# Automating safety property verification, intro to liveness

# Stateful invariants

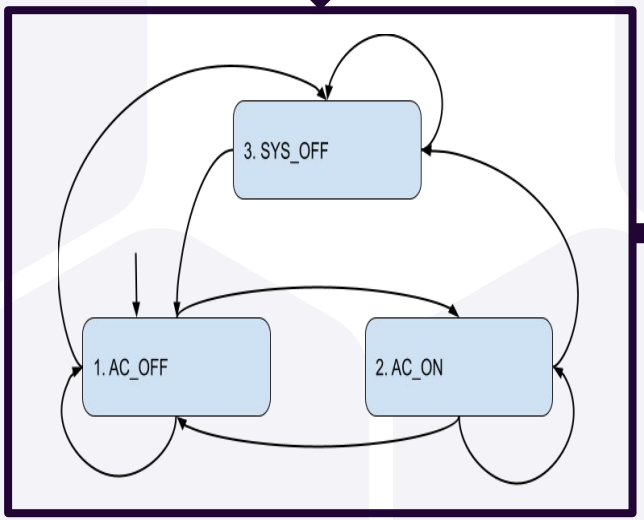
For a transition system *S*, Create a *safety monitor FSM* called *M* where:

- inputs of *M* are a subset of the inputs and outputs of system
- Some subset *E* of the states of *M* are designated as “error” states
- The behavior of *M* is designed such that if the sequence of inputs to *M* leads *M* to an error state in *E*, this is an invariant violation

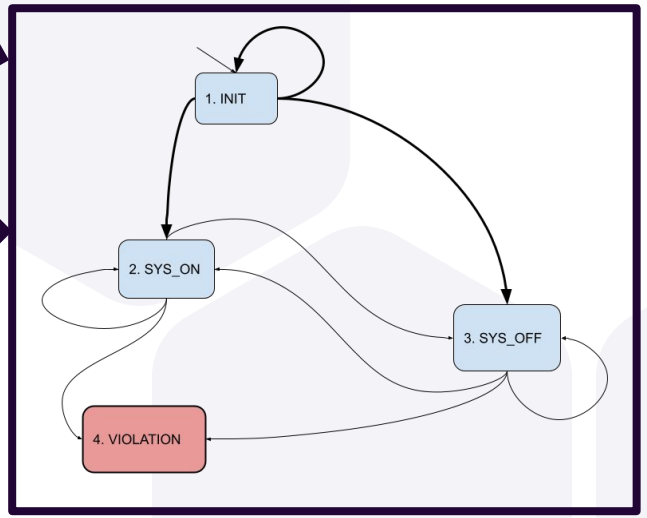
Compose *M* and *S*. The invariant becomes that any state in *E* is not reachable

*What similarities do you see between the safety monitor FSM definition and the runtime monitor you wrote in lab 8?*

on/off button
current_temp
desired_temp
mils

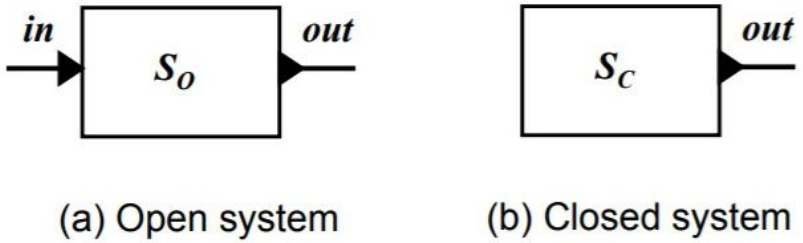



status_msg
AC LED



# Open and closed systems

To automate invariant verification, we need to work with a closed system




(a) Open system (b) Closed system

Figure 15.1: Open and closed systems.

