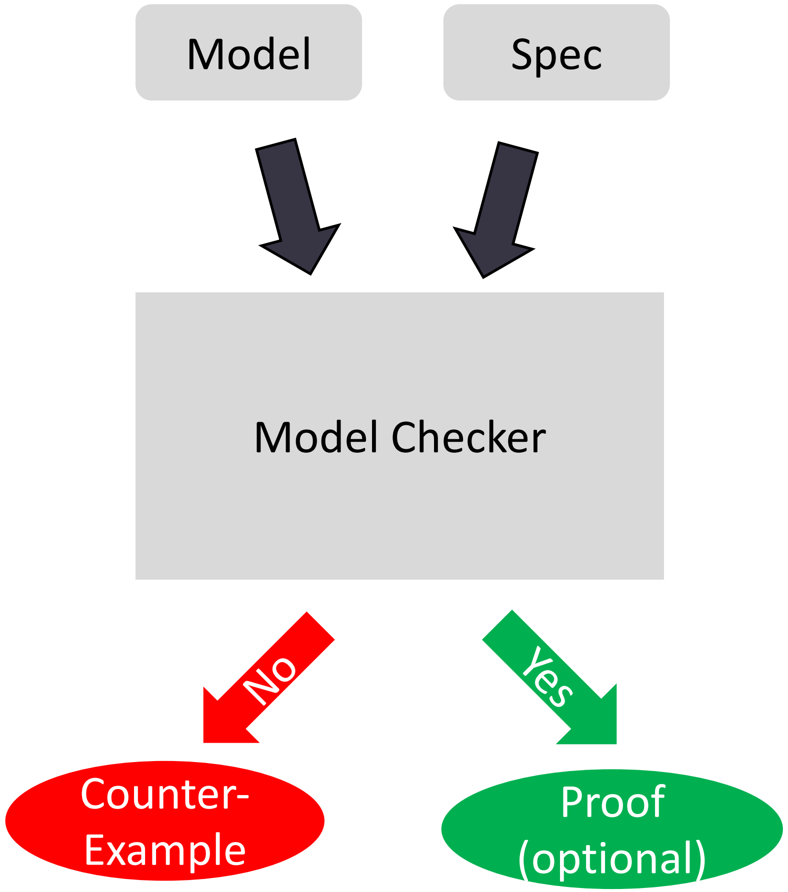
Outline

- What is Model Checking?
  - Modeling: Transition Systems
  - Specification: Linear Temporal Logic
- Historical Verification Approaches
  - Explicit-state
  - BDDs
- SAT/SMT-based Verification Approaches
  - Bounded Model Checking
  - K-Induction
- Inductive Invariants

* Many of the slides today are contributed by [Makai Mann].

What is Model Checking?

- Approach for verifying the temporal behavior of a system
- **Model:** Representation of the system
- **Specification:** High-level desired property of system
- Considers infinite sequences



Modeling: Transition System

- Model checking typically operates over *Transition Systems*
  - A (symbolic) state machine

- A Transition System is $\langle S, I, T \rangle$
  - S: a set of states
  - I: a set of initial states (sometimes use $Init$ instead of I for clarity)
  - T: a transition relation: $T \subseteq S \times S$
    - $T(s_0, s_1)$ holds when there is a transition from s_0 to s_1

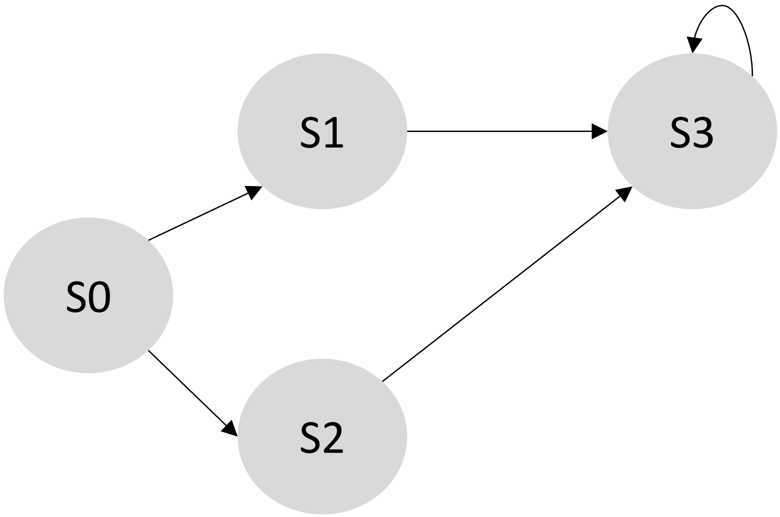
Symbolic Transition Systems in Practice

- States are made up of state variables $v \in V$
  - A state is an assignment to all variables
- A Transition System is $\langle V, I, T \rangle$
  - V: a set of state variables, V' denotes next state variables
  - I: a set of initial states
  - T: a transition relation
    - $T(v_0, \dots, v_n, v_0', \dots, v_n')$ holds when there is a transition
    - Note: will often still use s to denote symbolic states (just know they're made up of variables)
- Symbolic state machine is built by translating another representation
  - E.g. a program, a mathematical model, a hardware description, etc…

Symbolic Transition System Example

- 2 variables: $V = \{v_0, v_1\}$
  - $S_0 \coloneqq \neg v_0 \wedge \neg v_1,\ \ S_1 \coloneqq \neg v_0 \wedge v_1$
  - $S_2 \coloneqq v_0 \wedge \neg v_1,\quad S_3 \coloneqq v_0 \wedge v_1$
- Transition relation

$$(\neg v_0 \wedge \neg v_1) \Rightarrow \big((\neg v_0' \wedge v_1') \vee (v_0' \wedge \neg v_1')\big) \wedge$$
$$(\neg v_0 \wedge v_1) \Rightarrow (v_0' \wedge v_1') \wedge$$
$$(v_0 \wedge \neg v_1) \Rightarrow (v_0' \wedge v_1') \wedge$$
$$(v_0 \wedge v_1) \Rightarrow (v_0' \wedge v_1')$$



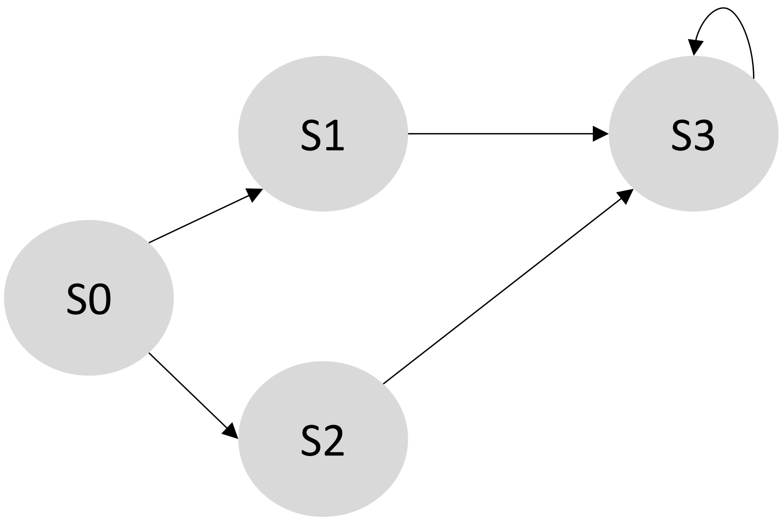
Modeling: Transition System Executions

- An *execution* is a sequence of states that respects I in the first state and T between every adjacent pair

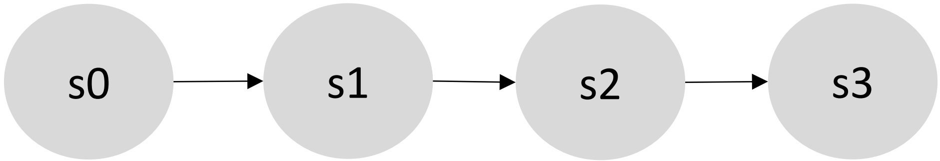
- $\pi \coloneqq s_0\ s_1\ \dots s_n$ is a finite sequence if $I(s_0) \wedge \bigwedge_{i=1}^{n} T(s_{i-1}, s_i)$

Meta Note: State Machine vs Execution Diagrams

State Machine uses capitals



Symbolic execution uses lowercase



Concrete Execution:

s0=S0, s1=S2, s2=S3, s3=S3

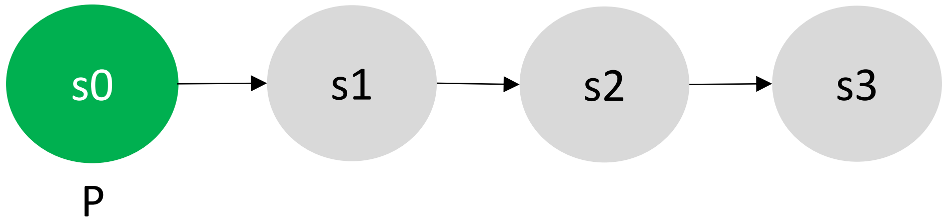
Specification: Linear Temporal Logic (LTL)

- Notation: $M \vDash f$
  - **Transition system** model, M, entails **LTL property**, f, for **ALL possible paths**
  - i.e. LTL is implicitly universally quantified

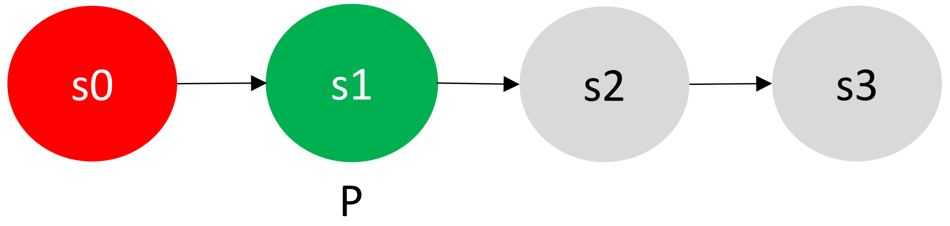
- Other logics include
  - CTL: computational tree logic (branching time)
  - CTL*: combination of LTL and CTL
  - MTL: metric temporal logic (for regions of time)

Specification: Linear Temporal Logic (LTL)

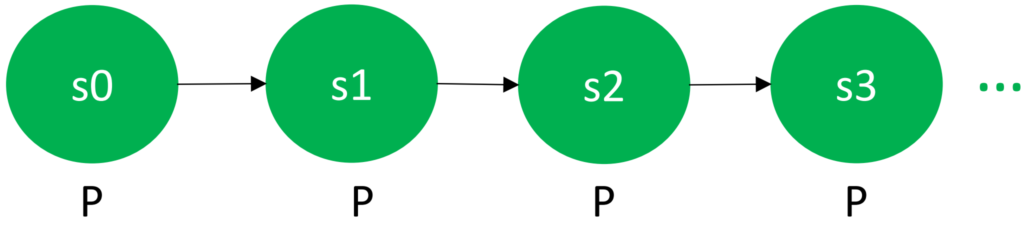
- Atomic state property $P \subseteq S$:
  - Holds iff $s_0 \in P$



- **Next** P: $X(P)$
  - P holds **Next** time
  - Also written ○p
  - True iff the next state meets property P

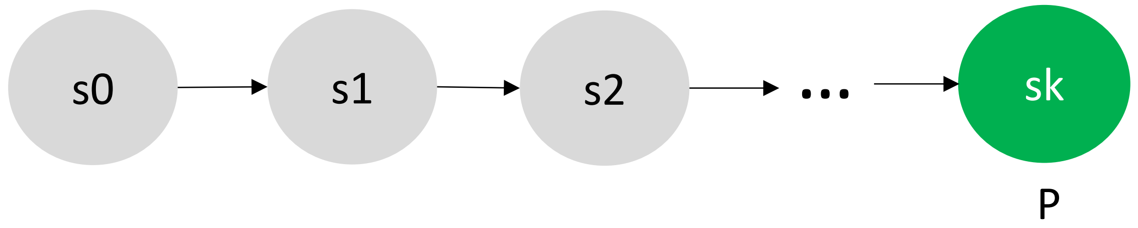


- **Invariant** P: G(P)
  - P **Globally** holds
  - Also written □p
  - True iff every reachable state meets property P

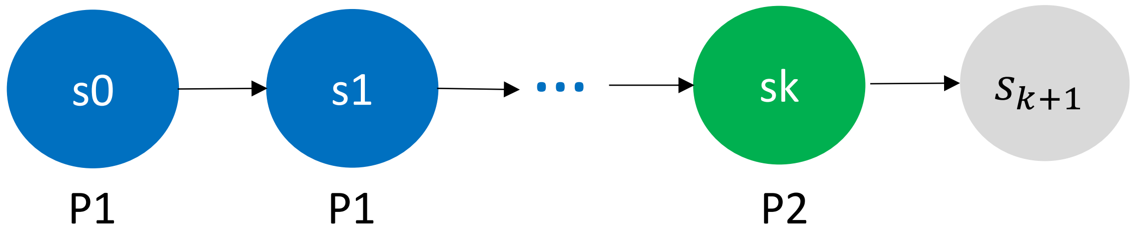


Specification: Linear Temporal Logic

- **Eventually** P: F(P)
  - P holds in the **Future**
  - Also written ◇p
  - True iff P eventually holds



- P1 **Until** P2: P1 U P2
  - P1 holds until P2 holds
  - True iff P1 holds up until (but not necessarily including) a state where P2 holds
  - P2 must hold at some point



Specification: Linear Temporal Logic

- LTL operators can be composed
  - $G\big(Req \Rightarrow F(Ack)\big)$
    - Every request eventually acknowledged
  - $G(F(DeviceEnabled))$
    - The device is enabled infinitely often (from every state, it's eventually enabled again)
  - $F(G(\neg Initializing))$
    - Eventually it's not initializing
    - E.g. there is some initialization procedure that eventually ends and never restarts

Specification: Safety vs. Liveness

- Safety: “something bad does not happen”
  - State invariant, e.g. $G(\neg bad)$
- Liveness: “something good eventually happens”
  - Eventuality, e.g. $\mathrm{G}F(good)$
- Fairness conditions
  - Fair traces satisfy each of the fairness conditions infinitely often
  - E.g. only fair if it doesn’t delay acknowledging a request forever
- Every property can be written as a conjunction of a safety and liveness property

Bowen Alpern and Fred B. Schneider. Defining liveness. Information Processing Letters, 21(4):181–185, October 1985.

