Model Checking

An Overview, Continued…

Goals

- Vocabulary
- High-level understanding of state-of-the-art algorithms
  - Could read the paper and understand it

Timeline

- Formal proof and model checking developments over the years
- Not necessarily to scale
- Today we will cover
  - Interpolant-based Model Checking
  - IC3

1950s First computer-generated proof

Davis-Putnam Algorithm 1960

1960s Stanford Pascal Verifier

1970s State exploration and temporal logic

Model Checking 1981
Emerson and Clarke, Sifrakis

1992 BDD-based model checking
Burch, Clarke, McMillan, Dill, Hwang

GRASP SAT Solver 1996

1999 BMC
Biere, Cimatti, Clarke, Zhu

K-induction 2000
Sheeran, Singh, Stalmarck

2001 Chaff SAT Solver

Interpolant-based MC 2003
Ken McMillan

2003 SMT
Clark Barrett and others independently

2011 IC3
Aaron Bradley

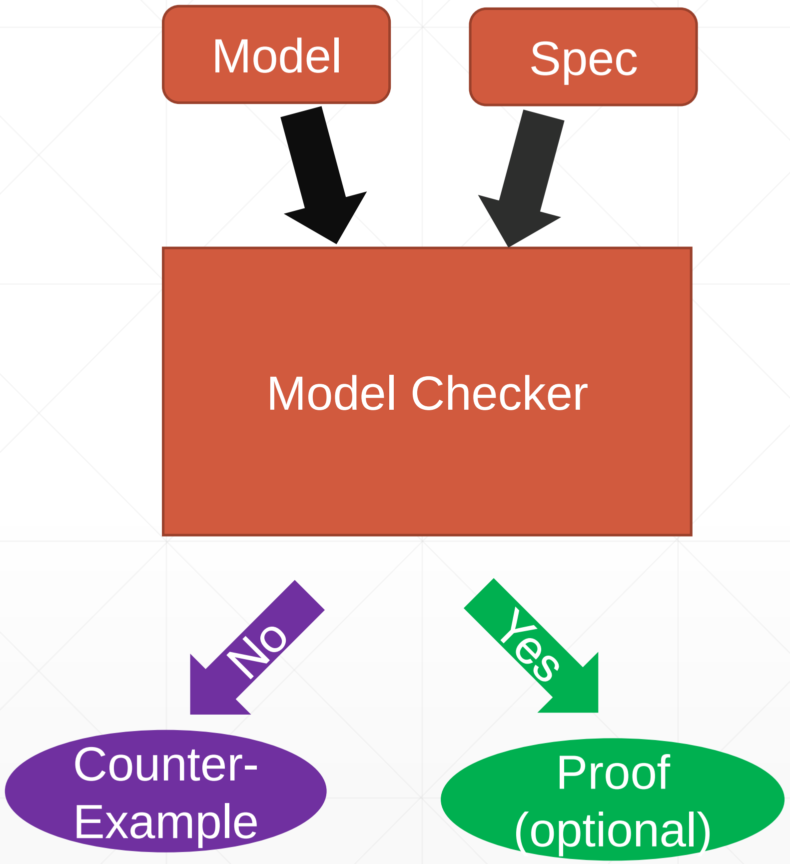
Reference: https://www.eeweb.com/profile/adarbari/articles/a-brief-history-of-formal-verification

Outline

- Review
- Approximations and Inductive Invariants
- Interpolation-based model checking
- IC3/PDR

Review: What Is Model Checking

- An approach for verifying the temporal behavior of a system
- Primarily fully-automated (“push-button”) techniques
- Model
  - Representation of the system
  - Need to decide the right level of granularity
- Specification
  - High-level desired property of system
- Considers infinite sequences
- PSPACE-complete for FSMs



Review: 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…

Review: 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
    - $T(v_0, \ldots, v_n, v_0', \ldots, v_n')$ 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…

Note:
Will often use
$$s \coloneqq \langle v_0, \ldots v_n \rangle$$
to represent a state.

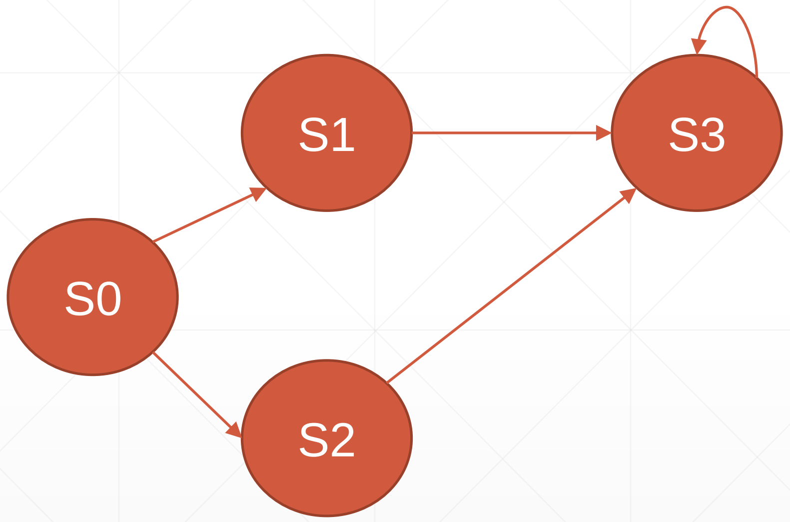
Will use a subscript for time when it matters

Might drop arguments in T

Review: Symbolic Transition System Example

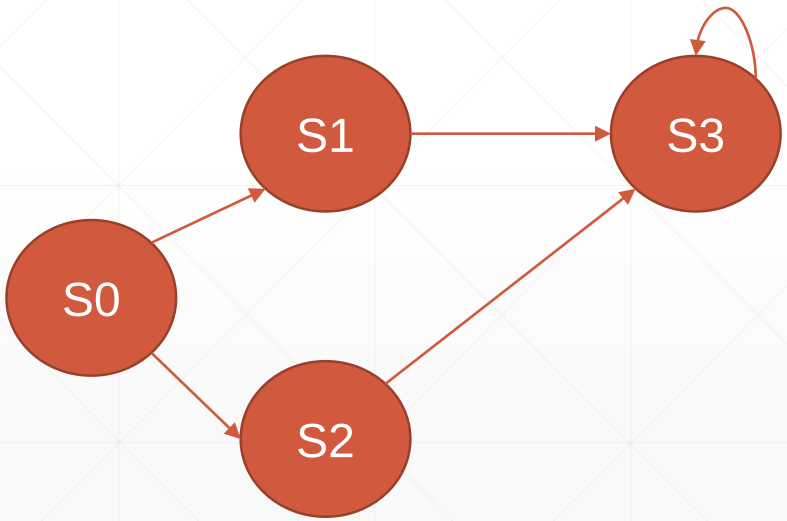
- 2 variables: $V = \{v_0, v_1\}$
  - $S_0 \coloneqq \neg v_0 \land \neg v_1, \quad S_1 \coloneqq \neg v_0 \land v_1$
  - $S_2 \coloneqq v_0 \land \neg v_1, \quad S_3 \coloneqq v_0 \land v_1$
- Transition relation

$$(\neg v_0 \land \neg v_1) \Rightarrow \big((\neg v_0' \land v_1') \lor (v_0' \land \neg v_1')\big) \land$$
$$(\neg v_0 \land v_1) \Rightarrow (v_0' \land v_1') \land$$
$$(v_0 \land \neg v_1) \Rightarrow (v_0' \land v_1') \land$$
$$(v_0 \land v_1) \Rightarrow (v_0' \land v_1')$$

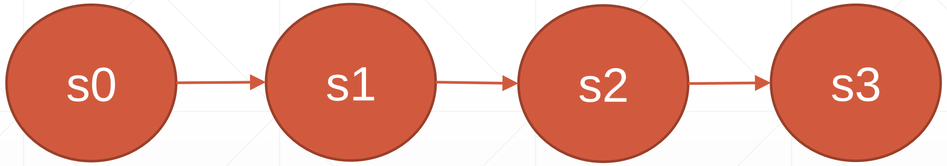


Reminder: State Machine vs Execution

State Machine uses capitals



Symbolic execution uses lowercase



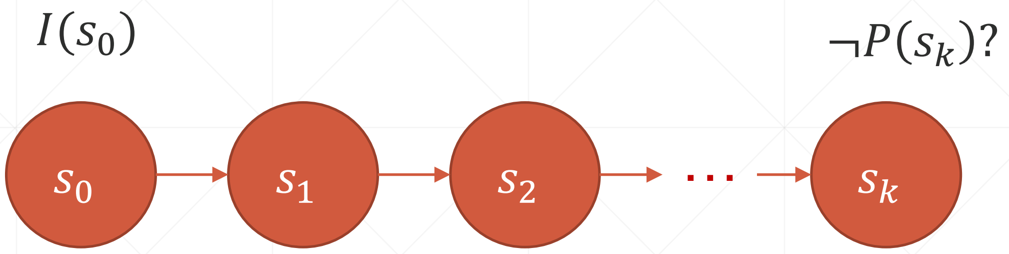
Concrete Execution:

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

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

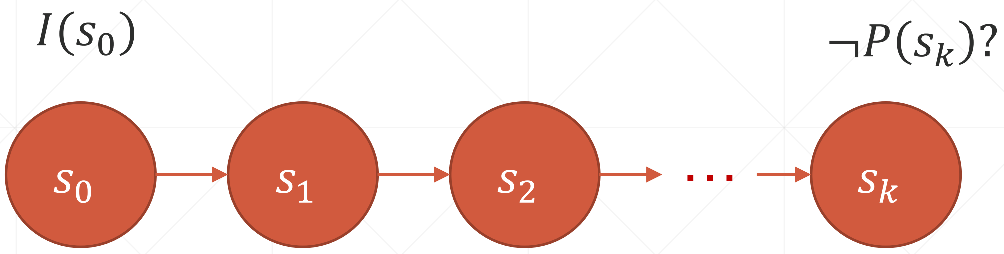
Review: BMC Graphically



s_0 must be an initial state

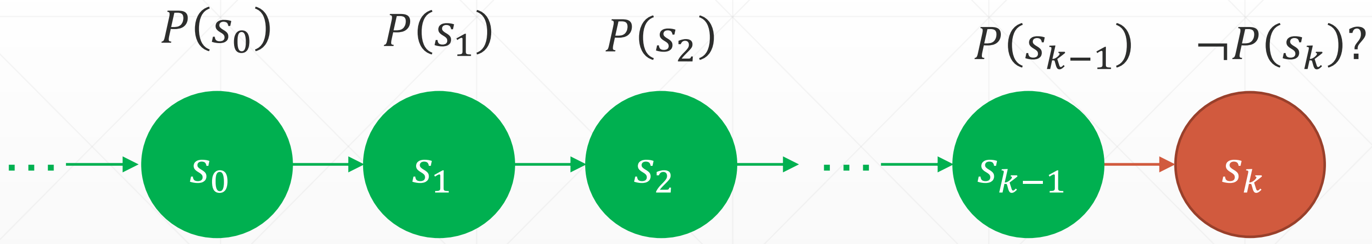
Check if it can violate the property at time k