*[Lee/Seshia, chapter 15]*

# Automated verification of invariants

Create a closed system by composing model of the system with model of the environment

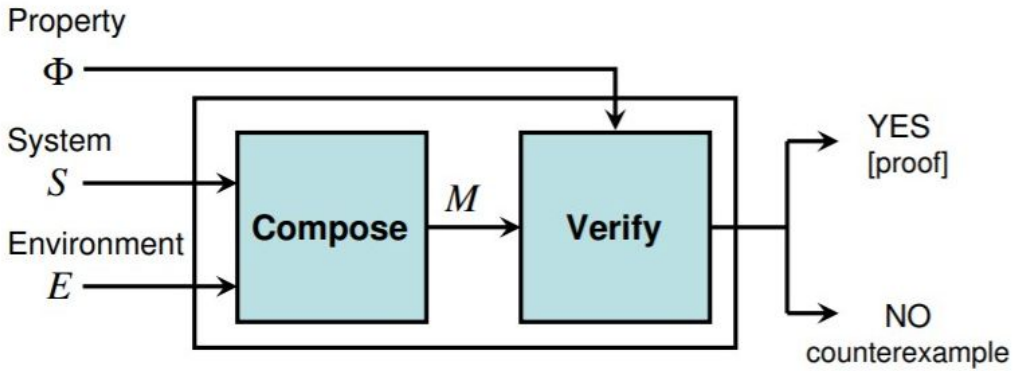



Figure 15.2: Formal verification procedure.