Specification: Liveness to Safety

- Can reduce liveness to safety checking
- For SAT-based:

  Armin Biere, Cyrille Artho, Viktor Schuppan. Liveness Checking as Safety Checking, Electronic Notes in Theoretical Computer Science. 2002

- Several approaches for first-order logic

- **From now on, we consider only safety properties**

Historical Verification Approaches: Explicit State

- Tableaux-style state exploration
- Form of depth-first search
- Many clever tricks for reducing search space
- Big contribution is handling temporal logics (including branching time)

Historical Verification Approaches: BDDs

- Binary Decision Diagrams (BDDs)
  - Manipulate sets of states symbolically

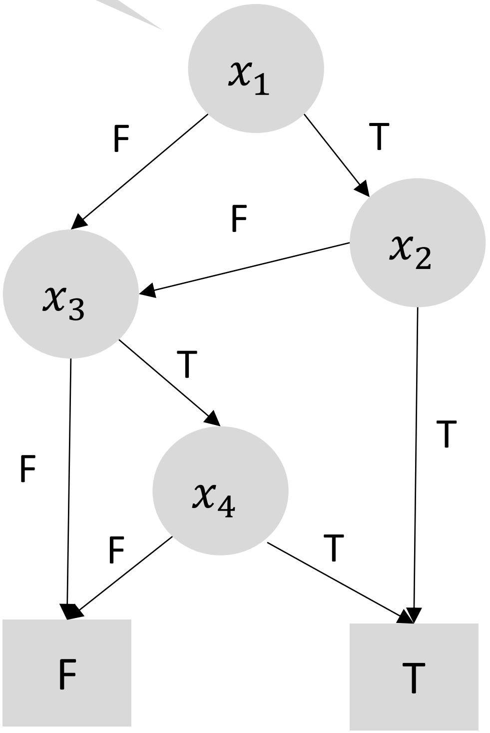
J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang. Symbolic Model Checking: 10^{20} States and beyond

- Great BDD resource: http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/reading/somenzi99bdd.pdf

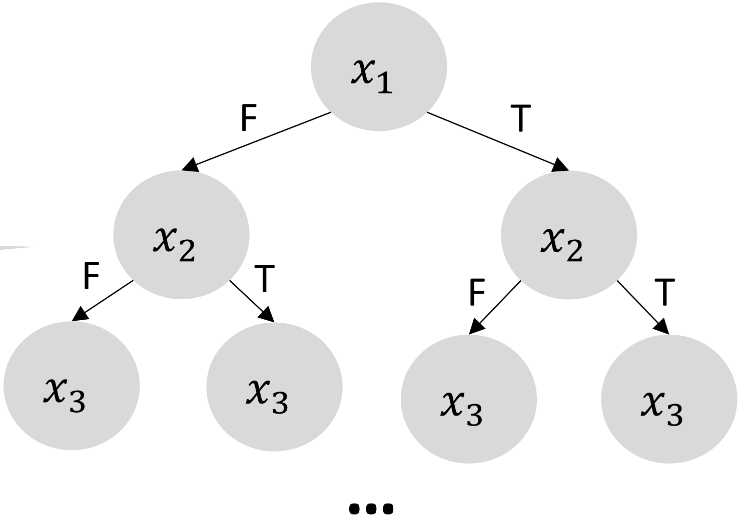
Historical Verification Approaches: BDDs

- Represent Boolean formula as a decision diagram
- Example: $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$
- Can be much more succinct than other representations

Binary Decision Diagram

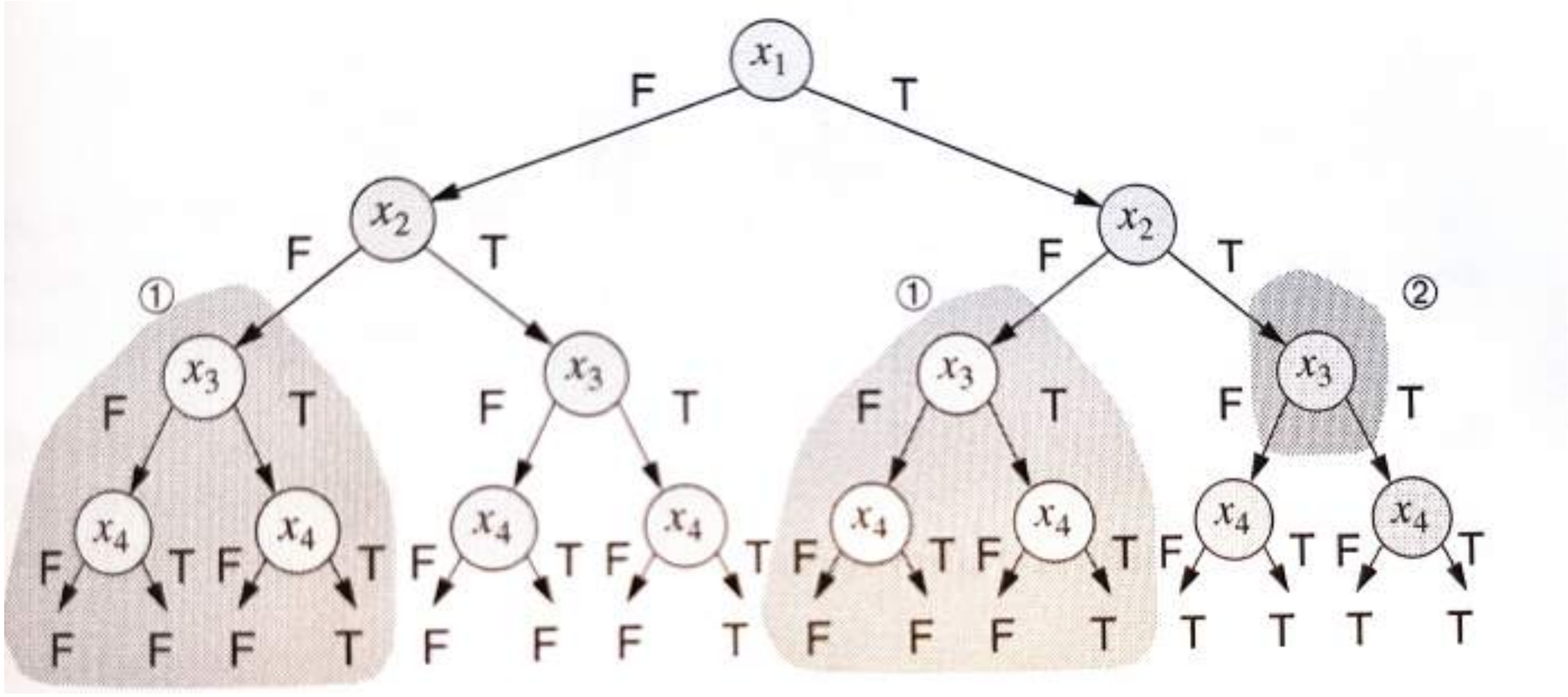


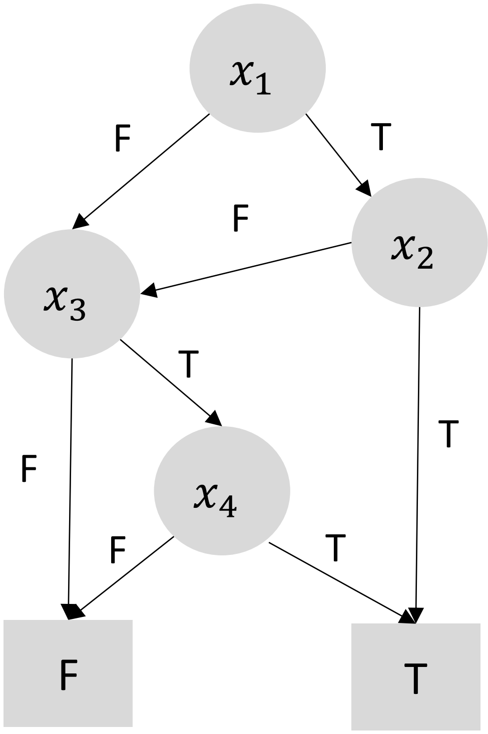
Binary Decision Tree



Credit for Example: <u>Introduction to Formal Hardware Verification</u> – Thomas Kropf

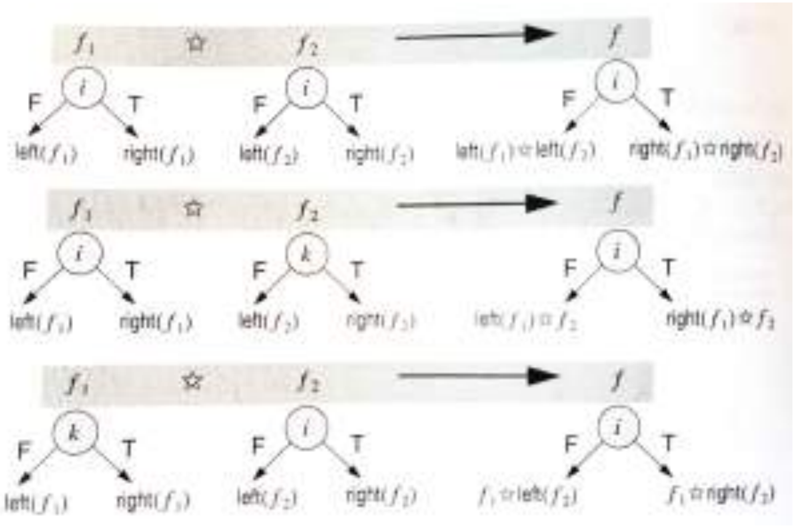
Historical Verification Approaches: BDDs





BDD Operators

- Negation
  - Swap leaves (F → T)
- AND
  - All Boolean operators implemented recursively
- These two operators are sufficient