Review: K-Induction Graphically



Base Case

s_0 must be an initial state

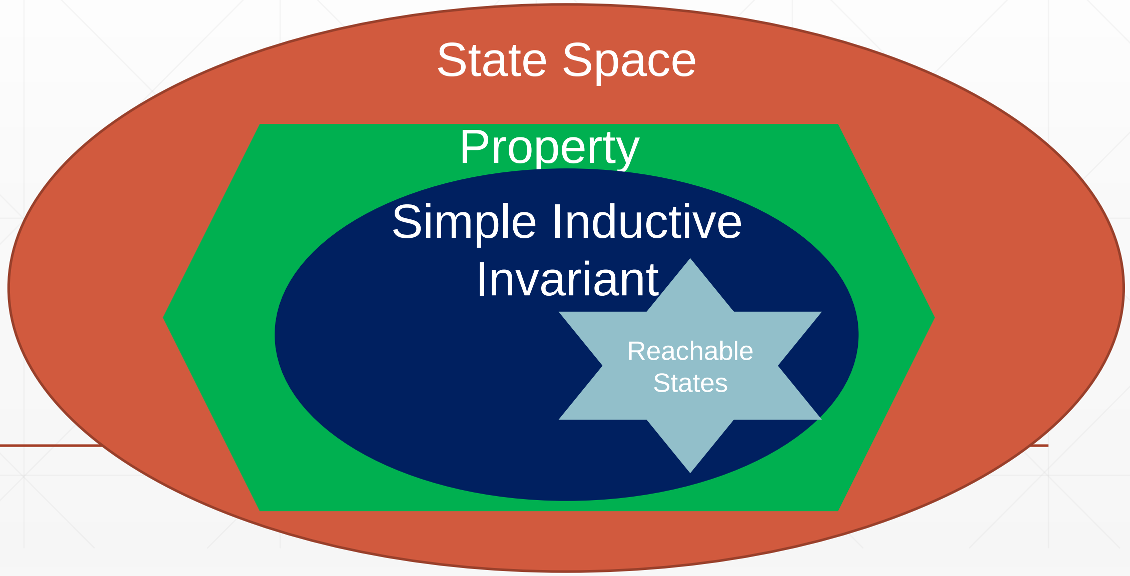


Inductive Case

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

Review: 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
  - E.g. 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
  - $Init(s) \Rightarrow II(s)$
  - $\mathrm{T}(s, s') \wedge II(s) \Rightarrow II(s')$
  - $II(s) \Rightarrow P(s)$



Searching for Inductive Invariants

- Interpolant-based model checking
- IC3/PDR

- For the remainder of this talk, we're assuming *safety* properties
  - Can always perform liveness to safety transformation

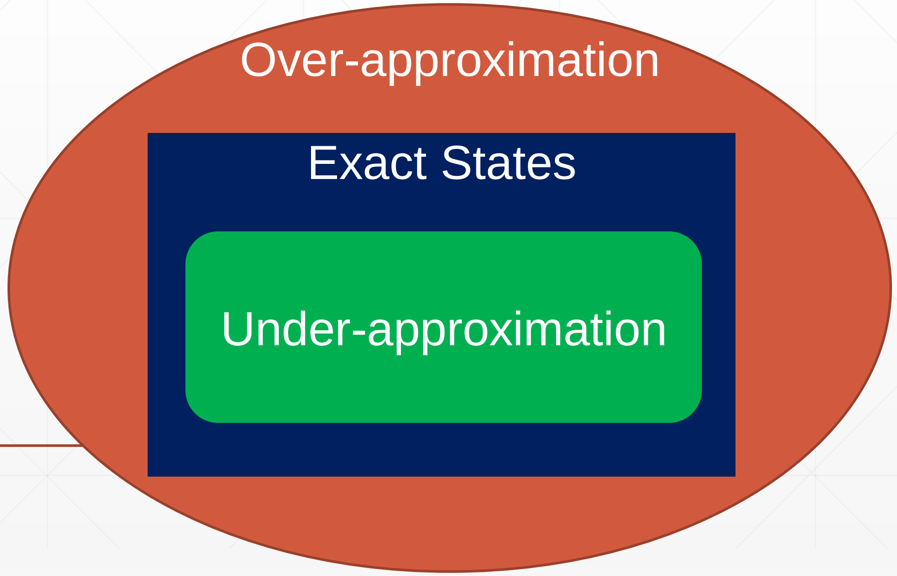
Building Blocks: Approximations

- 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, $O: R \to O$
  - Proofs on over-approximation holds
  - Counterexamples can be spurious
- Under-approximation, $U: U \to R$
  - Proofs on under-approximation can be spurious
  - Counterexamples are real



Craig Interpolation

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

- Craig Interpolant, I
  - $A \rightarrow I$
  - $I \wedge B$ 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

Obtaining Craig Interpolants

- Not all theories admit (quantifier-free) interpolants
  - Arrays do not guarantee quantifier-free interpolants

- Example:

$A \;\coloneqq a = store(b, i, e)$
$B \;\coloneqq select(a, j) \neq select(b, j) \wedge select(a, k) \neq select(b, k) \wedge j \neq k$
$V(A) \cap V(B) \coloneqq \{a, b\}$

- There is an extension to the array theory for supporting quantifier free interpolants: “Quantifier-Free Interpolation of a Theory of Arrays”

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

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

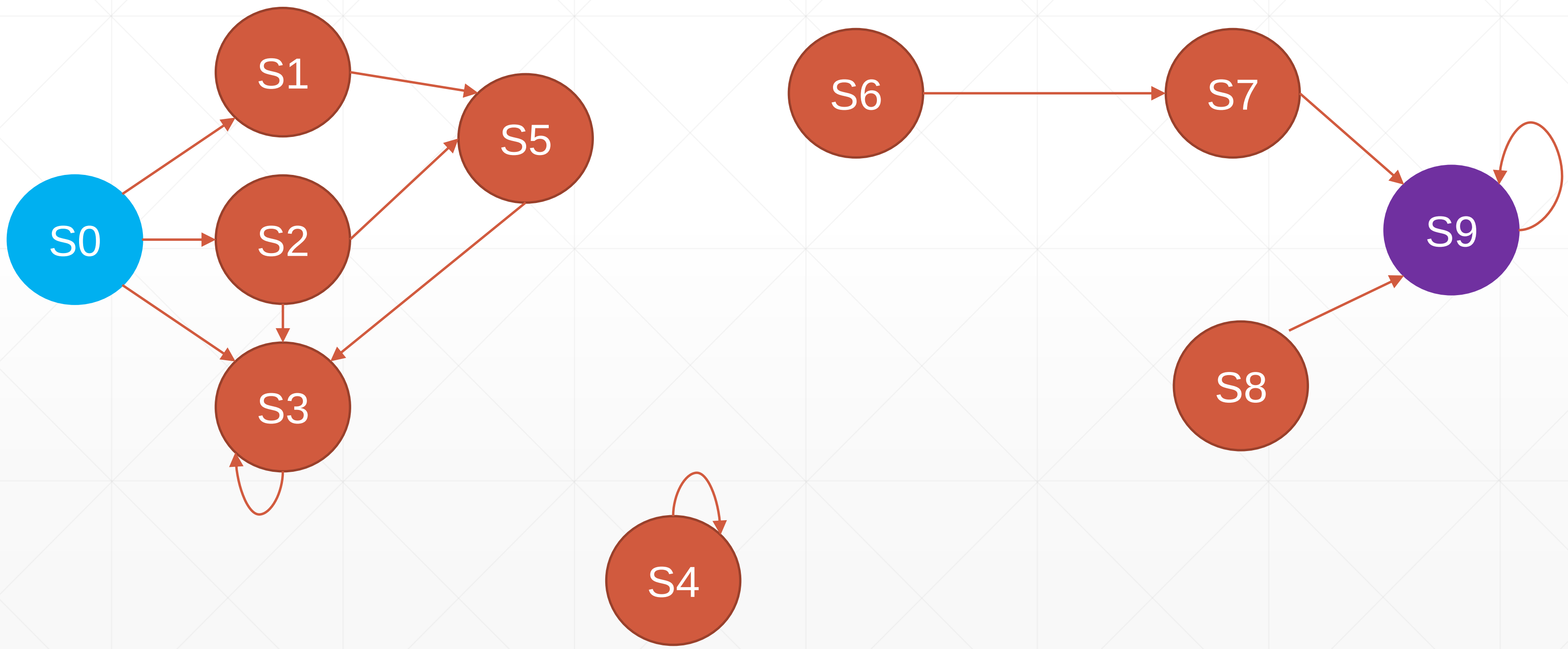
Interpolation-based Model Checking

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

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

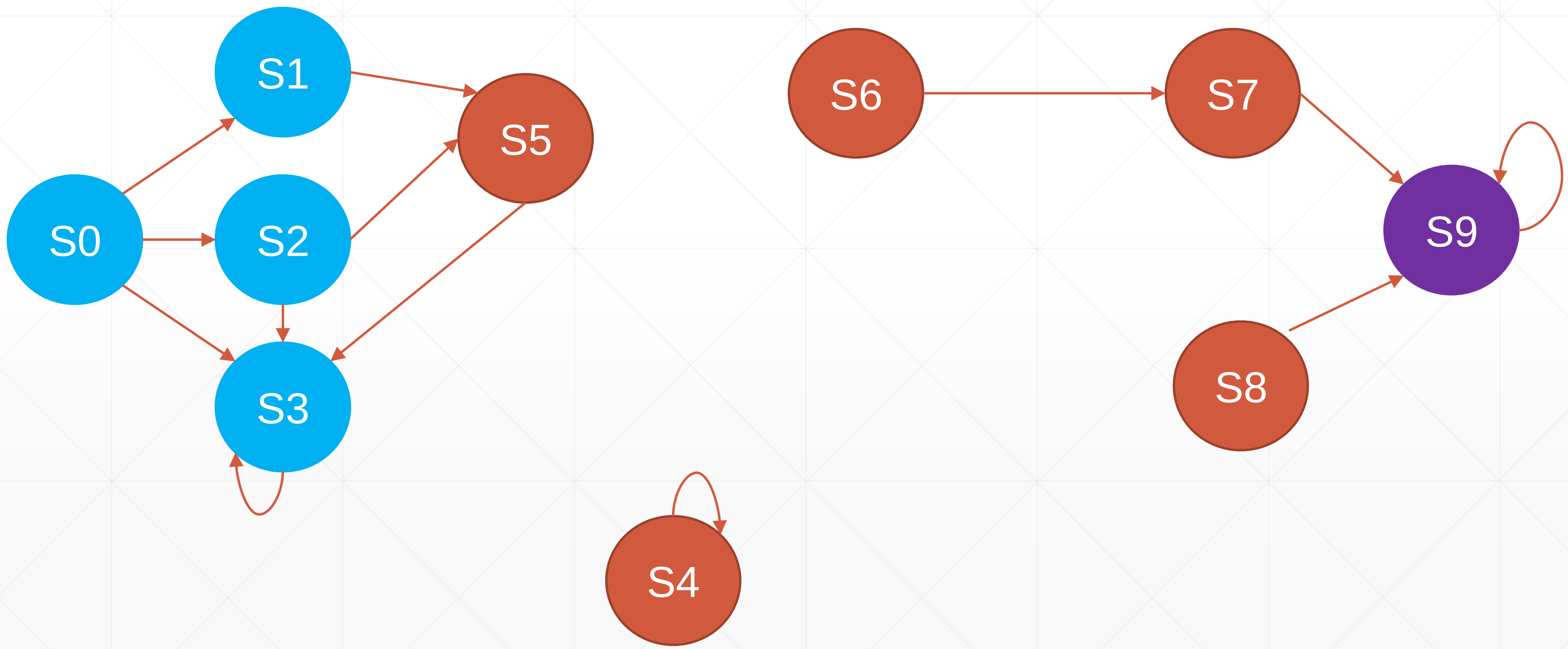
Interpolant-based Model Checking Example