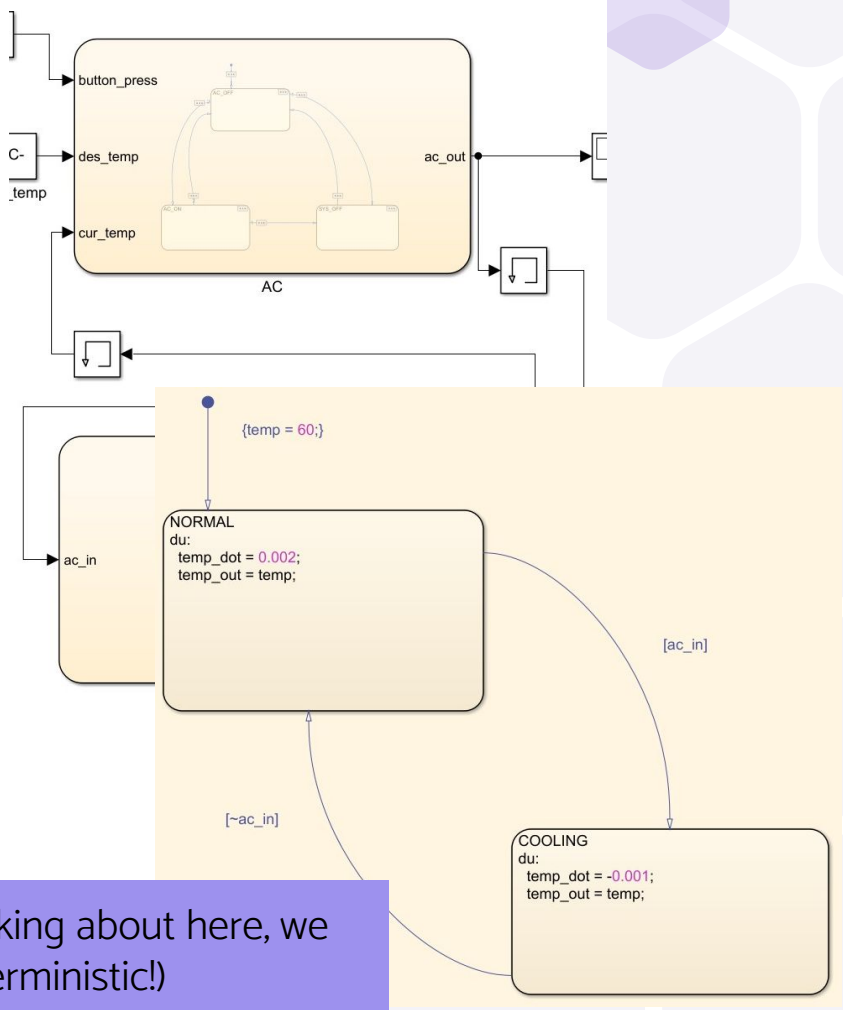
*[Lee/Seshia, chapter 15]*

# Closed AC model

System: AC FSM

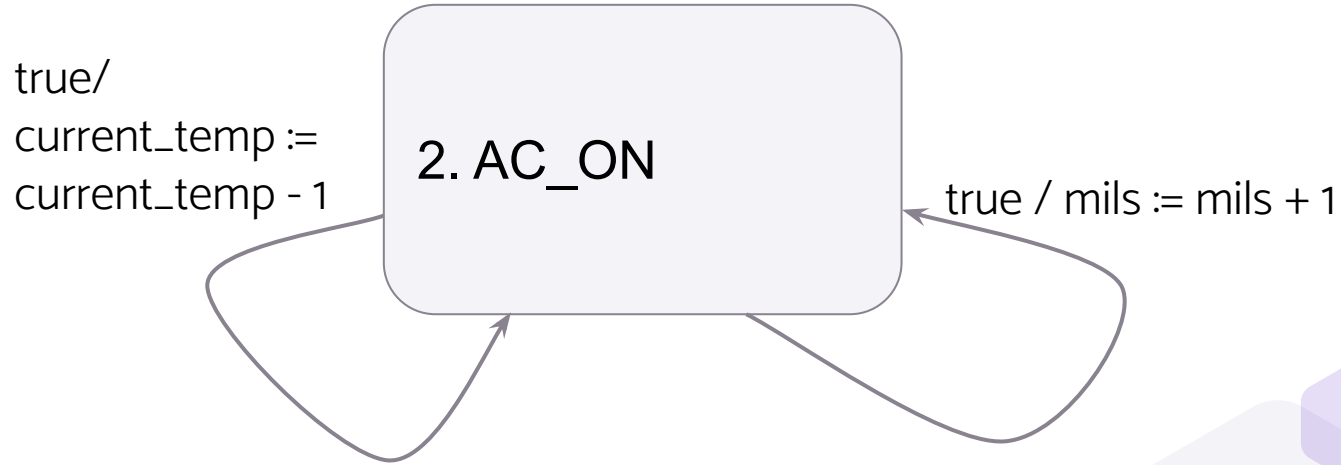
Environment:

- Time
- Button
- Current temp
- Desired temp



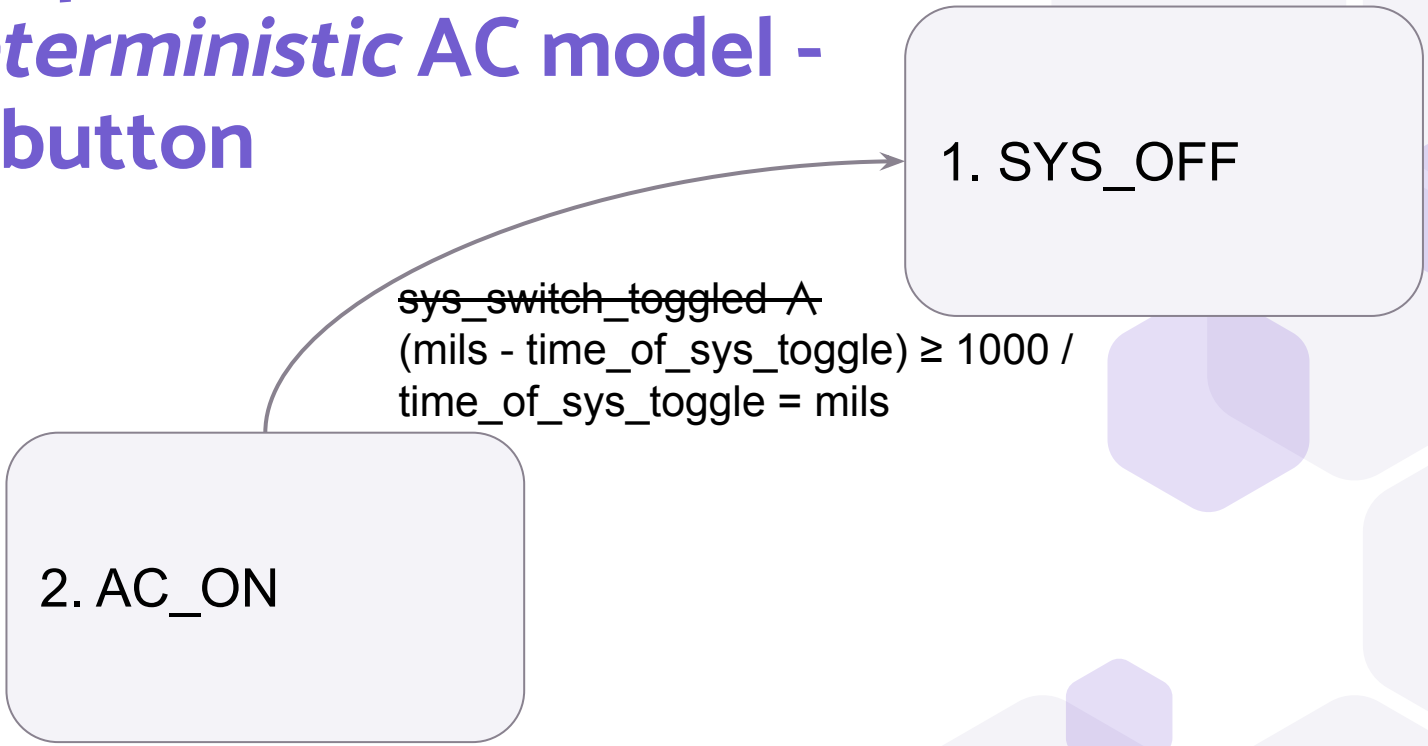

Note: for the logics/computation models we are talking about here, we are using *discrete* systems (but not necessarily deterministic!)

