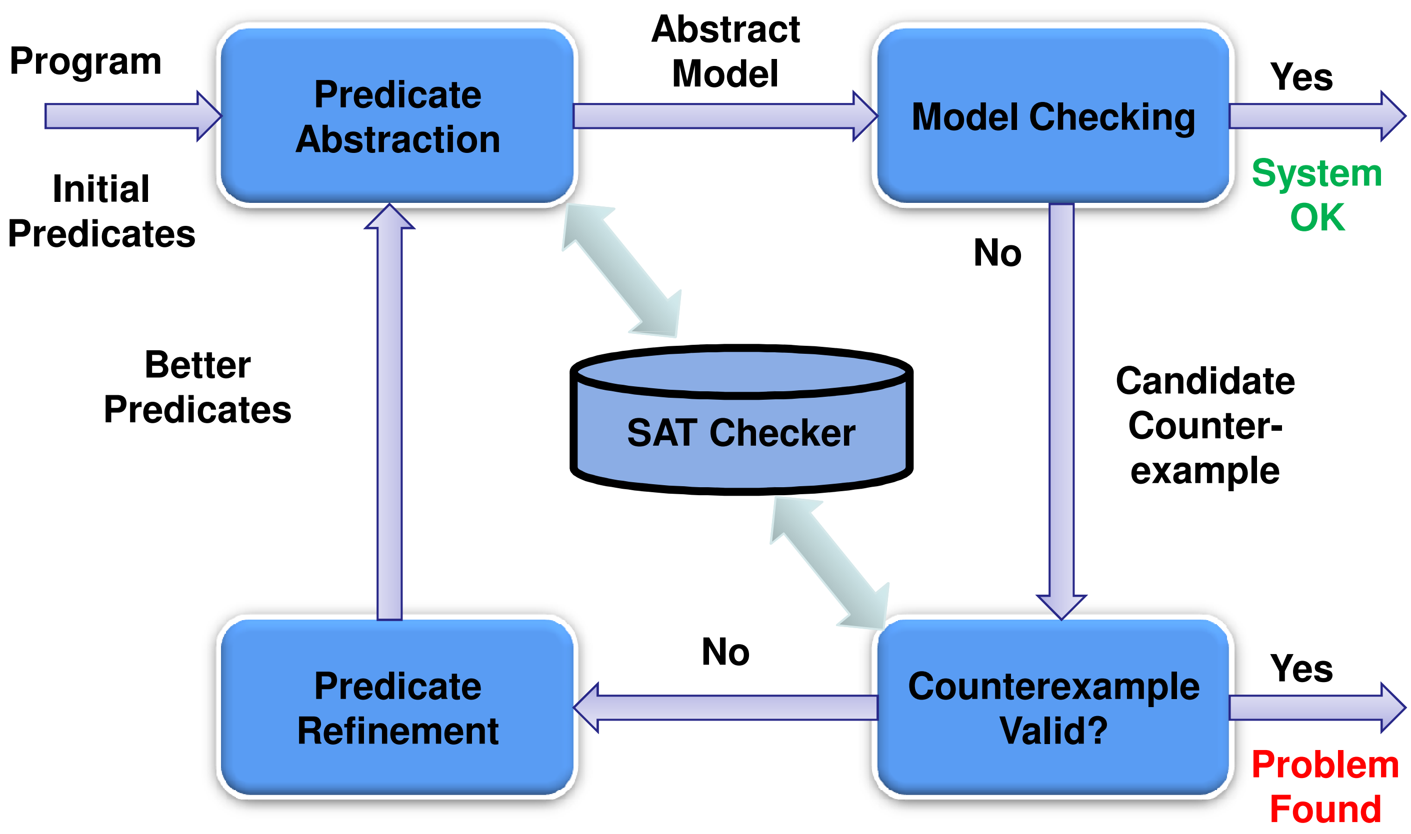


# Iterative Refinement: Summary

Choose an initial set of predicate, and proceed iteratively as follows:

1. **Abstraction:** Construct an abstract model $M$ of the program using the 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.