- Start – can't violate in 2 steps



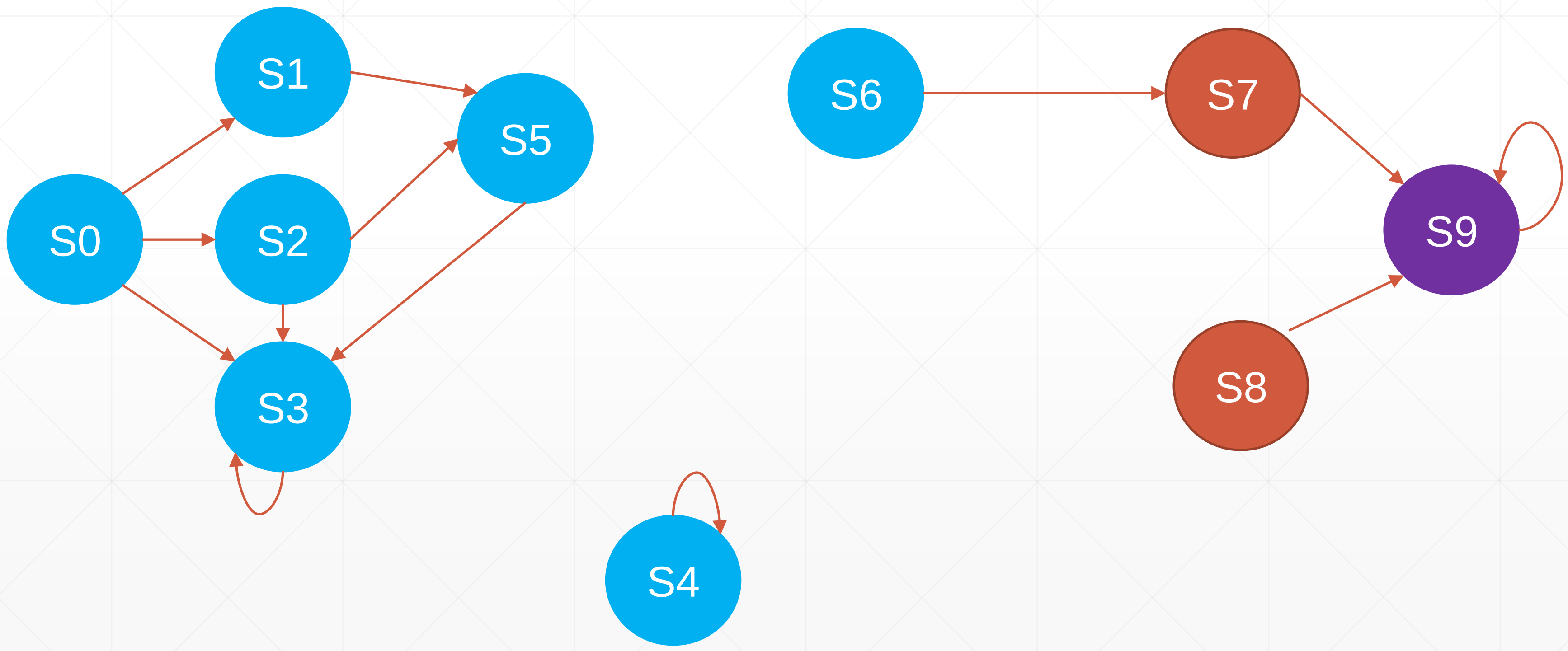
Interpolant-based Model Checking Example

- k = 2



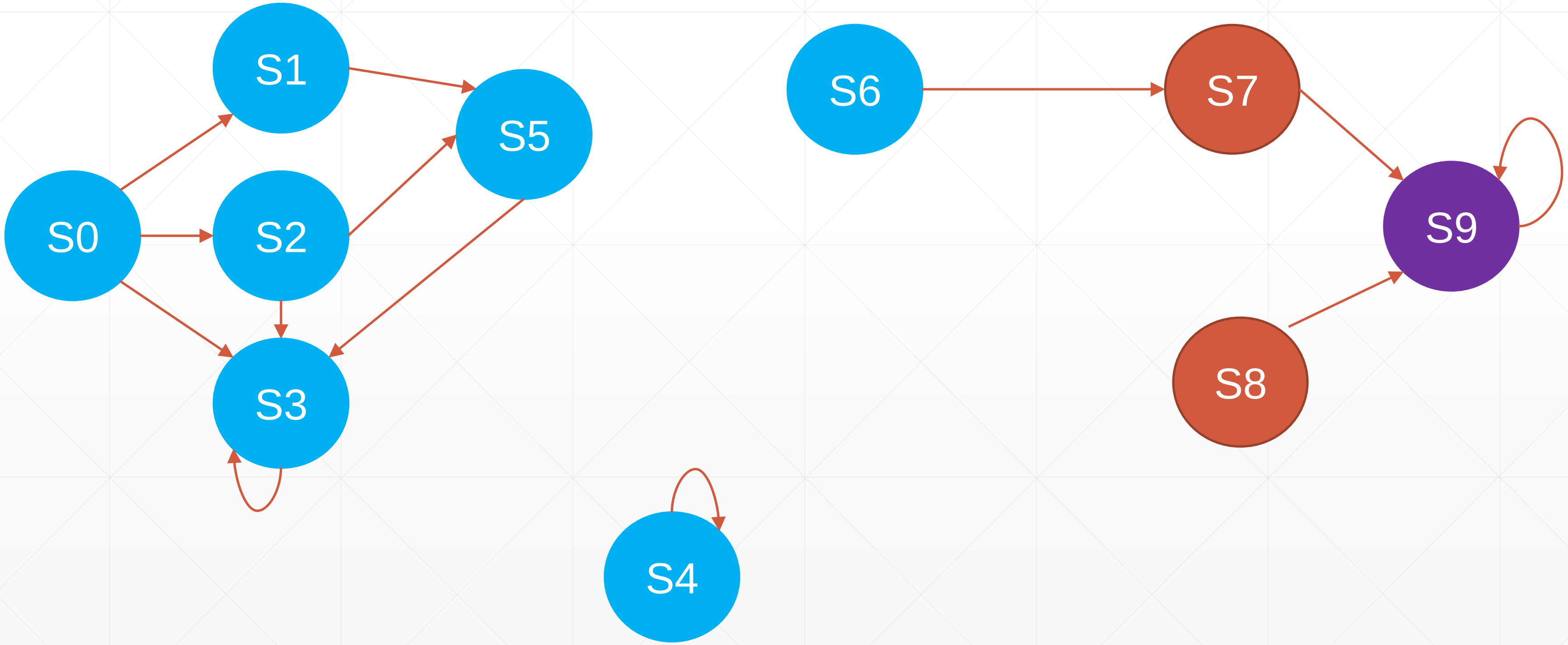
Interpolant-based Model Checking Example

- k = 2



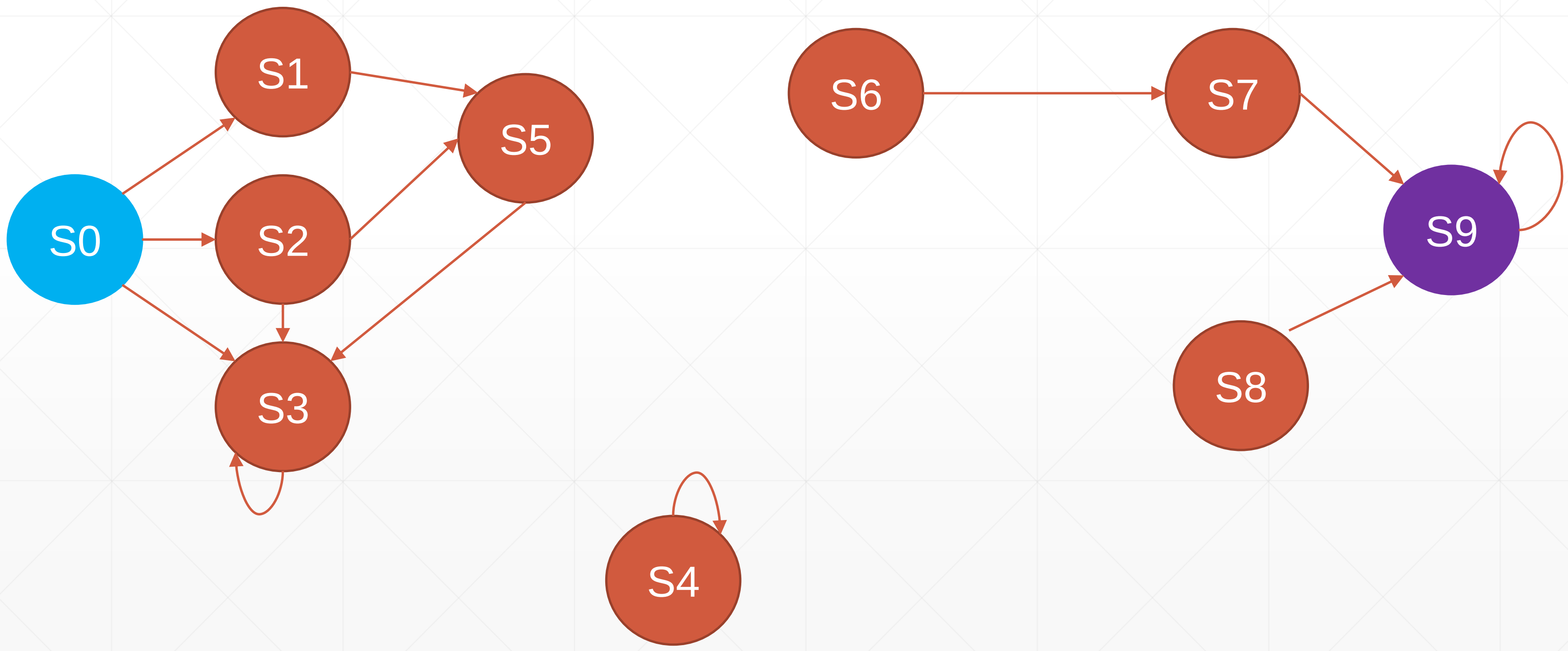
Interpolant-based Model Checking Example

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



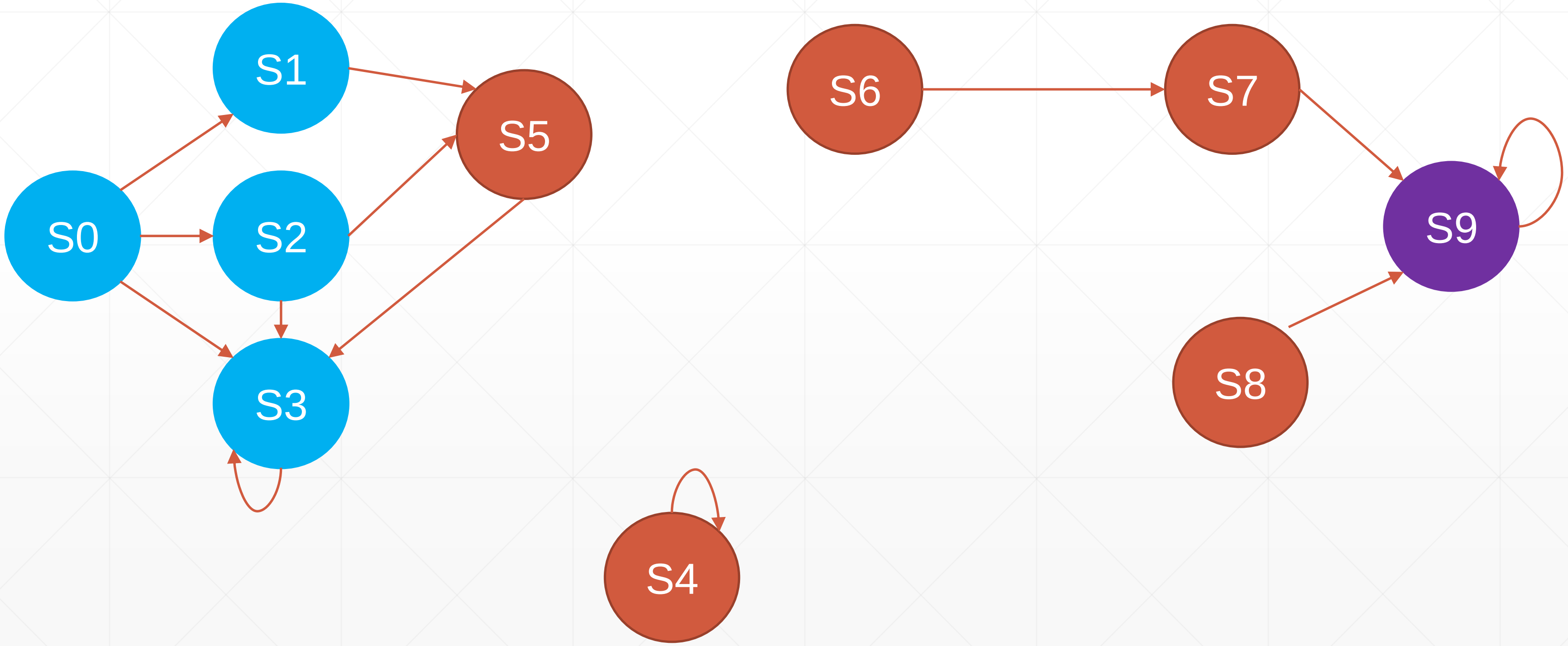
Interpolant-based Model Checking Example