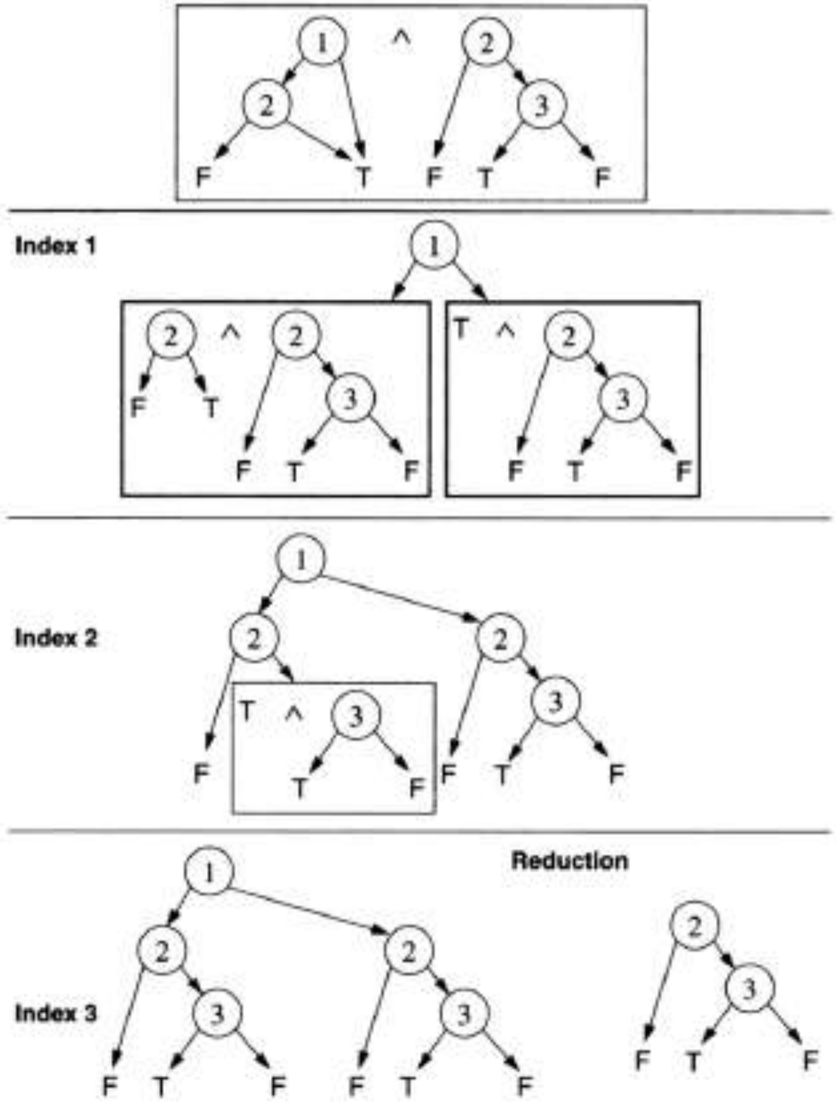


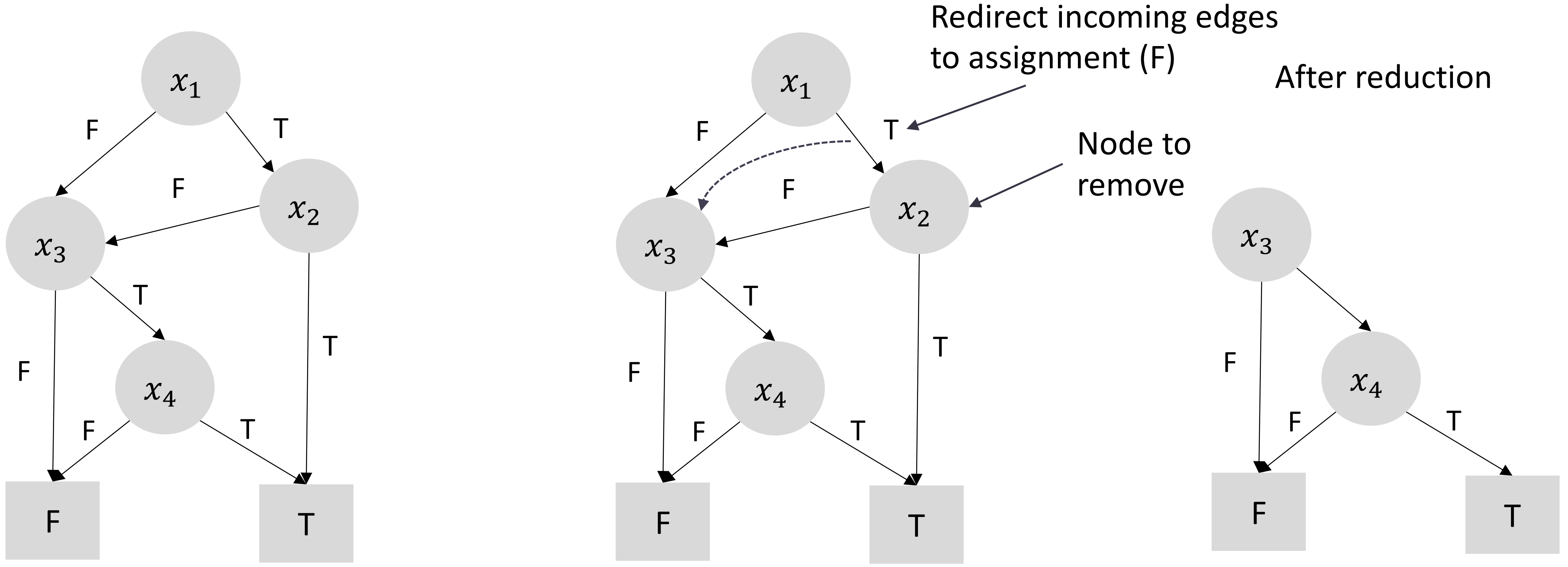
Fig. 2-7. AND-Operation between $x_1 \vee x_2$ and $x_2 \neg x_3$

Image Credit: <u>Introduction to Formal Hardware Verification</u> – Thomas Kropf

BDDs: Cofactoring

$f(x) \coloneqq (x \wedge f\big|_{x}) \vee (\neg x \wedge f\big|_{\neg x})$

- $f|_{\neg x_2}$ for BDD f is fixing x_2 to be negative



Credit for Example: <u>Introduction to Formal Hardware Verification</u> – Thomas Kropf

BDD Image Computation

- Current reachable states are BDD R
  - Over variable set V
- Compute next states with:
  - $N := \exists V \; T(V, V') \land R(V)$
  - Existential is implemented cofactoring: $\exists x_i \,.\, f(\ldots, x_i, \ldots) := f(\ldots, F, \ldots) \lor f(\ldots, T, \ldots)$
- Grow reachable states
  - $R = R \lor N[V'/V]$
  - Map next-state variables to current state, then add to reachable states

BDD image computation is based on the idea that **all reachable next states** are either **already in R** or they are the **result of applying the transition function** to some set of states V in R to reach the set of states V'.

$T(V, V') \land R(V)$ using BDD operations. Then, use cofactoring operation to remove (non-next state) state-variables.

T, R, and N are all BDDs

Convert next state variables V' to state variables V

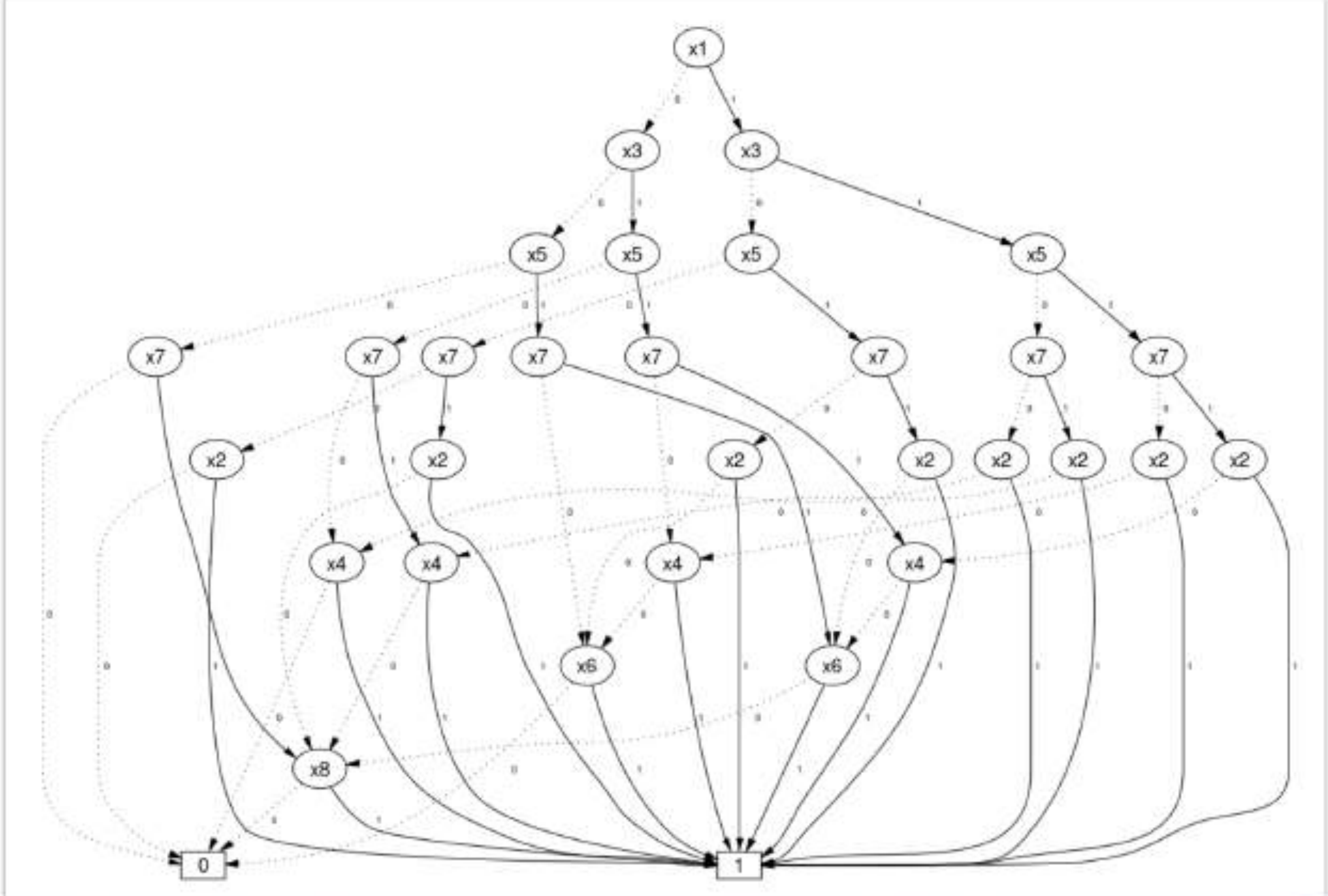
BDD-based model checking

- Start with $R = Init$
- Keep computing image and growing reachable states
- Stop when there's a fixpoint (reachable states not growing)
- Can handle ~10^{20} states
    - More with abstraction techniques and compositional model checking

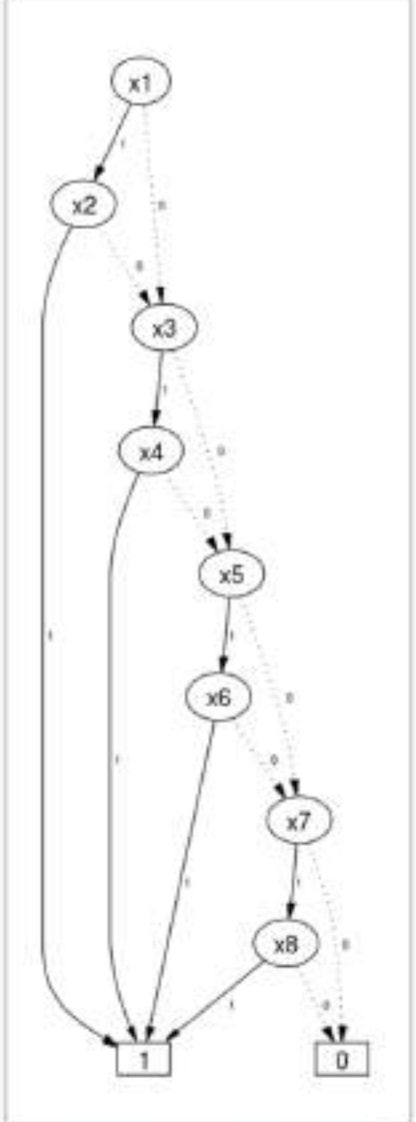
BDD: Variable Ordering

- Good variable orderings can be exponentially more compact
  - Finding a good ordering is NP-complete

- There are formulas that have no non-exponential ordering



BDD for the function $f(x_1, \ldots, x_8) = x_1x_2 + x_3x_4 + x_5x_6 + x_7x_8$ using bad variable ordering



Good variable ordering

Image Credit: https://en.wikipedia.org/wiki/Binary_decision_diagram

SAT-based model checking

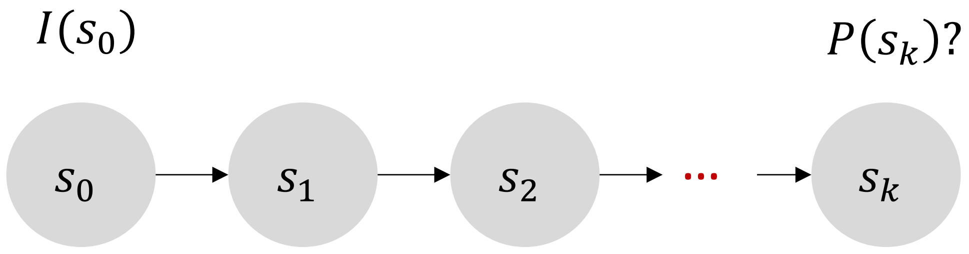
- Edmund Clarke
  - One of the founders of model checking
- SAT solving taking off
- Clarke hired several post-doctoral students to try to use SAT as an oracle to solve model checking problems
- Struggled for a while to find a general technique
  - What if you give up completeness? → Bounded Model Checking

Armin Biere, Alessandro Cimatti, Edmund Clarke, Yunshan Zhu. Symbolic Model Checking without BDDs. TACAS 1999

Bounded Model Checking (BMC)

- Sacrifice completeness for quick bug-finding
- Unroll the transition system
  - Each variable $v \in V$ gets a new symbol for each time-step, e.g. v_k is v at time k
  - Space-Time duality: unrolls temporal behavior into space
- For increasing values of k, check:
  - $I(s_0) \land \bigwedge_{i=1}^{k} T(s_{i-1}, s_i) \land \neg P(s_k)$
- If it is ever SAT, return FALSE
  - Can construct a counter-example trace

BMC Graphically



s_0 must be an initial state

Check if it can violate the property at time k

Bounded Model Checking: Completeness

- Completeness condition: reaching the diameter
  - Diameter: d
    - Depth needed to unroll to such that every possible state is reachable in d steps or less

$$rd(M) := \min\{i \mid \forall s_0, \ldots, s_{i+1}.\ \exists s'_0, \ldots, s'_i. \\ I(s_0) \wedge \bigwedge_{j=0}^{i} T(s_j, s_{j+1}) \rightarrow (I(s'_0) \wedge \bigwedge_{j=0}^{i-1} T(s'_j, s'_{j+1}) \wedge \bigvee_{j=0}^{i} s'_j = s_{i+1})\} \tag{3}$$

  - Recurrence diameter: d_r
    - The depth such that *every* execution of the system of length $\geq d_r$ *must* revisit states
    - Can be exponentially larger than the diameter

$$rdr(M) := max\{i \mid \exists s_0 \ldots s_i.\ I(s_0) \wedge \bigwedge_{j=0}^{i-1} T(s_j, s_{j+1}) \wedge \bigwedge_{j=0}^{i-1} \bigwedge_{k=j+1}^{i} s_j \neq s_k\} \tag{4}$$

  - $d_r \geq d$

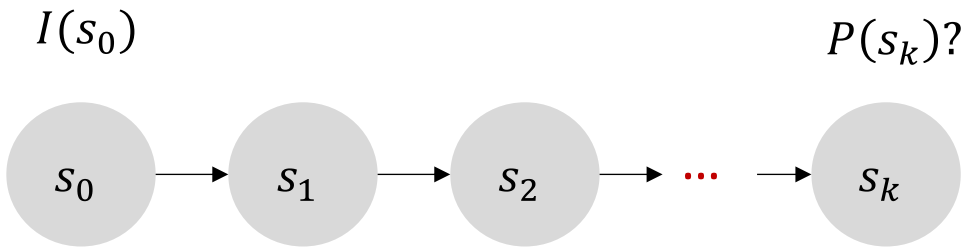
- Very difficult to compute the diameter
  - Requires a quantifier: find d such that any state reachable at $d + 1$ is also reachable in $\leq d$ steps (replace “i” with “d” in equation (3) above)