# Very simplified closed *discrete, non-deterministic* AC model - current temp and time

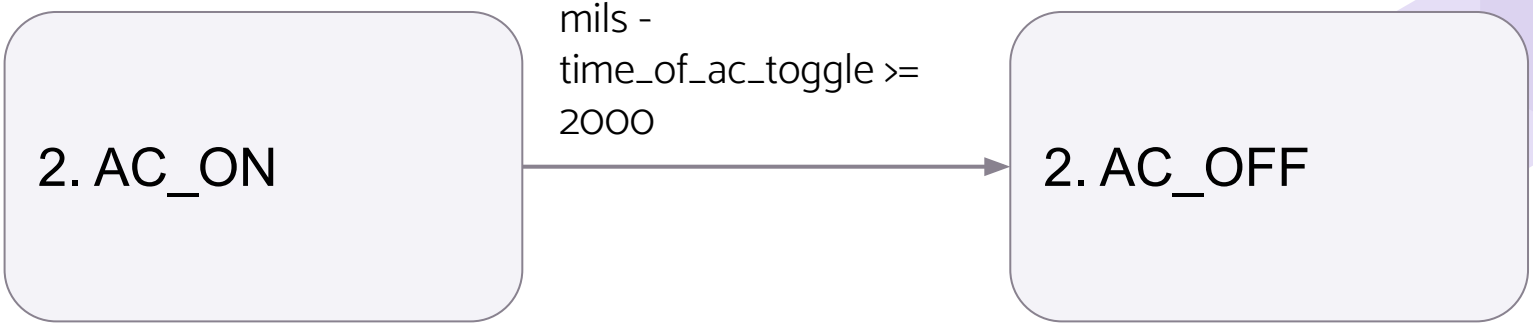


*What do we lose when switching to a discrete model?*

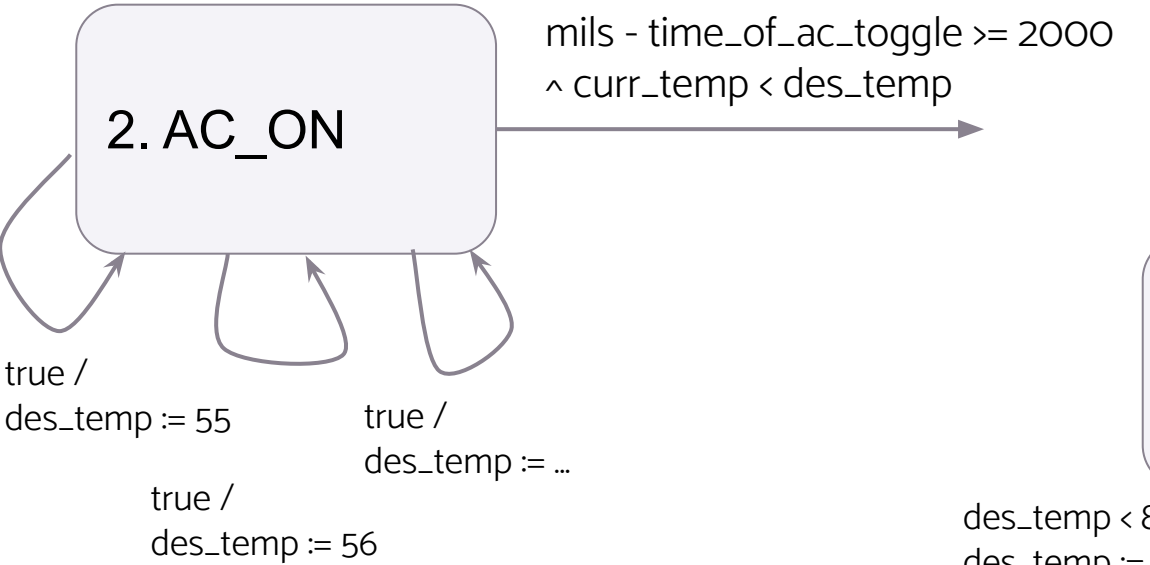
# Very simplified closed *discrete, non-deterministic* AC model - on/off button

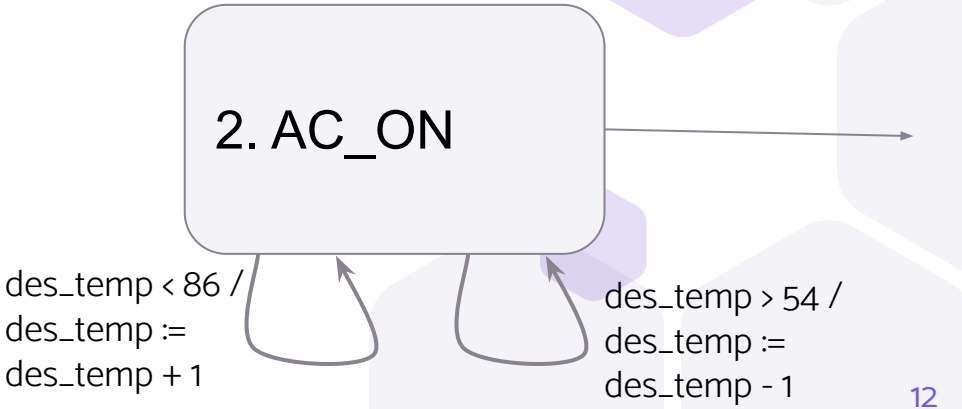


# Very simplified closed *discrete, non-deterministic* AC model - desired temp (option 1)



# Very simplified closed *discrete, non-deterministic* AC model - desired temp (options 2 and 3)



# Automated reachability analysis

A property *p* of a transition system* *S* is an *invariant* of *S* if every **reachable** state of *S* satisfies *p*

How would you automatically determine the set of reachable states?

Assume a system of finite states

(Verification for a system of infinite states is *undecidable)*

# Depth-first search

**Input** : Initial state $s_0$ and transition relation $\delta$ for closed finite-state system $M$
**Output**: Set $R$ of reachable states of $M$

```
1  Initialize: Stack Σ to contain a single state s0; Current set of reached
   states R := {s0}.
2  DFS_Search() {
3  while Stack Σ is not empty do
4  |   Pop the state s at the top of Σ
5  |   Compute δ(s), the set of all states reachable from s in one
   |   transition
6  |   for each s' ∈ δ(s) do
7  |   |   if s' ∉ R then
8  |   |   |   R := R ∪ {s'}
9  |   |   |   Push s' onto Σ
10 |   |   end
11 |   end
12 end
13 }
```