- k = 3



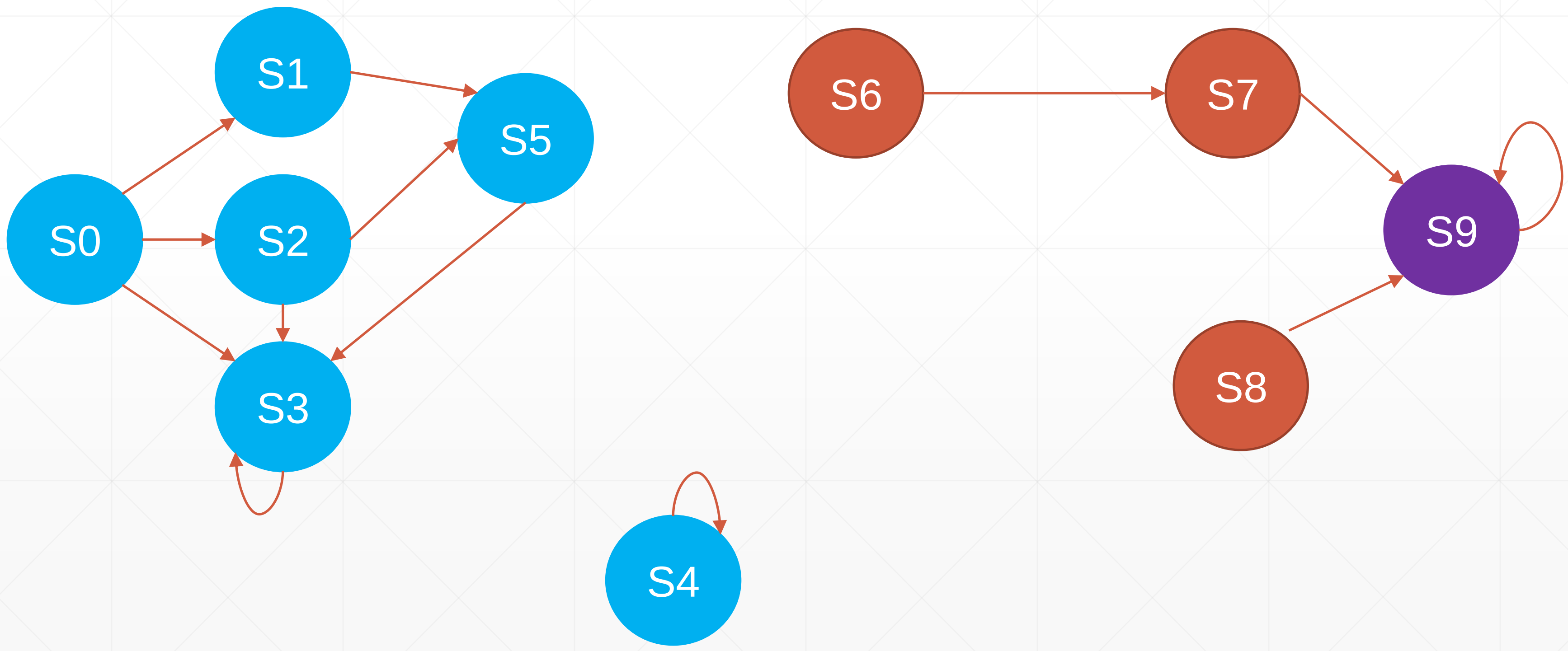
Interpolant-based Model Checking Example

- k = 3



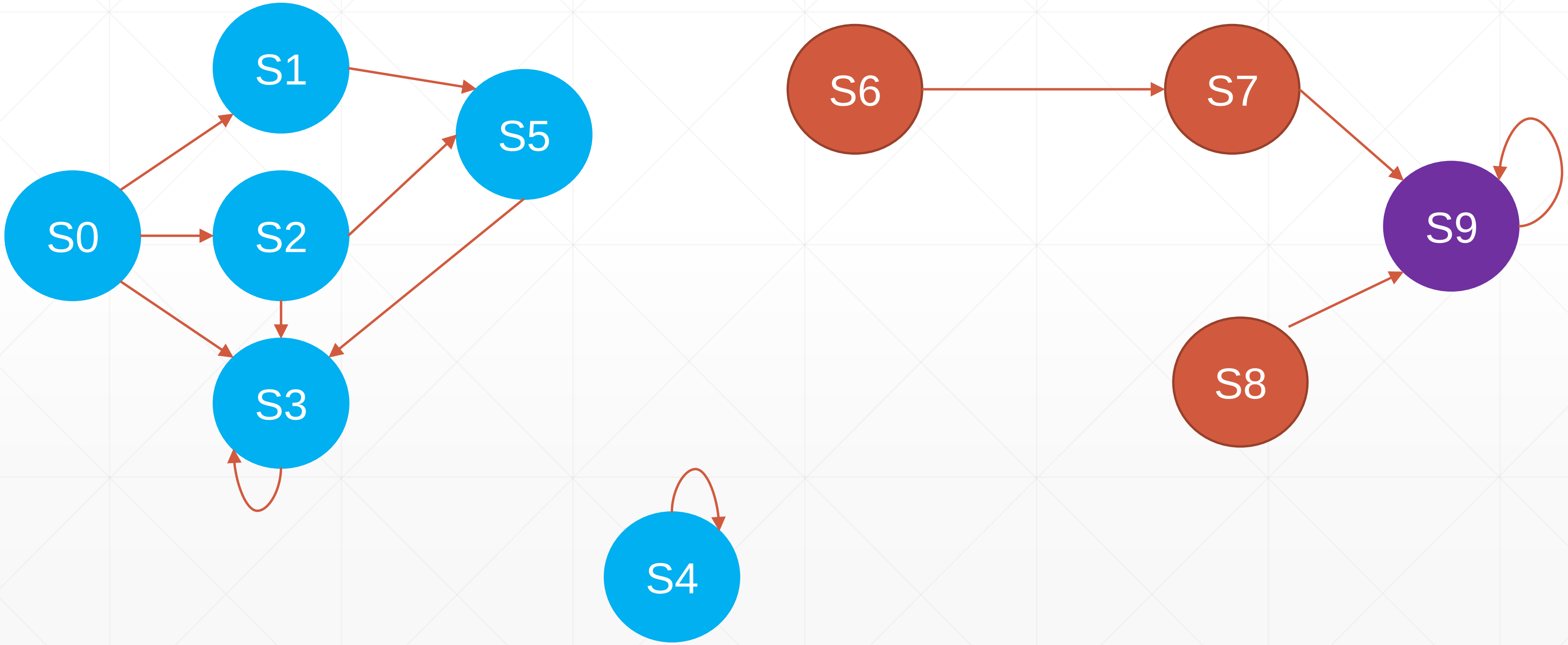
Interpolant-based Model Checking Example

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



Interpolant-based Model Checking Example

- Terminate with 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

IC3 / PDR

- State-of-the-art model checking approach for proofs
- It can also find bugs faster than BMC in some cases
- For the purposes of the talk, focus on SAT
  - Has been extended to SMT, but it's more complicated
  - Covering the simplest version of SAT-based IC3
    - Hybrid of original IC3 paper and PDR paper

IC3: Vocabulary

- Inductive Candidate: C
  - $Init(s) \Rightarrow C(s)$ — Initiation
  - $\mathrm{T}(s, s') \wedge C(s) \Rightarrow C(s')$ — Consecution

- Manipulating variables
  - $v_0 \vee \neg v_2 \vee v_8$ — Clause
  - $\neg v_0 \wedge v_2 \wedge \neg v_8$ — Cube (inverse of clause)
- State
  - $s = v_0 \wedge \neg v_1 \wedge \cdots \wedge v_n$ — Cube over all variables (aka a “minterm”)

IC3: Vocabulary

```
x = 1; y = 1;
while *:
    y = y + x;
    x = x + 1;

Property: y ≥ 1
```

- Counterexample to Induction (CTI)
  - Model assignment from failed consecution

- Attempt consecution on this program using property as inductive candidate
  - E.g. k-induction for k = 1
  - $\underbrace{y \geq 1}_{P} \wedge \underbrace{x' = x + 1 \wedge y' = y + x}_{\text{transition relation}} \wedge \underbrace{\neg(y' \geq 1)}_{P'}$ is SAT (consecution fails)
  - CTI: {x = -1, y = 1}

IC3: Relative Induction

```
x = 1; y = 1;
while *:
    y = y + x;
    x = x + 1;

Property: y ≥ 1
```

- Property $y \geq 1$ is not inductive

- System does have an easy invariant: $\phi \;\coloneqq\; x \geq 0$
  - $x \geq 0$ true in the initial state
  - $\mathrm{x} \geq 0 \land x' = x + 1 \land y' = y + x \land \neg(x' \geq 0)$ is UNSAT (inductive proof)

- Property $y \geq 1$ is inductive *relative* to this invariant, ϕ
  - $\underbrace{\mathrm{x} \geq 0}_{\phi} \land \underbrace{y \geq 1}_{P} \land \underbrace{x' = x + 1 \land y' = y + x}_{\text{transition relation}} \land \neg\underbrace{(y' \geq 1)}_{P'}$ is UNSAT

High-level Idea

- Build a sequence of over-approximations (e.g. formulas)
  - Sequence of *frames*, F
    - where F[k] is an over-approximation of the states reachable in k steps
  - Frames are in CNF

- Refine these frames using CTIs

- When there is an F[i] that is (one-step) inductive, you are done

- If the property is false, you'll discover that when trying to refine a frame

Another View

- What are frames?
  - F[k] over-approximates the states reachable in k steps

- Alternatively,
  - F[k] contains a “guess” at invariants
  - They don’t hold inductively yet
  - But, they hold for up to k steps
  - i.e. they seem like reasonable guesses for an invariant

IC3: Details

- IC3 maintains the following invariants on its frames:
  - $F[0] = Init$
  - $F[i] \wedge T \rightarrow F[i+1]$ for $0 \leq i < k$
  - $F[i] \rightarrow P$ for $0 \leq i < k$

- Note that F[k] does not necessarily imply P
  - We iteratively refine it until it does imply P,

IC3: Proof Obligations

- Proof obligation (s, i)
  - Cube s at frame i

- Handling proof obligations: Check $F[i-1] \wedge \neg s \wedge T \wedge s'$
  - If UNSAT
    - $\neg s$ is inductive relative to F[i-1] (aka not reachable in one-step from F[i-1]
  - If SAT, get a CTI
    - $\exists\, c\,.\, F[i-1] \wedge \neg s \wedge c \wedge T \rightarrow s'$
    - There’s a state contained in F[i-1] that reaches s’ in one step
    - Add proof obligation $(c, i-1)$ and recurse

IC3: Proof Obligation Outcomes

- Case 1: Counterexample

F[0] = Init

F[1]

F[2]

⋮

F[k] $(s_k$, k) obtained from $F[k] \wedge \neg P'$

IC3: Proof Obligation Outcomes

- Case 1: Counterexample – obtain trace from recursive proof obligations

F[0] = Init	s_0 reachable from Init
F[1]	$(s_1, 1)$ obtained from $F[1] \wedge \neg s_2 \wedge T \wedge s_2'$
F[2]	$(s_2, 2)$ obtained from $F[2] \wedge \neg s_3 \wedge T \wedge s_3'$
⋮	⋮
F[k]	(s_k, k) obtained from $F[k] \wedge \neg P'$

IC3: Proof Obligation Outcomes

- Case 2: s is not reachable

F[0] = Init	
F[1]	$(s_1, 1)$
F[2]	$(s_2, 2)$
⋮	⋮
F[k]	(s_k, k) obtained from $F[k] \wedge \neg P'$

IC3: Proof Obligation Outcomes

- Case 2: s is not reachable

F[0] = Init	s_0 not reachable from Init
F[1]	$(s_1, 1)$
F[2]	$(s_2, 2)$
⋮	⋮
F[k]	(s_k, k) obtained from $F[k] \wedge \neg P'$

IC3: Proof Obligation Outcomes

- Case 2: s is not reachable

F[0] = Init	s_0 not reachable from Init	block in F[1]
F[1]	(s_1, 1)	
F[2]	(s_2, 2)	
⋮	⋮	
F[k]	(s_k, k) obtained from $F[k] \wedge \neg P'$	

IC3: Proof Obligation Outcomes

- Case 2: s is not reachable – refined frames

F[0] = Init	s_0 not reachable from Init	block in F[1]
F[1]	(s_1, 1)	block in F[2]
F[2]	(s_2, 2)	block in F[3]
⋮	⋮	
F[k]	(s_k, k) obtained from $F[k] \wedge \neg P'$	blocked by recursion

IC3 Main Loop

```
while SAT ? [F[k] ∧ ¬P]
   extract a bad state, s
   recursively block proof obligation (s, k)


Termination conditions:
    1. For some i, F[i] is inductive:    Property is TRUE
    2. Pushed proof obligation to Init:  Property is FALSE
```

Congratulations!