K-Induction

- Extends bounded model checking to be able to prove properties
- Based on the concept of (strong) mathematical induction
- For increasing values of k, check:
  - Base Case: $I(s_0) \wedge \bigwedge_{i=1}^{k} T(s_{i-1}, s_i) \wedge \neg P(s_k)$
  - Inductive Case: $\left(\bigwedge_{i=1}^{k+1} T(s_{i-1}, s_i) \wedge P(s_{i-1})\right) \wedge \neg P(s_{k+1})$
  - If base case is SAT, return a counter-example
  - If inductive case is UNSAT, return TRUE
  - Otherwise, continue

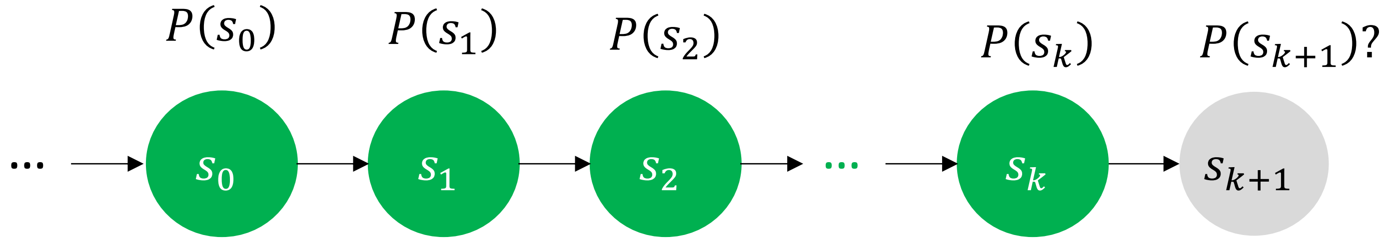
Mary Sheeran, Satnam Singh, and Gunnar Stälmarck. Checking safety properties using induction and a SAT-solver. FMCAD 2000

K-Induction Graphically



Base Case

s_0 must be an initial state

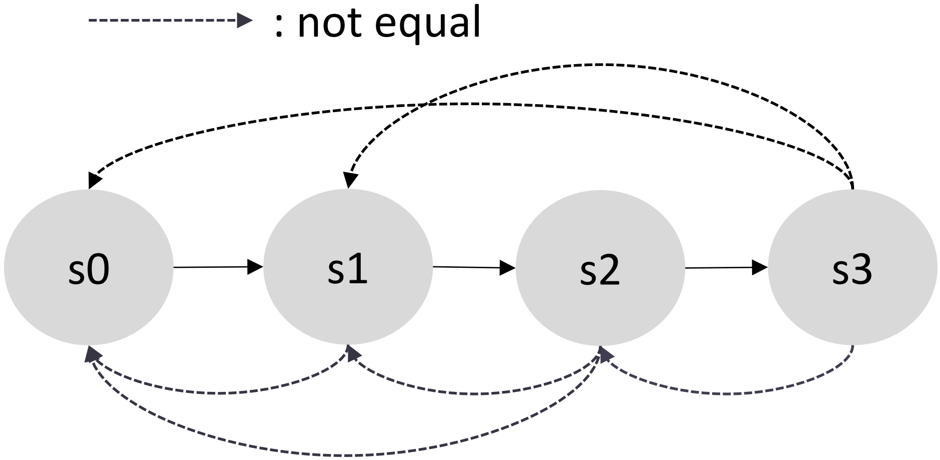


Inductive Case

Arbitrary starting state s_0
such that $P(s_0)$ holds

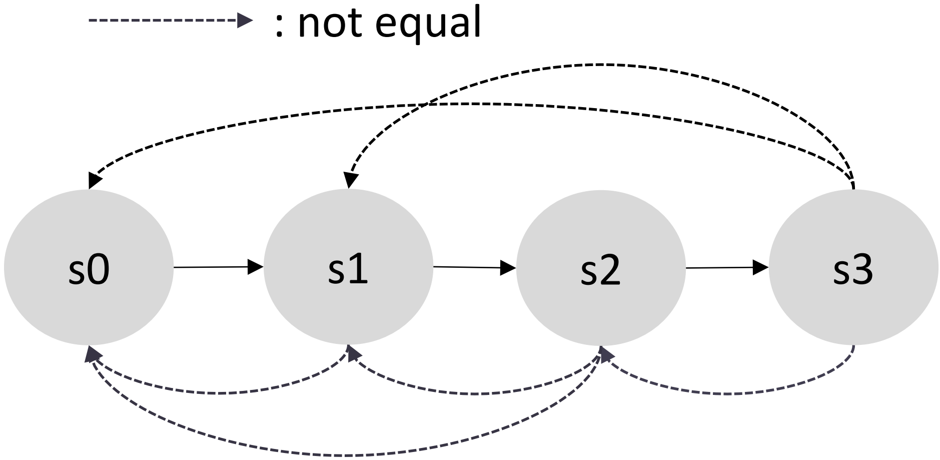
K-Induction: Simple Path

- This approach can be complete over a finite domain
  - requires the simple path constraint
  - each state is distinct from other states in trace
- If simple path is UNSAT, then we can return true

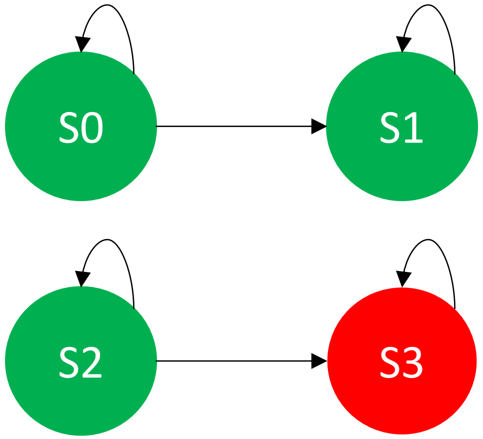


K-Induction: Simple Path

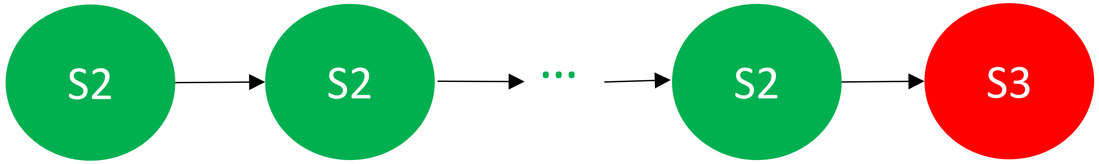
- This approach can be complete over a finite domain
    - requires the simple path constraint
    - each state is distinct from other states in trace
- If simple path is UNSAT, then we can return true



Why?



Without simple path, inductive step could get:

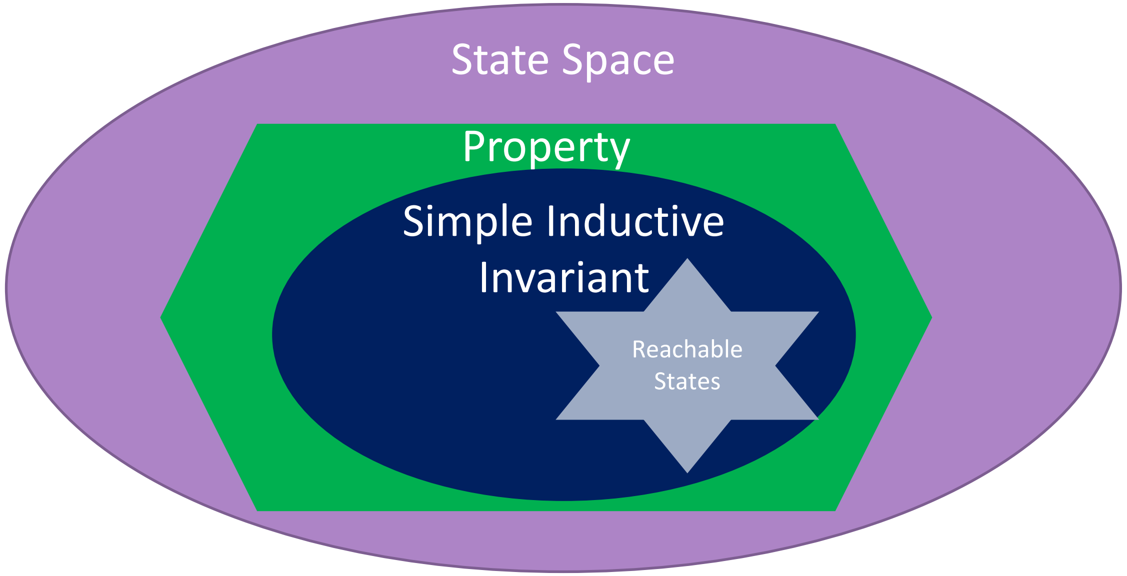


K-Induction Observation

- Crucial observation
  - Does not depend on direct computation of reachable state space
- Beginning of “property directed” techniques
  - We do not need to know the exact reachable states, as long as we can guarantee they meet the property
  - “Property directed” is associated with a family of techniques that build inductive invariants automatically

Inductive Invariants

- The goal of most modern model checking algorithms
- Over finite-domain, just need to show that algorithm makes progress, and it will eventually find an inductive invariant
  - In the worst case, the reachable states are themselves an inductive invariant
  - Hopefully there's an easier to find inductive invariant that is sufficient
- Inductive Invariant: II
  - $Init(s) \Rightarrow II(s)$
  - $\mathrm{T}(s, s') \wedge II(s) \Rightarrow II(s')$
  - $II(s) \Rightarrow P(s)$



Advanced Algorithms

- Interpolant-based model checking
  - Constructs an over-approximation of the reachable states
  - Terminates when it finds an inductive invariant or a counterexample
- IC3 / PDR
  - Computes over (under) approximations of forward (backward) reachable states
  - Refines approximations by guessing relative inductive invariants
  - Terminates when it finds an inductive invariant or a counterexample

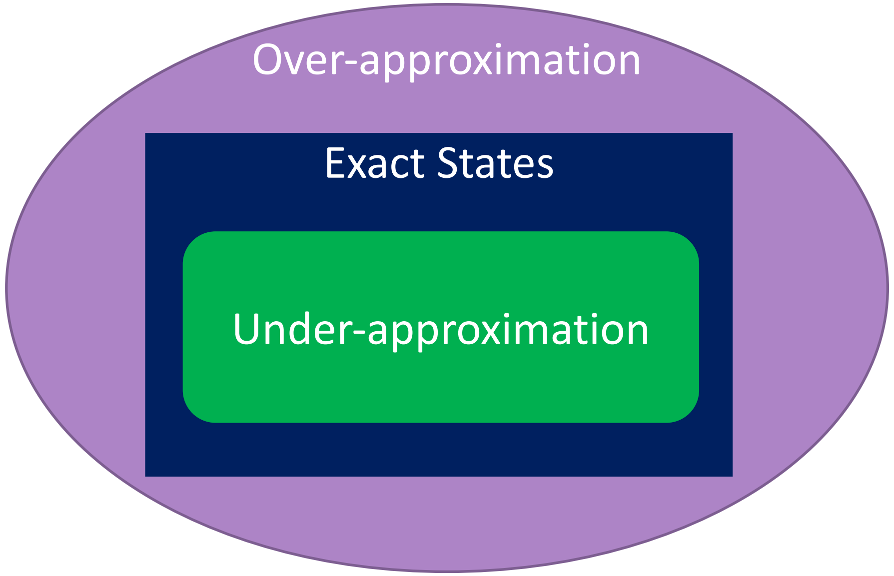
Building Blocks: Approximations

- Problems
  - Explicit reachability computation (e.g. BDDs) is difficult
  - Inductive invariants are difficult to find

- Solution (motivation for approximations)
  - Build approximations of reachable states
  - Iteratively refine it until inductive

What is an approximation?

- Actual reachable state set: R
- Over-approximation, $O: R \rightarrow O$
  - Proofs on over-approximation holds
  - Counterexamples can be spurious
- Under-approximation, $U: U \rightarrow R$
  - Proofs on under-approximation can be spurious
  - Counterexamples are real



Craig Interpolation

- Given an unsatisfiable formula, $A \land B$

- Craig Interpolant, I
  - $A \rightarrow I$
  - $I \land B$ is UNSAT
  - $V(I) \subseteq V(A) \cap V(B)$
    - Where V returns the free variables (uninterpreted constants) of a formula

- We can use interpolants as over-approximations of A

Obtaining Craig Interpolants

- Mechanical over SAT
  - Label clauses in the proof
  - Some straightforward post-processing
- Non-trivial for SMT
  - But there are solvers that support it
    - MathSAT
    - Smt-Interpol
    - CVC4 – through SyGuS

K. L. McMillan, Interpolation and SAT-based Model Checking, CAV 2003

Interpolant-based Model Checking

- Big picture
  - Perform BMC
  - Iteratively compute and refine an over-approximation of states reachable in k steps
  - If it becomes inductive, you’re done

Interpolants for Abstraction from BMC Run

- Obtain interpolant, I, from an unsat BMC run with A and B as shown below
- Useful properties
  - I over-approximates A, i.e. states reachable in one-step from Init: $A \rightarrow I$
  - There are no states reachable in $k-1$ steps from I that violate the property: $I \wedge B$ UNSAT
  - I only contains symbols from one time step (time 1): $V(I) \subseteq V(A) \cap V(B)$

$$\underbrace{Init \wedge T(s_0, s_1)}_{\text{A}} \qquad \underbrace{T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k)}_{\text{B}}$$

From UNSAT $A \wedge B$, Craig Interpolant, I:

$A \rightarrow I$

$I \wedge B$ is UNSAT

$V(I) \subseteq V(A) \cap V(B)$

Interpolant-based Model Checking

```
if check(Init ∧ T(s_0, s_1) ∧ (¬P(s_0) ∨ ¬P(s_1)))
    return False
k=2

R = Init
while True
    A := R ∧ T(s_0, s_1),  B := ¬P(s_k) ∧ ⋀_{i=1}^{k-1} T(s_i, s_{i+1})
    if check(A ∧ B)
        if R == Init
            return False
        else
            R = Init
            k++
    else
        I = get_interpolant()
        R = R ∨ I[1/0] // map symbols at 1 to symbols at 0
        if ¬check(R ∧ T(s_0, s_1) ∧ ¬R)
            return True
```

Base case: Check if s_0 or s_1 violate P

Initialize R to the initial states.