# Iterative Refinement



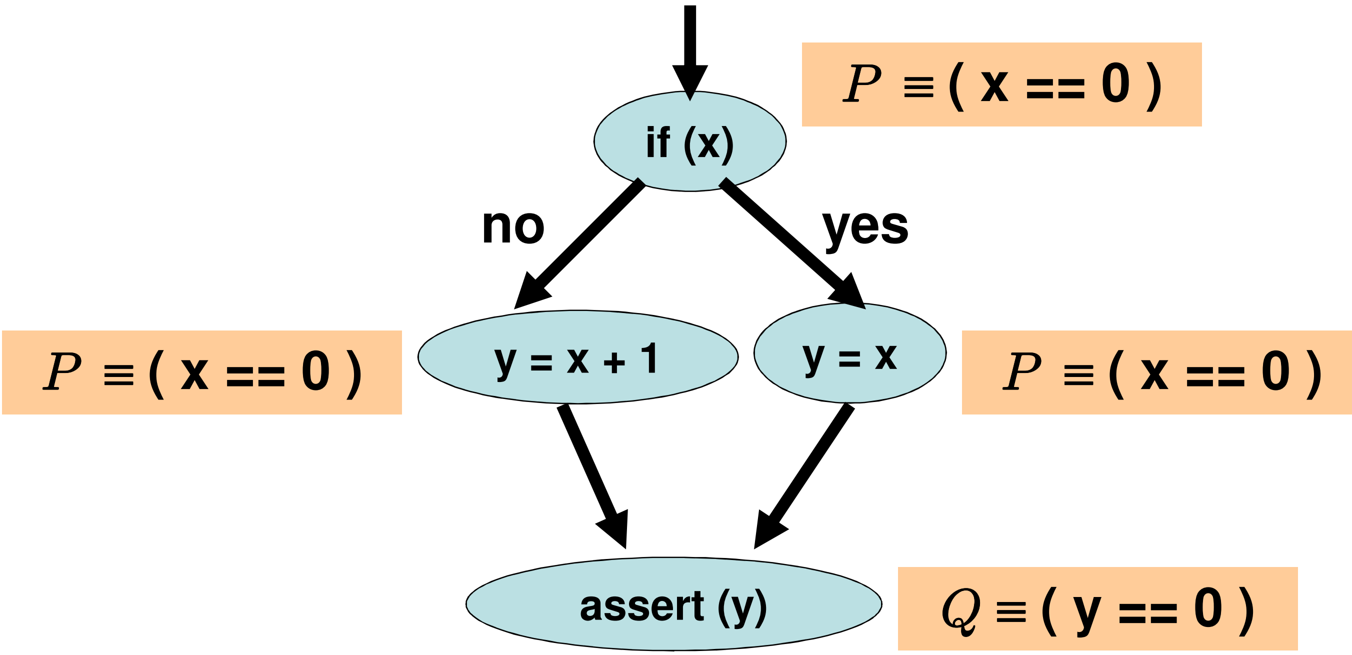

Localization Reduction, Kurshan, Bell Labs

Counterexample-guided Abstraction Refinement for Symbolic Model Checking, Clarke et al., CMU

Software Model Checking, SLAM Project, Microsoft, Ball & Rajamani

# Predicate Abstraction: Optimizations

1. Construct transitions on-the-fly
2. Different set of predicates at different control locations




3. Avoid exponential number of theorem-prover calls

# Research Areas

Finding “good” predicates

- Technically as hard as finding “good” loop invariants
- Complexity is linear in LOC but exponential in number of predicates

Combining with static analysis

- Alias analysis, invariant detection, constant propagation
- Inexpensive, and may make subsequent model checking more efficient

Bounded model checking

# Software Model Checking Tools

Iterative Refinement

- SLAM, BLAST, MAGIC, Copper, …

Bounded Model Checking

- CBMC, …

Others

- Engines: MOPED, BEBOP, BOPPO, …
- Java: Java PathFiner, Bandera, BOGOR, …
- C: CMC, …

# Bibliography

**Predicate Abstraction:** Construction of abstract state graphs with PVS, S. Graf, H. Saidi, Proceedings of Computer Aided Verification (CAV), 1997

**Abstraction Refinement for C:** Automatically Validating Temporal Safety Properties of Interfaces, T. Ball, S. Rajamani, Proceedings of the SPIN Workshop, 2001

**Software Model Checking Technology Transfer:** SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft, T. Ball, B. Cook, V. Levin, S. Rajamani, Proceedings of Intergrated Formal Methods, 2004

Software Engineering Institute | Carnegie Mellon