- You made it through the IC3 explanation!!

Congratulations!

- You made it through the IC3 explanation!!

- But you might be wondering, is that it?
  - We CAN'T just be blocking one state at a time, right?

Generalization

- For counterexample to induction, s
  - Before creating a proof obligation: (s, i)
  - Generalize s to cover more states
- Recall, the more literals in a cube s, the less states it covers
- Several generalization techniques
  - Simplest one: ternary simulation
  - Get model, replace one literal with X and simulate
  - If no X makes it to next state, then that literal is unnecessary (drop it)

IC3 Example

Init does not intersect with bad state

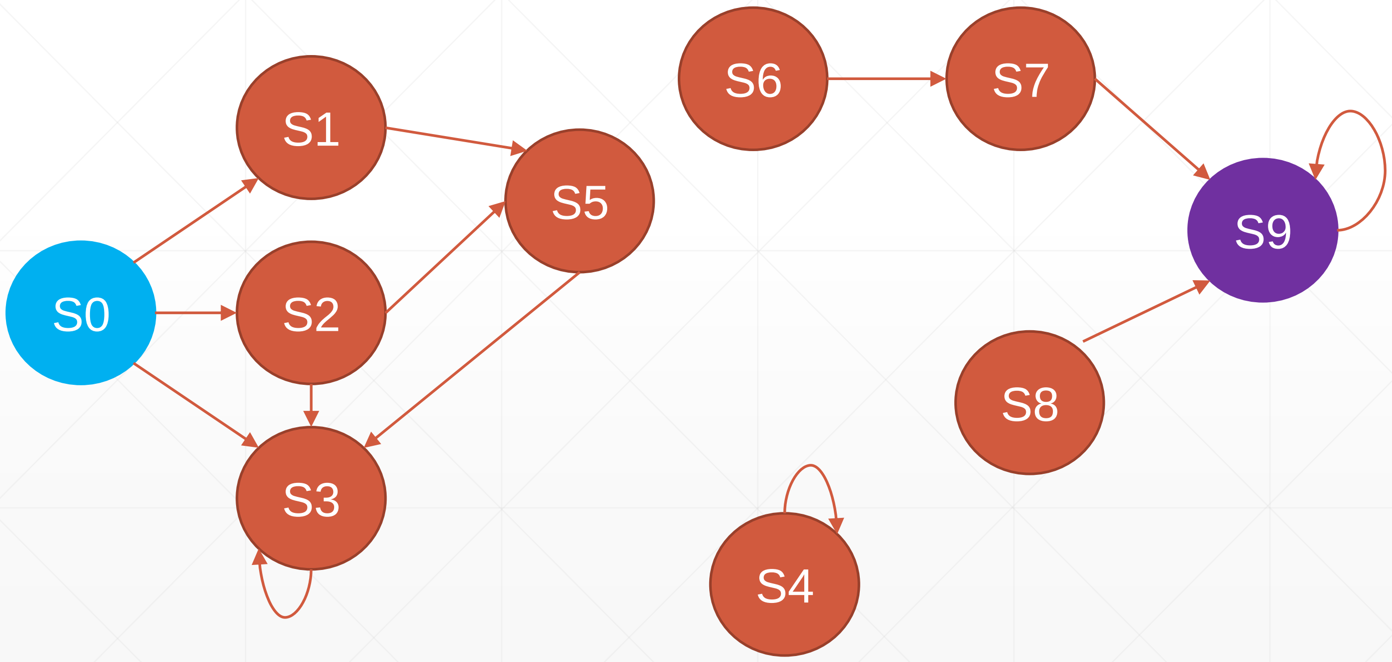
F[0] = Init

In F[k-1]*

Bad

P = $\neg S9$

*Let's just ignore the 0 case



In F[k-1]*
Bad

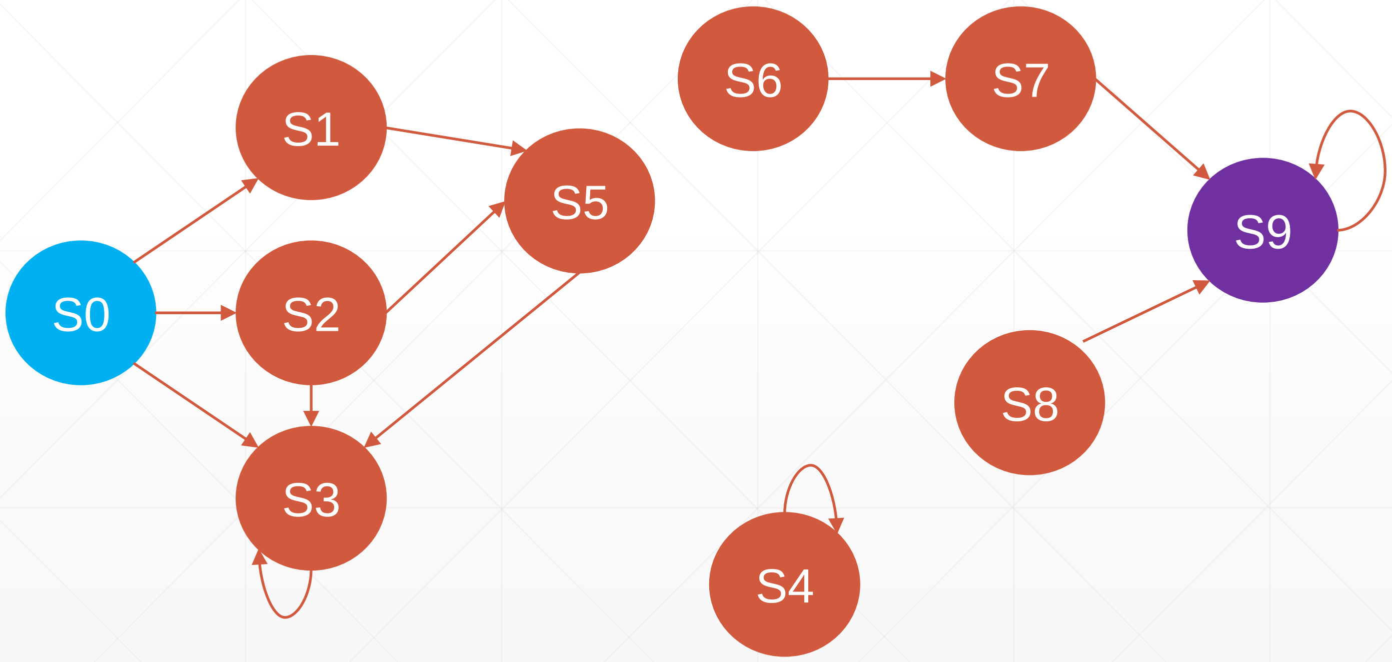
*Let's just ignore the 0 case

IC3 Example

Push Frame

F[0] = Init

F[1] = True



In F[k-1]*
Bad

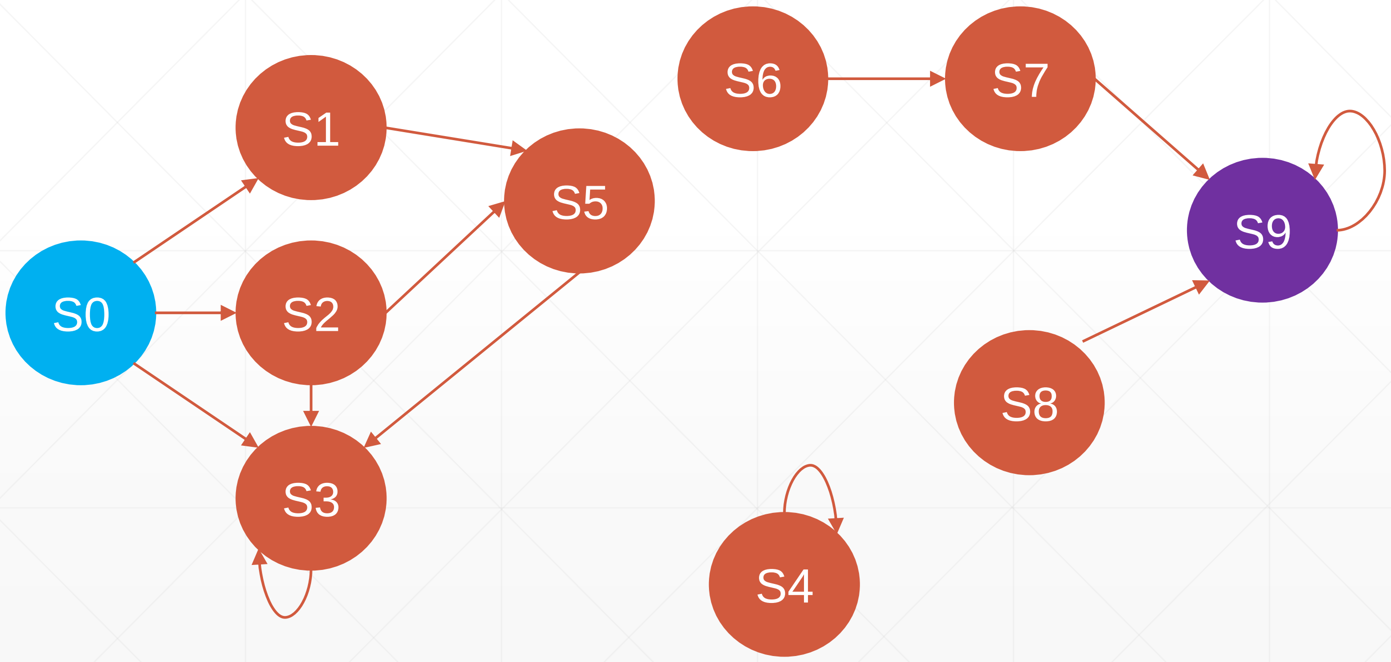
IC3 Example

*Let's just ignore the 0 case

$F[1] \land \neg P$ is SAT, proof obligation (S9, 1)