B = Represents a violation of the property P in K-1 steps from the states represented by A.

A = set of states reachable in 1 step from R.

Check to see if P is violated is K steps from R.

If it is and R = Init, return false. True counterexample.

Otherwise, increment, reset R to Init and restart. We may have found a spurious counterexample.

If A and B is UNSAT, we find an **interpolant I.** Recall that I over-approximates A, i.e. states reachable in one-step from R: $\boldsymbol{A \rightarrow I}$**.** Also, there are no states reachable in $k-1$ steps from I that violate the property: $\boldsymbol{I \wedge B}$ **UNSAT.**

We reached a fixed point where R is not changing. We found an invariant and proved the property.

Check to see if $R \wedge T(s_0, s_1) \rightarrow R$ is valid. I.e., check to see if $R \wedge T(s_0, s_1) \wedge \neg R$ is SAT. If UNSAT, the validity check holds which means the transition function will not grow R.

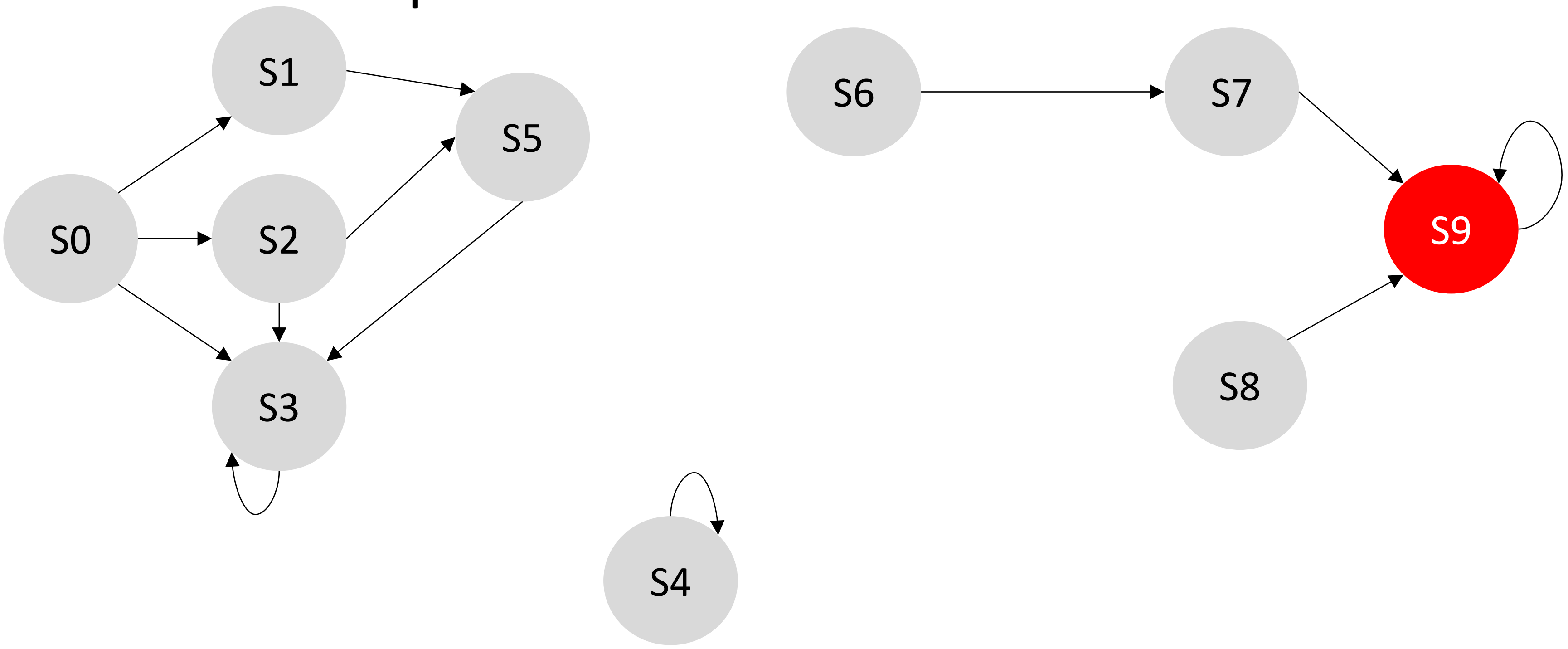
Interpolant-based Model Checking Example

```
if check(Init ∧ T(s_0, s_1) ∧ (¬P(s_0) ∨ ¬P(s_1)))
    return False
```

- Check to see if initial states or states reachable in 1 step violate P

Init: S0

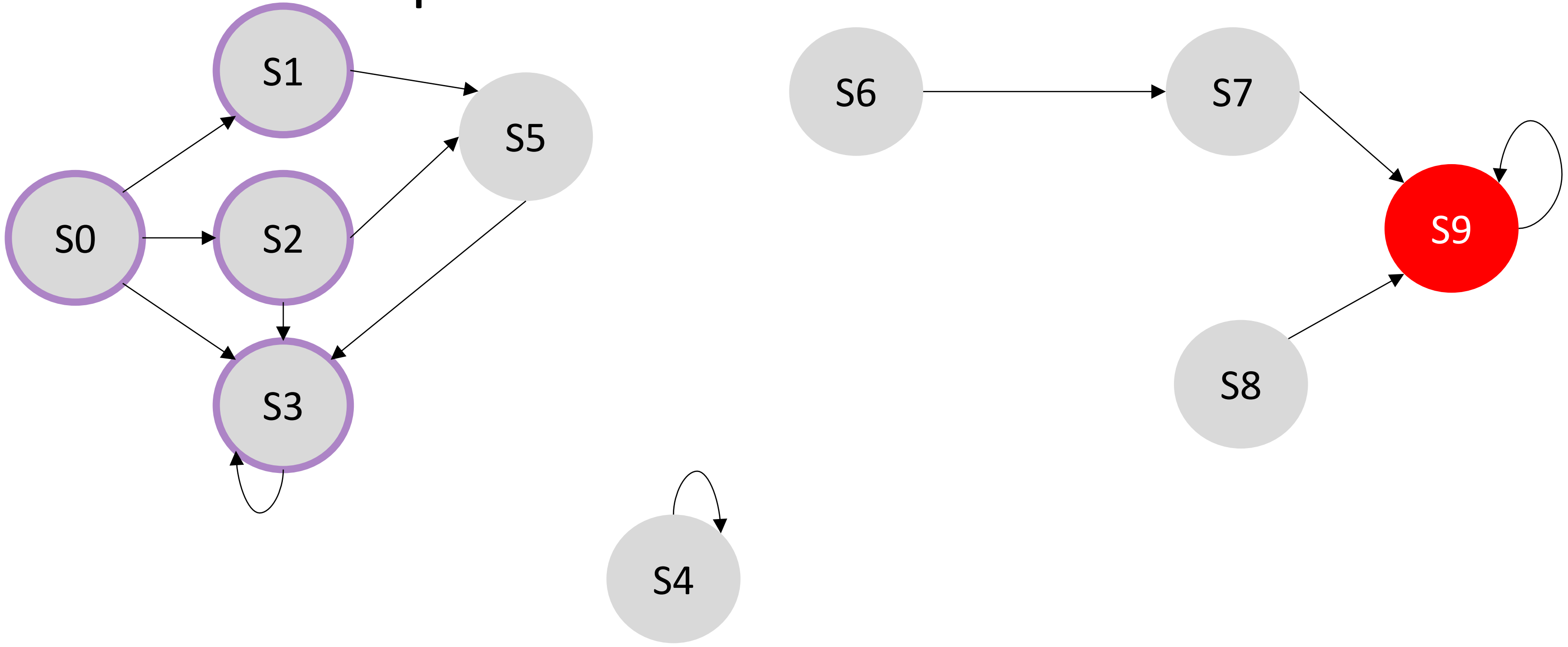
Bad: P = $\neg S9$



Interpolant-based Model Checking Example

```
if check(Init ∧ T(s0, s1) ∧ (¬P(s0) ∨ ¬P(s1))
    return False
```

- Check to see if initial states or states reachable in 1 step violate P



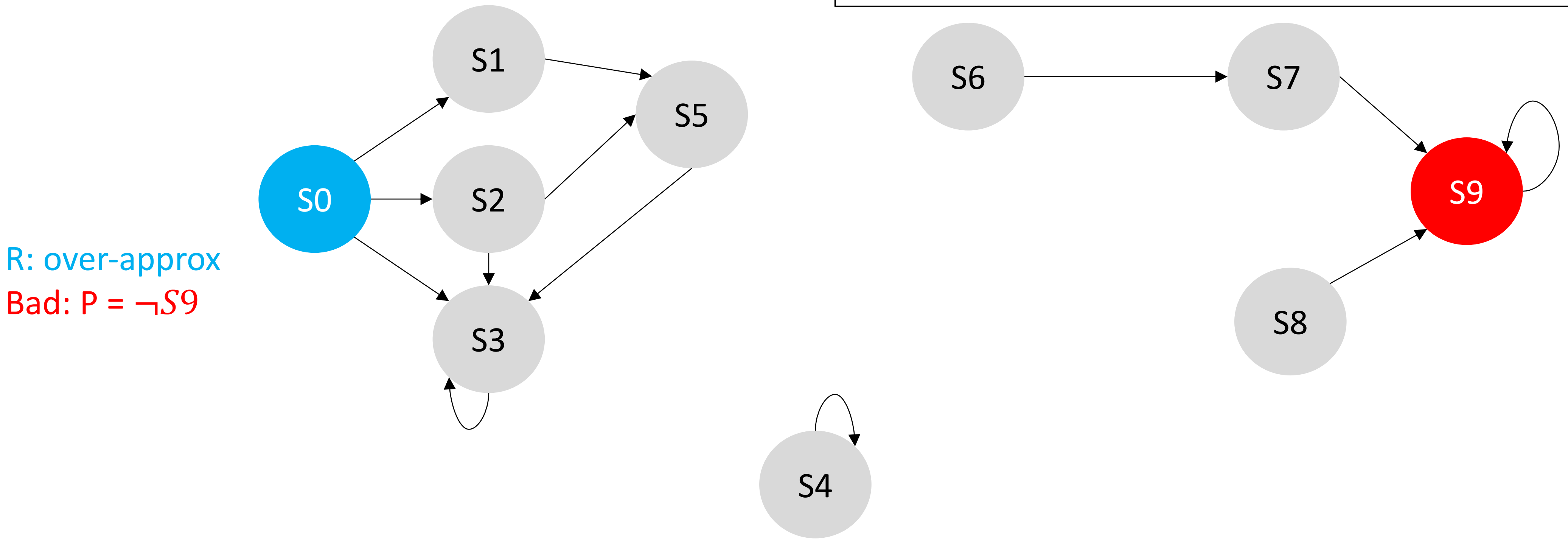
Init: S0

Bad: P = $\neg S9$

Interpolant-based Model Checking Example

- k = 2

$k = 2;\ R = Init$
```
while True
```
$\quad A := R \wedge T(s_0, s_1), B := \neg P(s_k) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$
$\quad$ if check$(A \wedge B)$

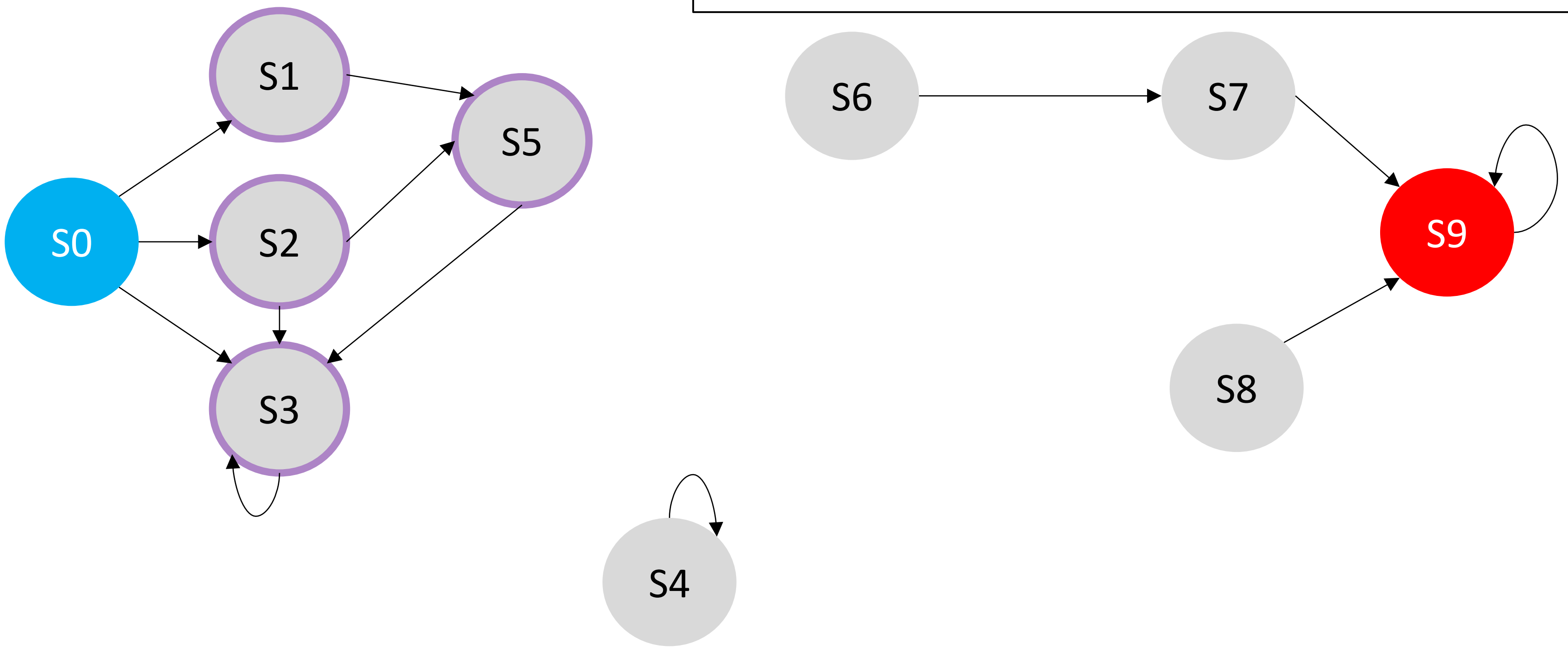


Interpolant-based Model Checking Example

- Start – can't violate in 2 steps

$R = Init$
`while True`
$\quad A := R \wedge T(s_0, s_1), B := \neg P(s_k) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$
$\quad$ `if check(`$A \wedge B$`)`