**Algorithm 15.1:** Computing the reachable state set by depth-first explicit-state search.

*[Lee/Seshia, chapter 15]*

# DFS board example for AC

*How would you modify the DFS algorithm to either produce a “YES” or a counterexample for a property p?*

# Reference for DFS question

**Input** : Initial state $s_0$ and transition relation $\delta$ for closed finite-state system $M$
**Output**: Set $R$ of reachable states of $M$

1 **Initialize:** *Stack* $\Sigma$ *to contain a single state* $s_0$*; Current set of reached states* $R := \{s_0\}$.
2 **DFS_Search**() {
3 **while** *Stack* $\Sigma$ *is not empty* **do**
4 Pop the state $s$ at the top of $\Sigma$
5 Compute $\delta(s)$, the set of all states reachable from $s$ in one transition
6 **for** *each* $s' \in \delta(s)$ **do**
7 **if** $s' \notin R$ **then**
8 $R := R \cup \{s'\}$
9 Push $s'$ onto $\Sigma$
10 **end**
11 **end**
12 **end**
13 }

**Algorithm 15.1:** Computing the reachable state set by depth-first explicit-state search.

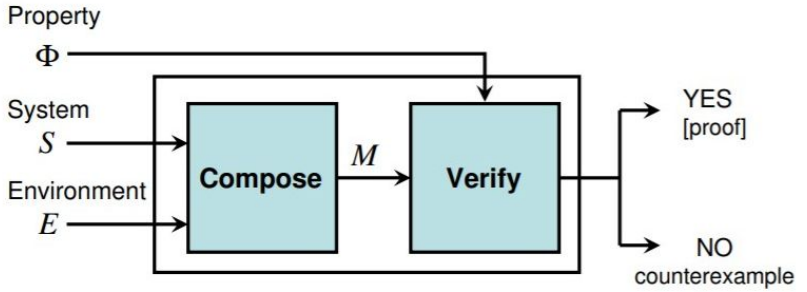



Figure 15.2: Formal verification procedure.

# Safety requirements vs liveness requirements

**Safety**: nothing bad *ever* happens

**Liveness**: something good *eventually* happens

- Means system is functioning as intended
- System requirements are often liveness requirements

*What are some liveness requirements for the AC?*

*How would you **monitor** that a liveness requirement is fulfilled?*

# Verifying some liveness properties

Saying something *eventually* happens is the same thing as saying that it is *not* the case that it always *doesn't* happen

# Linear Temporal Logic (LTL)

Assume you have *some* execution trace

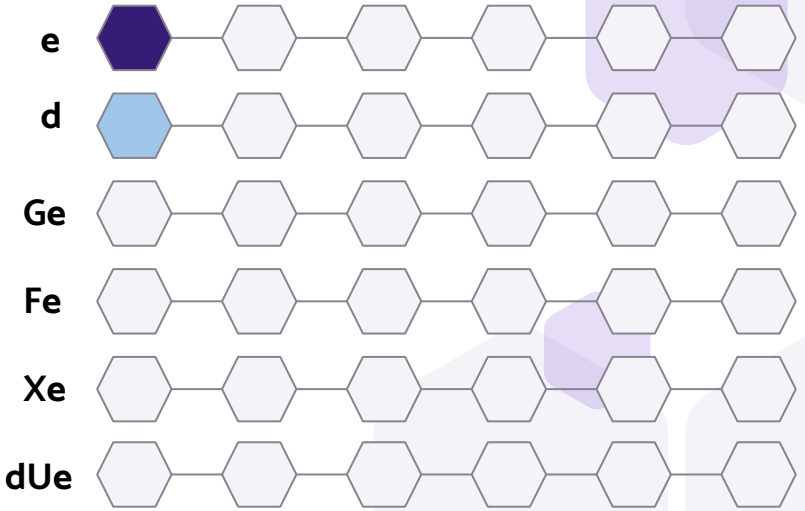
LTL operators are propositional logic operators PLUS:

G (globally/always)

F (eventually/finally)

X (next state)

U (until)



# FG vs G, GF vs F, FG vs GF