F[0] = Init

F[1] = True



IC3 Example

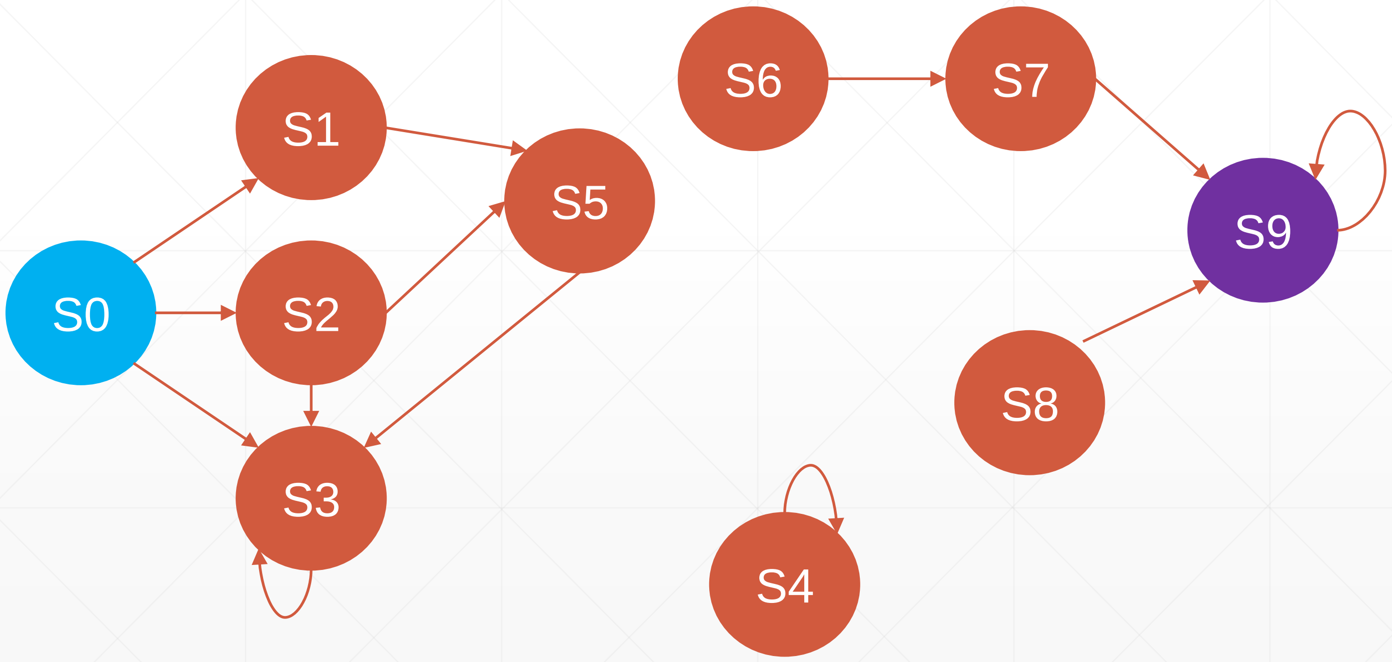
$F[0] \wedge \neg S9 \wedge S9'$ is UNSAT, block S9

F[0] = Init

F[1] = $\neg S9$

In F[k-1]*
Bad

*Let's just ignore the 0 case



In F[k-1]*
Bad

IC3 Example

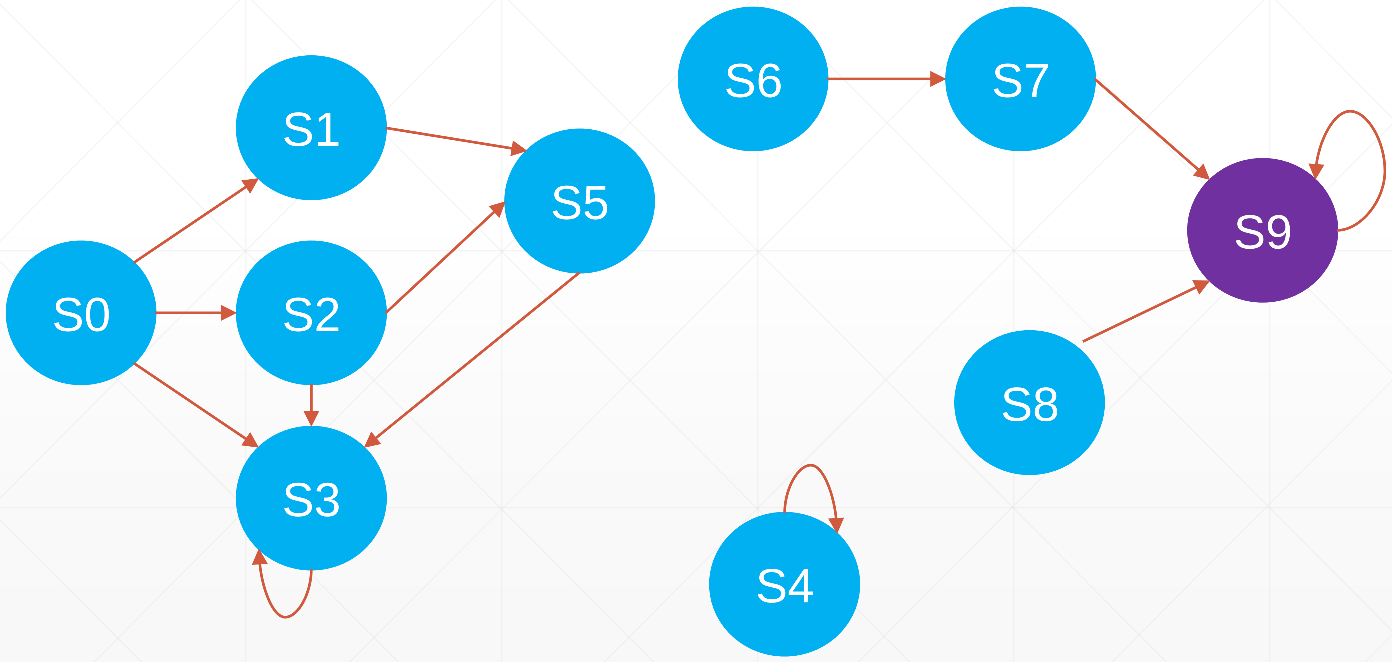
*Let's just ignore the 0 case

Push Frame

F[0] = Init

F[1] = $\neg S9$

F[2] = True



In F[k-1]*
Bad

*Let's just ignore the 0 case

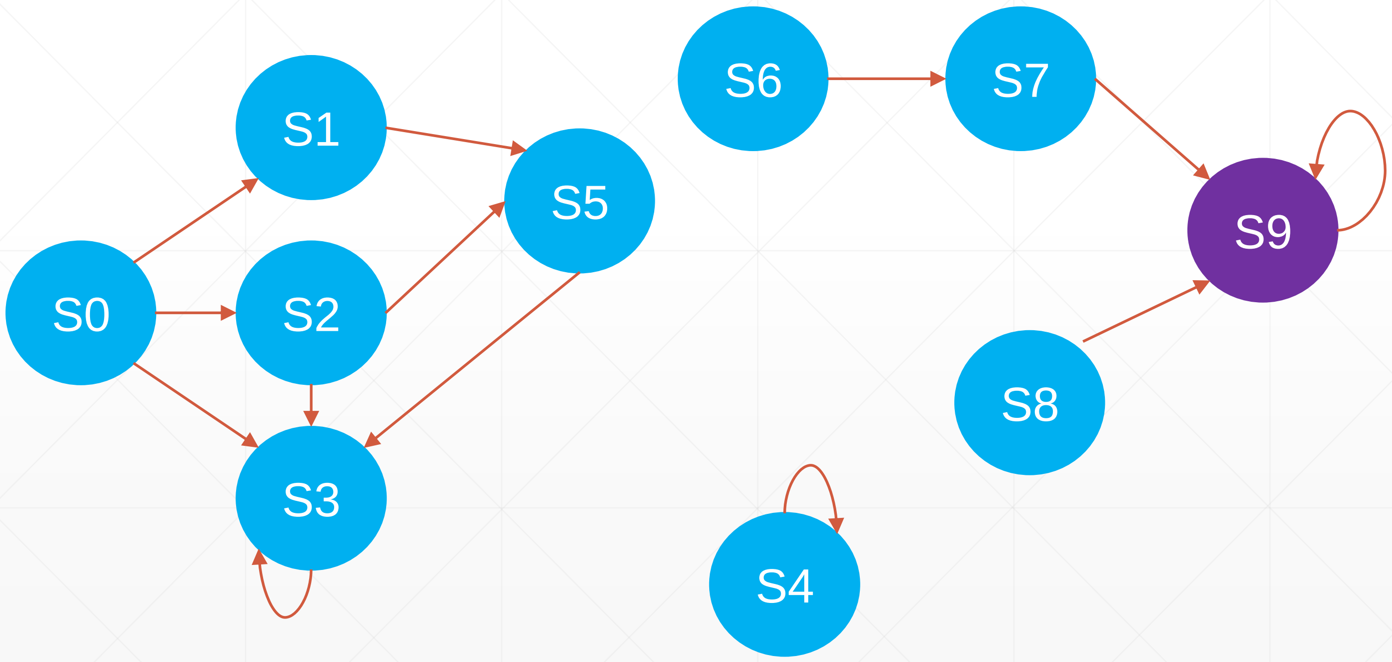
IC3 Example

$F[2] \wedge \neg P$ is SAT, proof obligation (S9, 2)

F[0] = Init

F[1] = $\neg S9$

F[2] = True



In F[k-1]*
Bad

IC3 Example

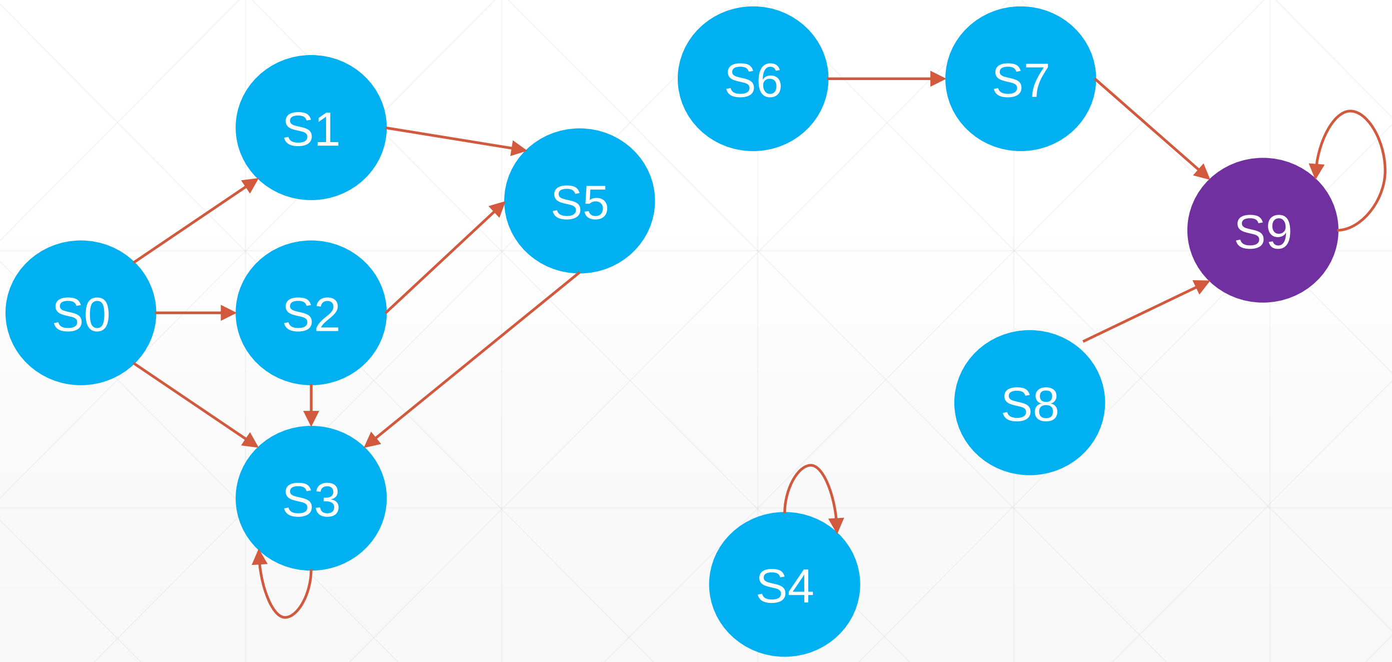
*Let's just ignore the 0 case

$F[1] \wedge \neg S9 \wedge S9'$ is SAT, proof obligation + generalization ($(S7 \vee S8$, 1)