R: over-approx
Bad: P = $\neg S9$



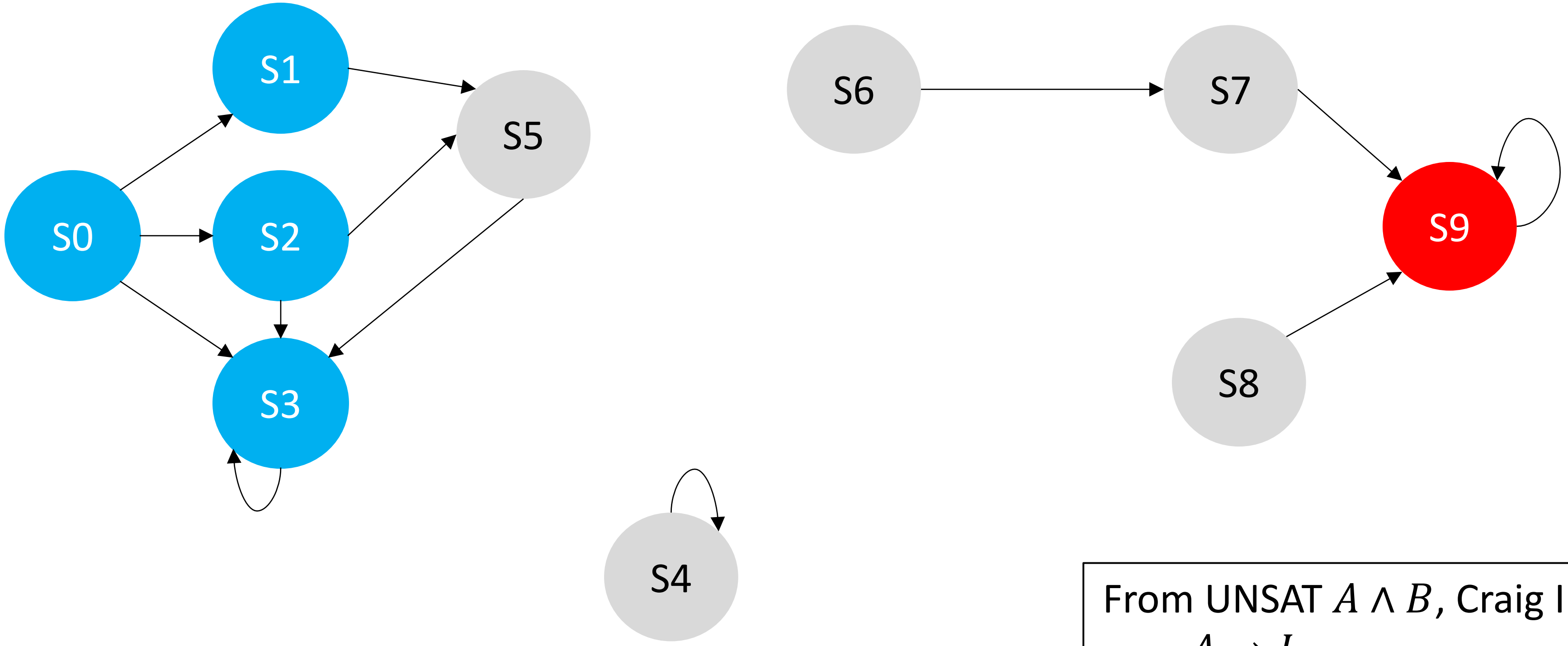
Interpolant-based Model Checking Example

- k = 2

```
I = get_interpolant()
        R = R ∨ I[1/0] // map symbols at 1 to symbols at 0
        if ¬check(R ∧ T(s0, s1) ∧ ¬R)
            return True
```

R: over-approx

Bad: P = $\neg S9$



From UNSAT $A \wedge B$, Craig Interpolant, I:

$A \rightarrow I$

$I \wedge B$ is UNSAT

$V(I) \subseteq V(A) \cap V(B)$

Interpolant-based Model Checking Example

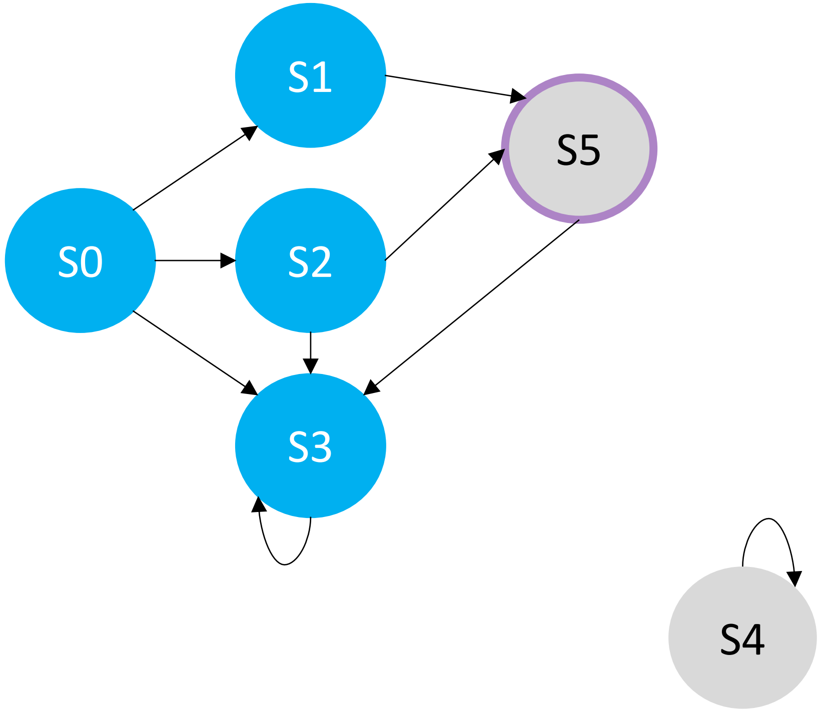
```
while True
```
$A := R \wedge T(s_0, s_1), B := \neg P(s_k) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$

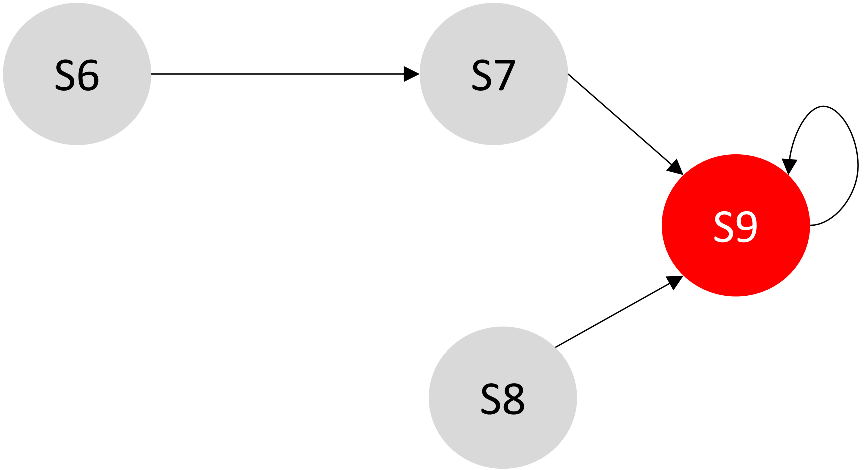
`if check(`$A \wedge B$`)`

- k = 2

R: over-approx

Bad: P = $\neg S9$





From UNSAT $A \wedge B$, Craig Interpolant, I:

$A \rightarrow I$

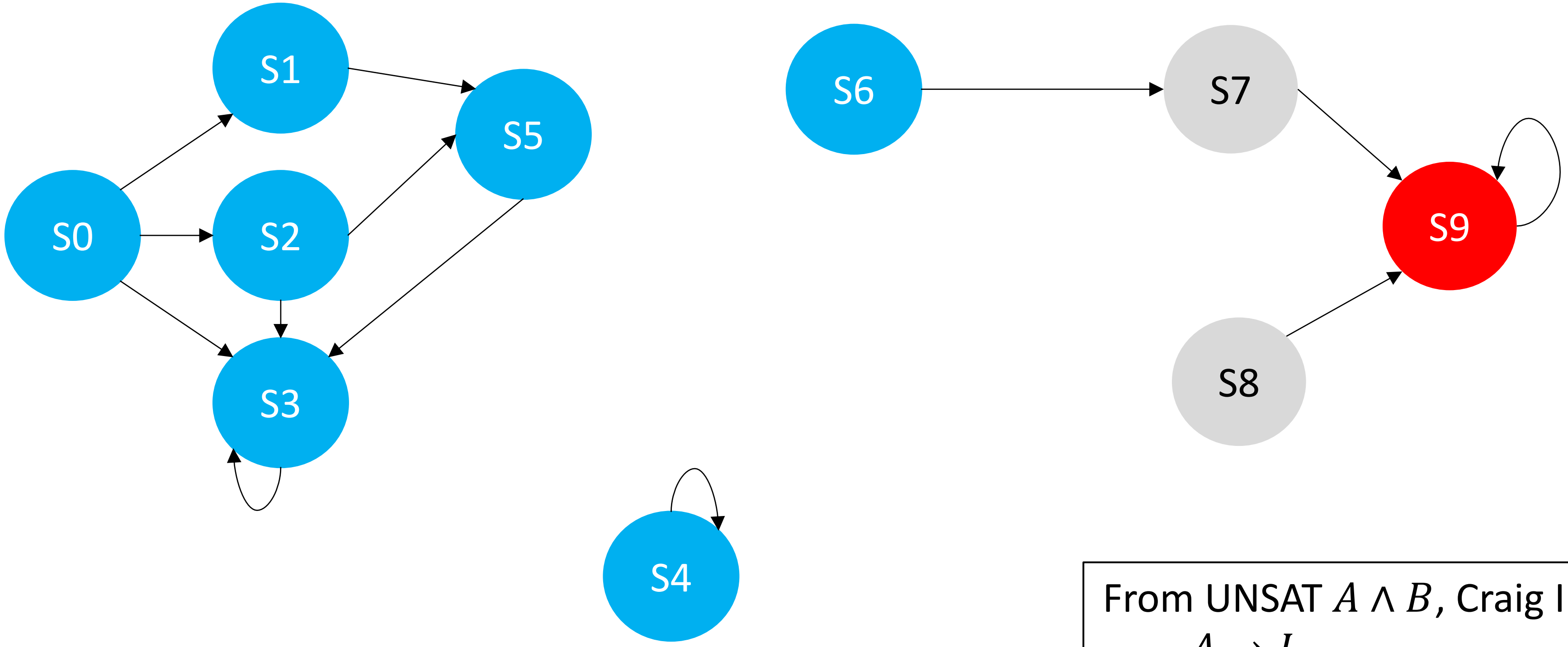
$I \wedge B$ is UNSAT

$V(I) \subseteq V(A) \cap V(B)$

Interpolant-based Model Checking Example

- k = 2

```
I = get_interpolant()
        R = R ∨ I[1/0] // map symbols at 1 to symbols at 0
        if ¬check(R ∧ T(s_0, s_1) ∧ ¬R)
            return True
```



R: over-approx

Bad: P = $\neg S9$

From UNSAT $A \wedge B$, Craig Interpolant, I:

$A \rightarrow I$

$I \wedge B$ is UNSAT

$V(I) \subseteq V(A) \cap V(B)$

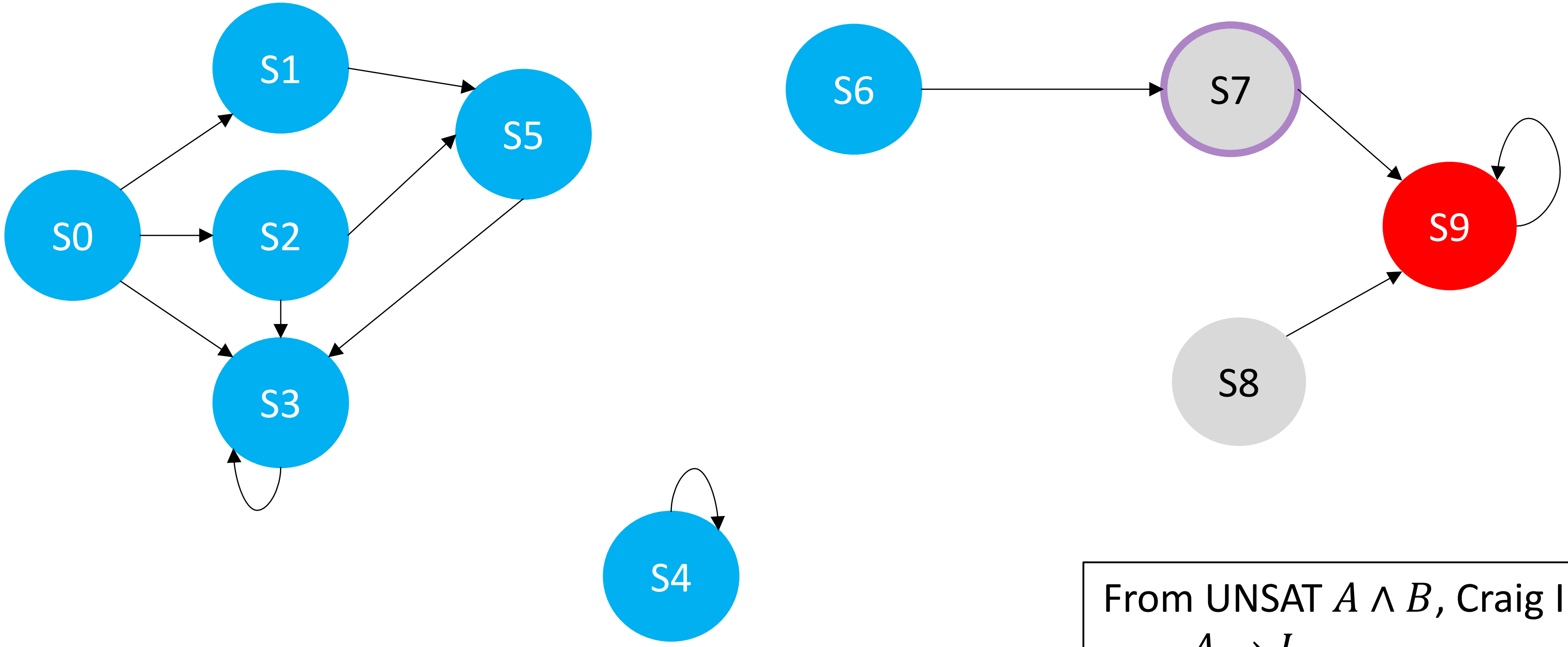
Interpolant-based Model Checking Example

- k = 2

```
I = get_interpolant()
        R = R ∨ I[1/0] // map symbols at 1 to symbols at 0
        if ¬check(R ∧ T(s_0, s_1) ∧ ¬R)
            return True
```

R: over-approx

Bad: P = $\neg S9$



From UNSAT $A \wedge B$, Craig Interpolant, I:

$A \rightarrow I$

$I \wedge B$ is UNSAT

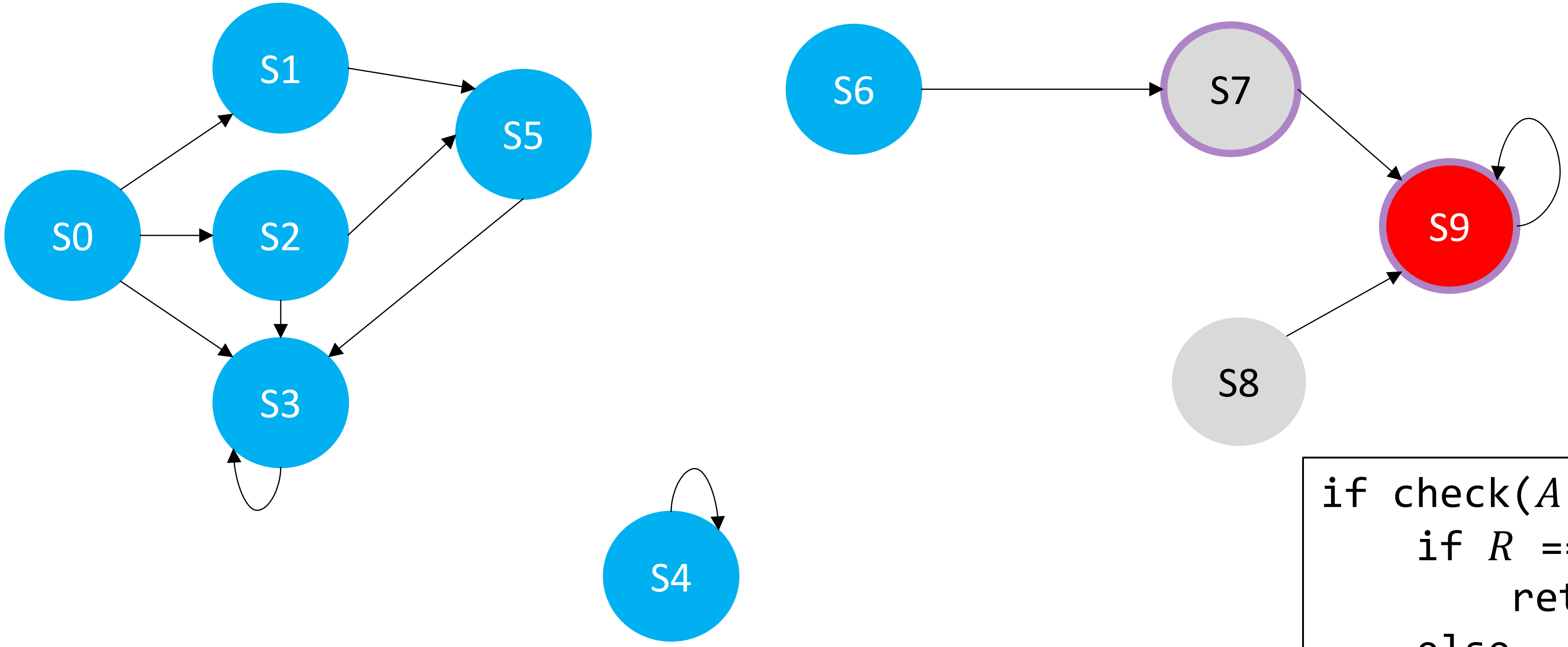
$V(I) \subseteq V(A) \cap V(B)$

Interpolant-based Model Checking Example

- k = 2, can reach S9 in 2 steps from R

R: over-approx

Bad: P = $\neg S9$



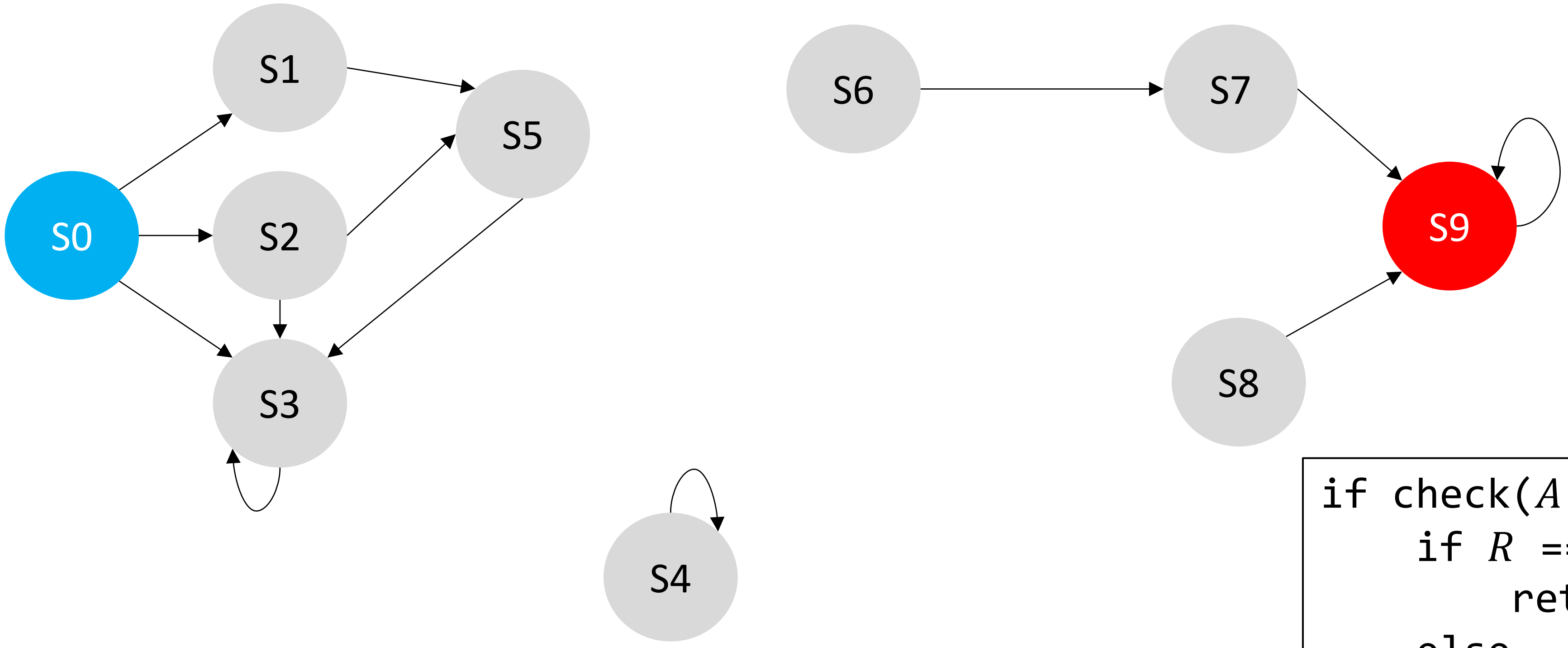
if check($A \wedge B$)
    if R == Init
        return False
    else
        $R = Init$
        k++

Interpolant-based Model Checking Example

- k = 3, restart with R = Init and increment K

R: over-approx

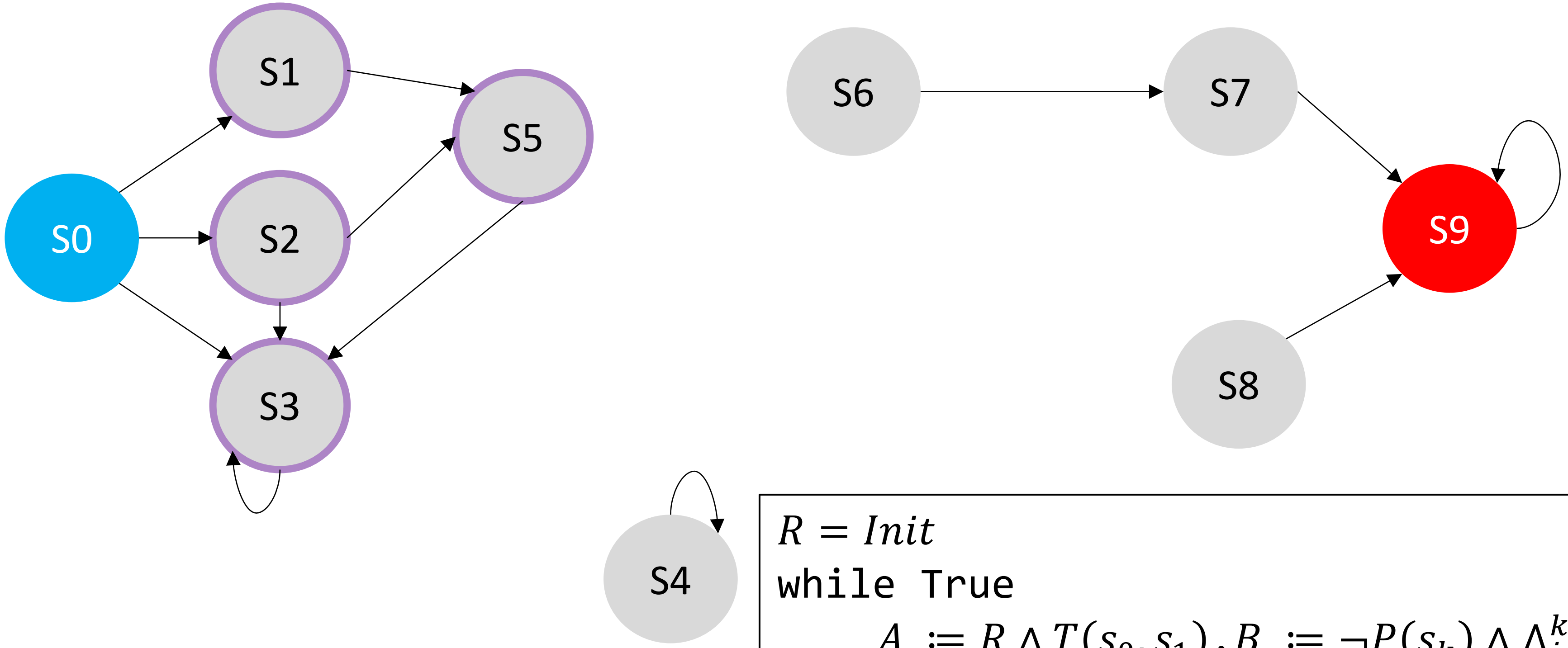
Bad: P = $\neg S9$




```
if check(A ∧ B)
    if R == Init
        return False
    else
        R = Init
        k++
```