F[0] = Init

F[1] = $\neg S9$

F[2] = True



In F[k-1]*
Bad

IC3 Example

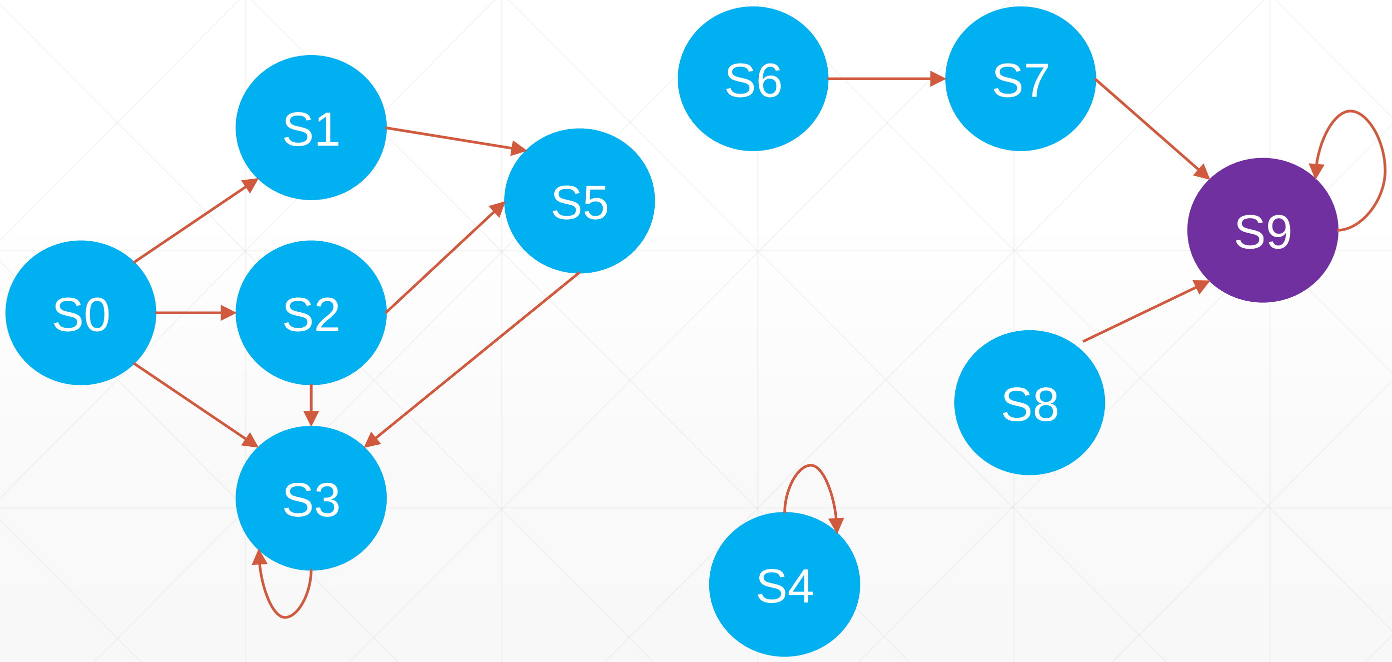
*Let's just ignore the 0 case

$F[0] \wedge \neg(S7 \vee S8) \wedge (S7' \vee S8')$ is UNSAT, block $S7 \vee S8$

F[0] = Init

F[1] = $\neg S9 \wedge \neg S7 \wedge \neg S8$

F[2] = $\neg S9$



In F[k-1]*
Bad

*Let's just ignore the 0 case

IC3 Example

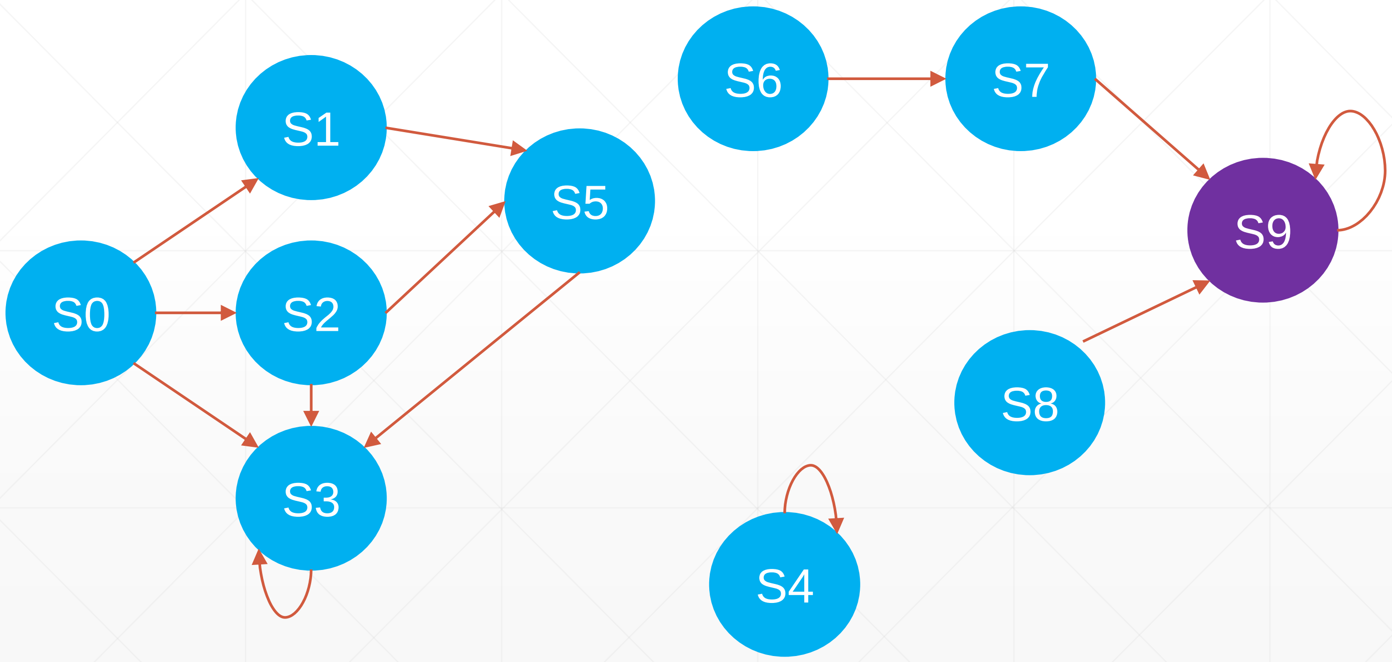
Push Frame

F[0] = Init

F[1] = $\neg S9 \wedge \neg S7 \wedge \neg S8$

F[2] = $\neg S9$

F[3] = True



In F[k-1]*
Bad

IC3 Example

*Let's just ignore the 0 case

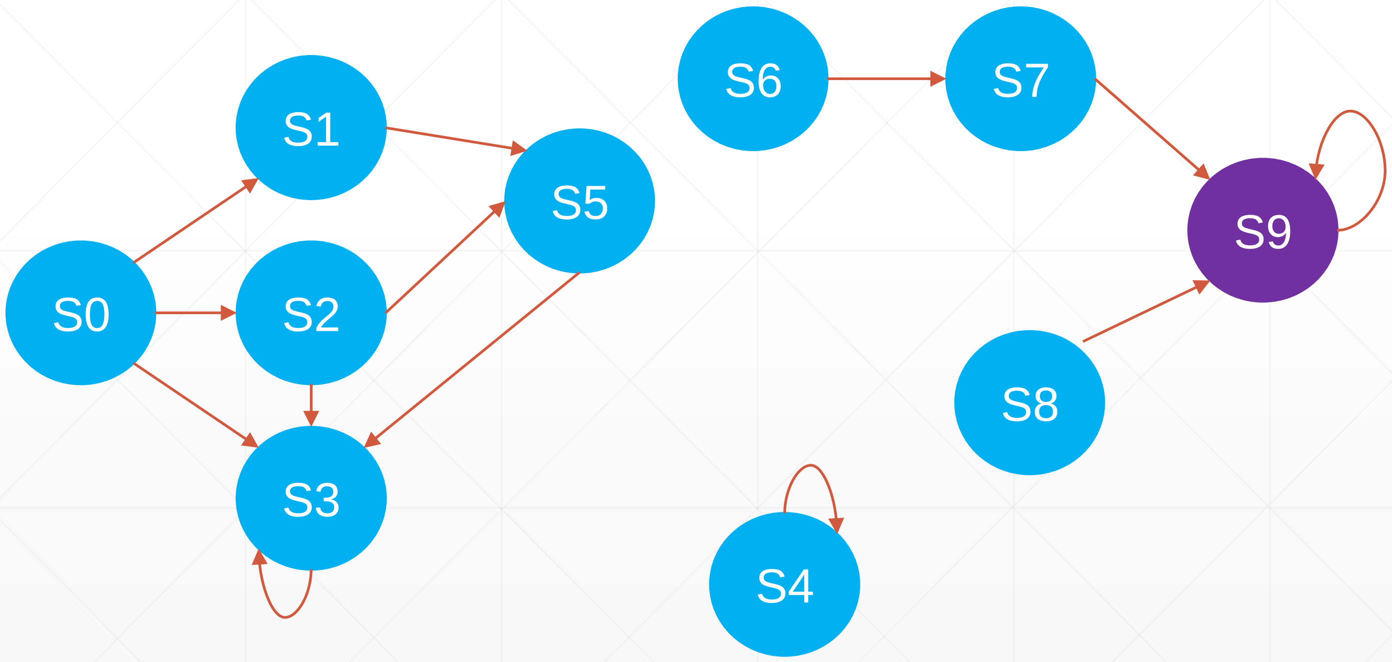
$F[3] \wedge \neg P$ is SAT, proof obligation (S9, 3)

F[0] = Init

F[1] = $\neg S9 \wedge \neg S7 \wedge \neg S8$

F[2] = $\neg S9$

F[3] = True



In F[k-1]*
Bad

*Let's just ignore the 0 case

IC3 Example

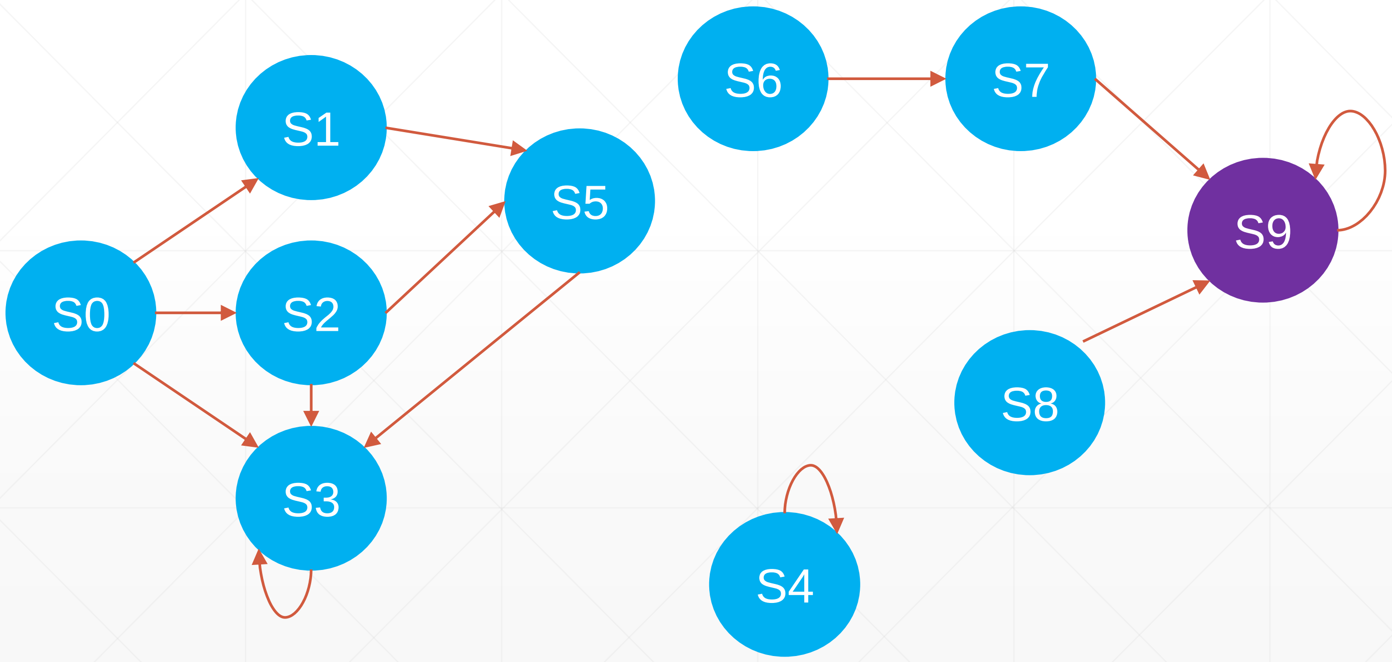
$F[2] \wedge \neg S9 \wedge S9'$ is SAT, proof obligation + generalization ($S7 \vee S8$, 2)

F[0] = Init

F[1] = $\neg S9 \wedge \neg S7 \wedge \neg S8$

F[2] = $\neg S9$

F[3] = True



In F[k-1]*
Bad

IC3 Example

*Let's just ignore the 0 case

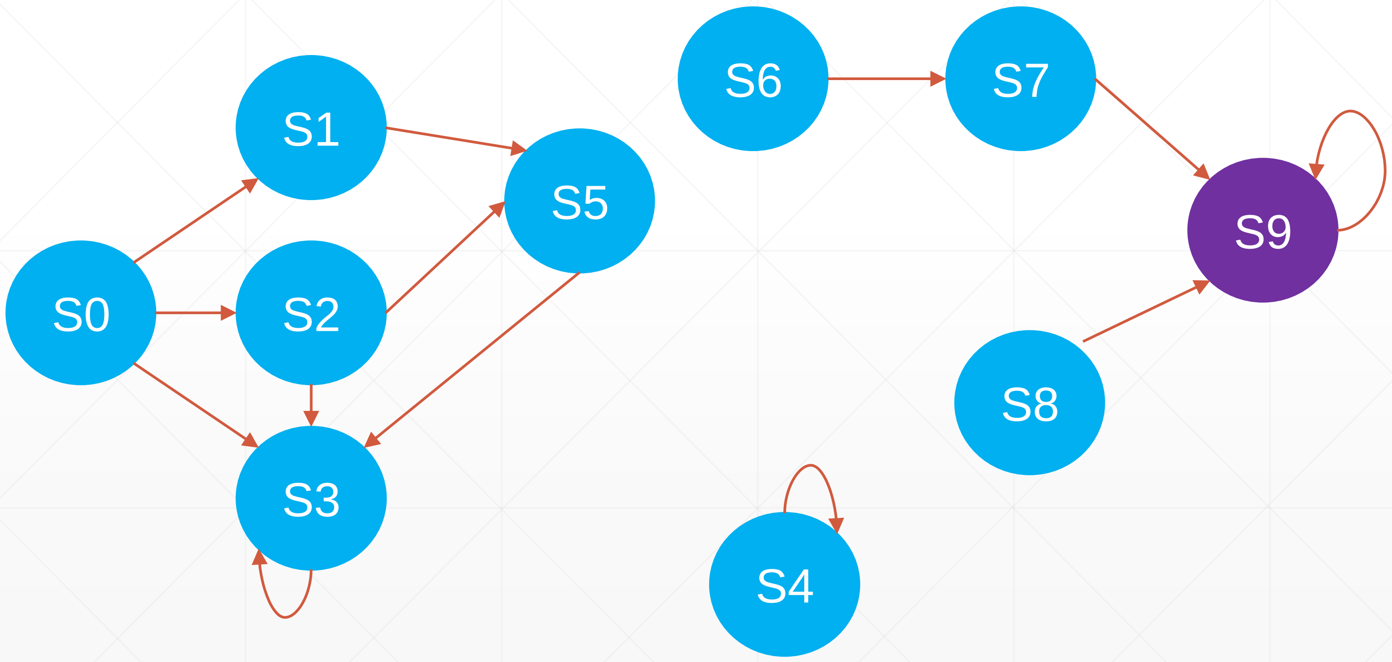
$F[1] \wedge \neg(S7 \vee S8) \wedge (S7' \vee S8')$ is SAT, proof obligation (S6, 1)

F[0] = Init

F[1] = $\neg S9 \wedge \neg S7 \wedge \neg S8$

F[2] = $\neg S9$

F[3] = True



In F[k-1]*

Bad

*Let's just ignore the 0 case

IC3 Example

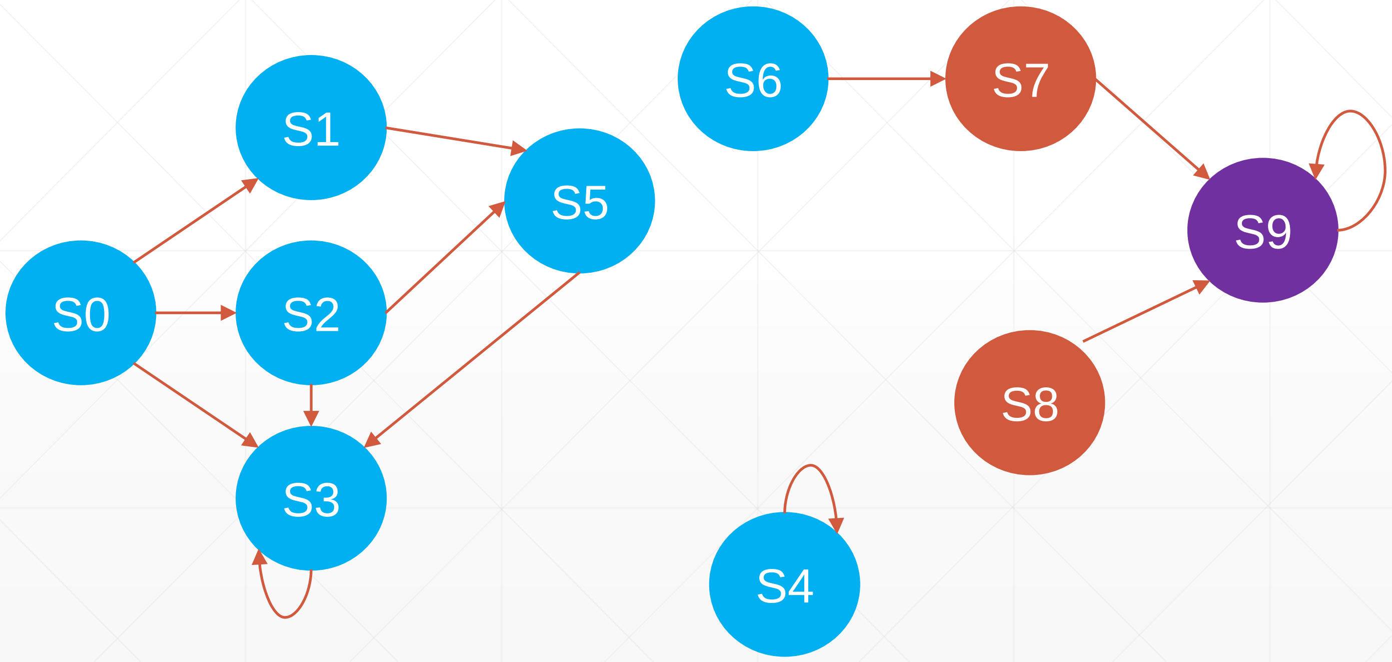
$F[0] \wedge \neg S6 \wedge S6$ is UNSAT, block S6 and previous proof obligations

F[0] = Init

F[1] = $\neg S9 \wedge \neg S7 \wedge \neg S8 \wedge \neg S6$

F[2] = $\neg S9 \wedge \neg S7 \wedge \neg S8$

F[3] = $\neg S9$



In F[1]
Bad

IC3 Example

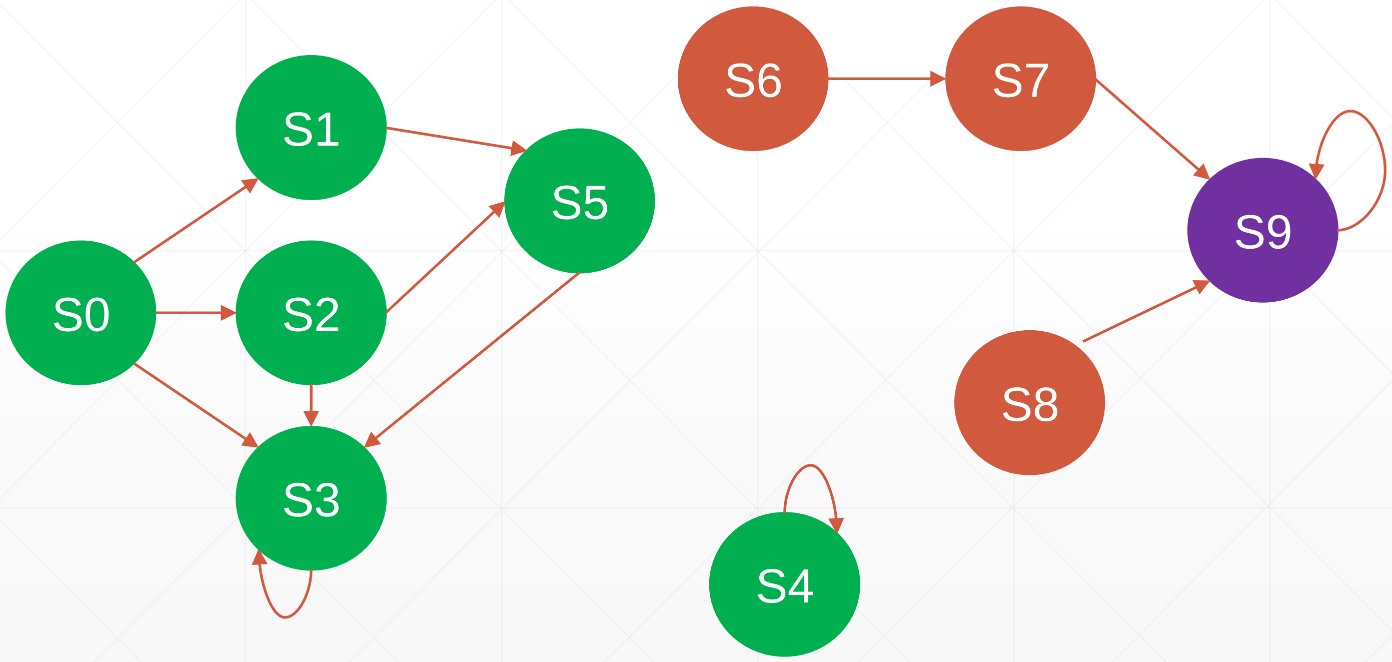
$F[1]$ is inductive!   Terminate with TRUE!

F[0] = Init

F[1] = $\neg S9 \wedge \neg S7 \wedge \neg S8 \wedge \neg S6$

F[2] = $\neg S9 \wedge \neg S7 \wedge \neg S8$

F[3] = $\neg S9$



IC3 In Practice

- Add extra invariant to algorithm: Clauses(F[i-1]) ⊆ Clauses(F[i])
  - Requires some processing during the algorithm
  - But, then inductiveness check is easier
    - Every clause c in F[i] was obtained with a relative inductive check
    - So if F[i-1] = F[i] *syntactically* then the set of clauses is inductive

- IC3 can be easily parallelized
  - Instances of IC3 share produced lemmas, but not how they were obtained

IC3 in Practice

- Maintain a sequence of frames that are backward reachable from *bad*
  - This is an *underapproximation* of states that can violate the property in up to k steps
  - Property is false if the forward and backward frames ever contain the same state (intersect)
- This version of the algorithm introduces choice
  - Previous model checking algorithms always had only one next step
  - IC3 with two sets of frames can have multiple next steps (like a proof calculus)
  - Many heuristics on when to apply which actions
- Plus many other optimizations, improvements and extensions (e.g. to SMT)

Intuition: Incremental vs Monolithic

- “When humans analyze systems, they produce a set of lemmas — typically inductive properties — that together imply the desired property. Each lemma holds relative to some subset of previously proved lemmas in that this prior knowledge is invoked in proving the new lemma. A given lemma usually focuses on just one aspect of the system”

  - Aaron Bradley in SAT-based Model Checking Without Unrolling

Intuition: Distribution of Responsibility

- BMC puts all the work on the solver
- Interpolation-based Model Checking puts most the work on the solver
- IC3, by contrast, is relatively easy on the solver
  - A typical IC3 run has tens of thousands (or more) calls to the solver checking for one-step inductiveness
  - But, each call is easy
  - A *controlled* SAT call that prioritizes local reasoning, as opposed to unrolling based approaches that consider an execution

Thank you!