Interpolant-based Model Checking Example

- k = 3, restart with R = Init and increment K



R: over-approx

Bad: P = $\neg S9$

$R = Init$

```
while True
```

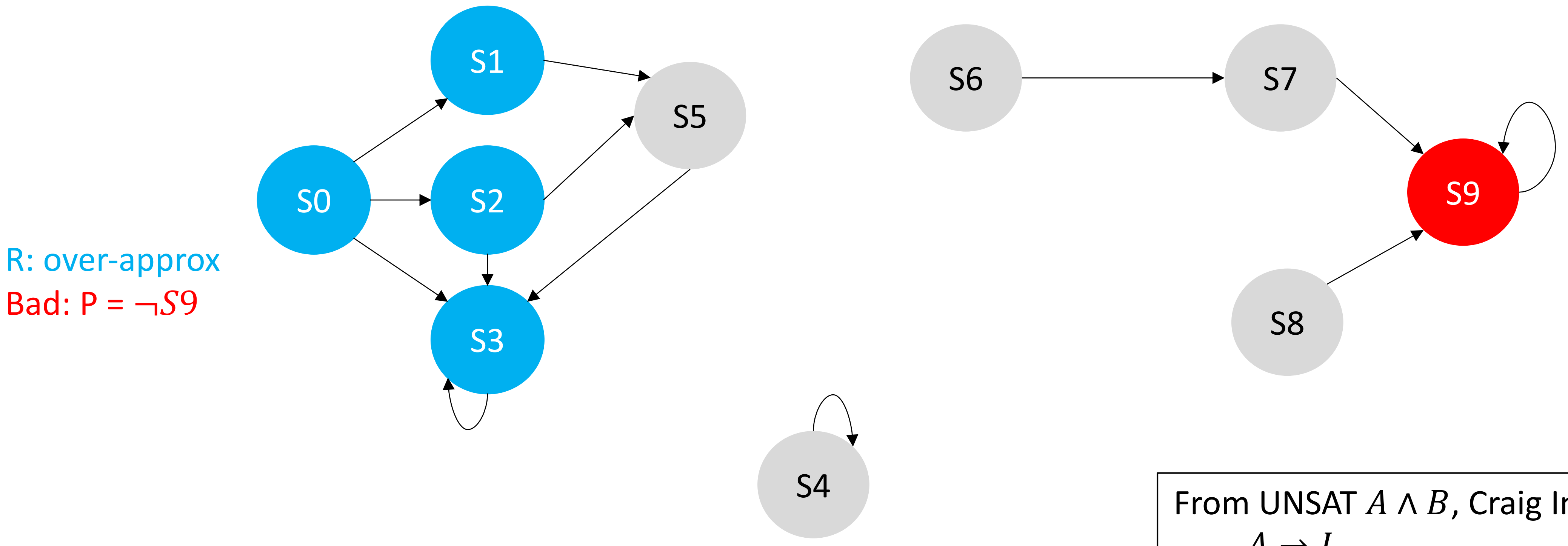
$A \coloneqq R \wedge T(s_0, s_1), B \coloneqq \neg P(s_k) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$

if check($A \wedge B$)

Interpolant-based Model Checking Example

- k = 3

```
I = get_interpolant()
        R = R ∨ I[1/0] // map symbols at 1 to symbols at 0
        if ¬check(R ∧ T(s_0, s_1) ∧ ¬R)
            return True
```



From UNSAT $A \wedge B$, Craig Interpolant, I:

$A \rightarrow I$

$I \wedge B$ is UNSAT

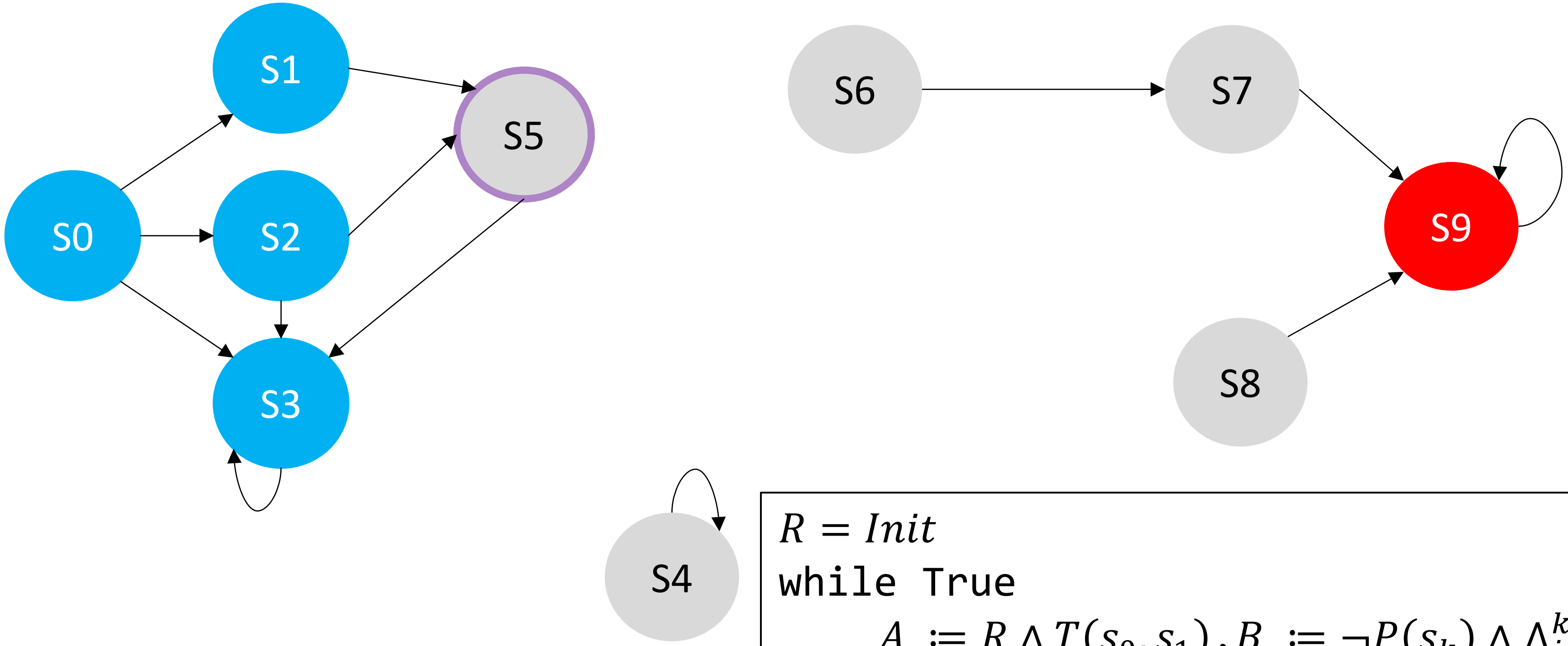
$V(I) \subseteq V(A) \cap V(B)$

Interpolant-based Model Checking Example

- k = 3

R: over-approx

Bad: P = $\neg S9$



$R = Init$

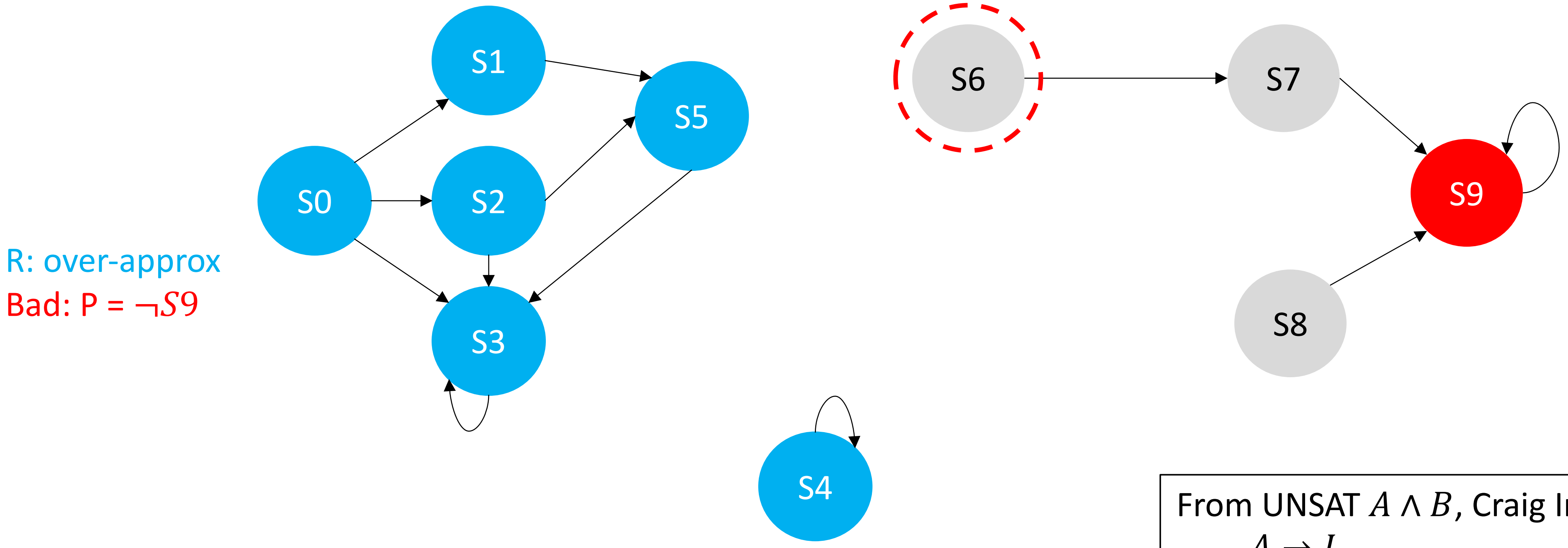
while True

$\quad A \coloneqq R \wedge T(s_0, s_1), B \coloneqq \neg P(s_k) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$

$\quad$ if check$(A \wedge B)$

Interpolant-based Model Checking Example

- k = 3, interpolant guarantees property not violated in k-1 → 2 steps



From UNSAT $A \wedge B$, Craig Interpolant, I:

$A \rightarrow I$

$I \wedge B$ is UNSAT

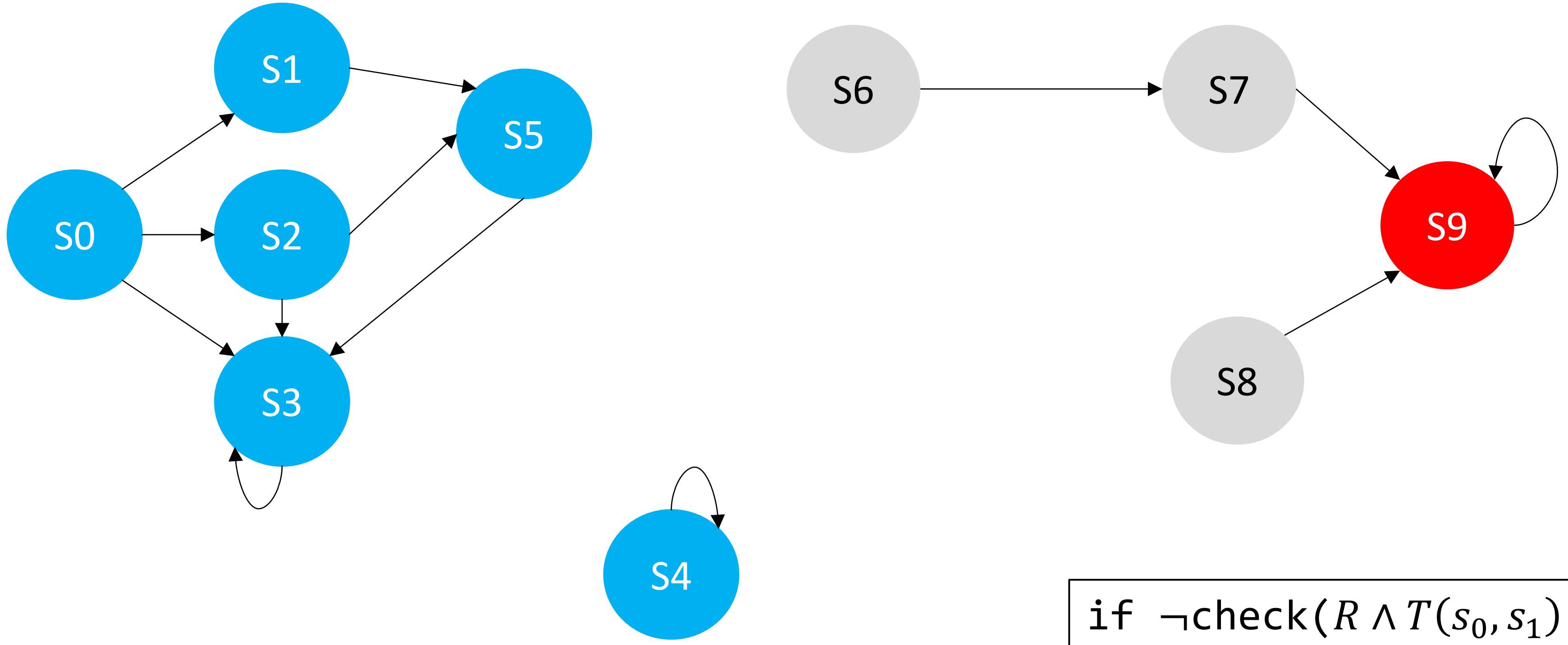
$V(I) \subseteq V(A) \cap V(B)$

Interpolant-based Model Checking Example

- Terminate with True! We reached a fixed point!

R: over-approx

Bad: P = $\neg S9$




```
if ¬check(R ∧ T(s_0, s_1) ∧ ¬R)
    return True
```

Interpolant-based model checking

- Advantages
  - Approximate reachability
  - Clever refinements

- Disadvantages
  - Requires unrolling (can become expensive)
  - Needs to restart every time k is incremented
  - Refinements are clever, but not directly targeting induction