Verification and invariants

Final project demo

Friday the 3rd!

Looking to find a bigger room, stay tuned

Walk-around demo (no formal presentation)

Produce a flyer or slide outlining the goals of the product (user-facing or product requirements)

Q & A by course staff (convince us you did what you said you would)

Final project report

Due date options?

Instructions + syllabus posted by this weekend

Initial feedback + time for revisions before final assessment

Individual component (+ possible meeting during reading period)

Capstone folks required to meet during reading period

What are some limitations of software testing?

Safety properties and invariants

Invariant: some computable property of a system that always holds (*more precise definition later*)

Safety property (or safety requirement): assertion that nothing bad ever happens

How are invariants and safety properties related?

Safety properties can be expressed as invariants

Define “bad thing” computably

Invariant: bad thing is not true

Example for AC from lab 8

1) *There is no more than 290 ms of delay between status_message messages.*

→ “bad thing”: two consecutive status_message messages come more than 290 ms apart

→ invariant: bad thing is not true

→ your monitor checked if the invariant always held

Another example from lab 8

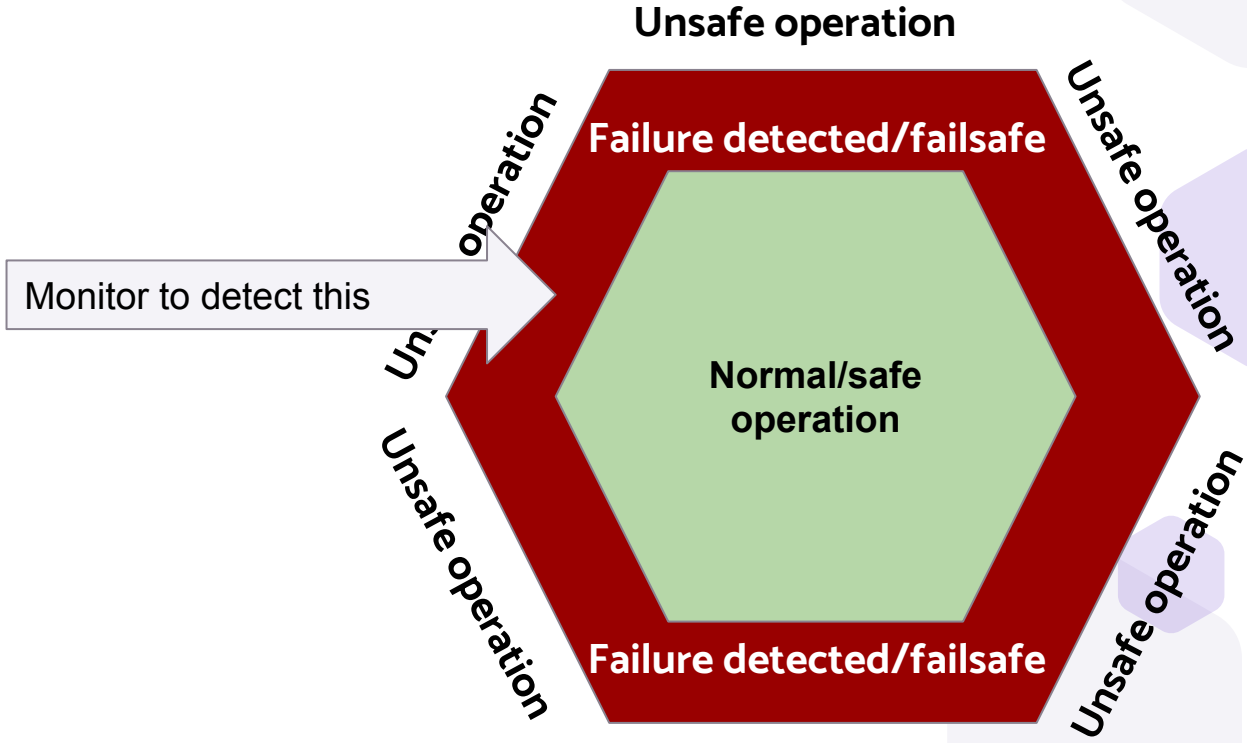
10) If the system is on, the control knob hasn't changed for at least 2450 ms, and the current temperature has been lower than the desired temperature for at least 2450 ms, the AC status is off as sent by status_message.

What is the "bad thing?"

Working with invariants

- Runtime monitoring on a deployed system
- Testing
- ...formally proving?

Runtime monitoring on a deployed system



Invariants for testing

Our basic understanding of testing so far has been largely **transactional:**

- Give input, observe that output matches what is expected

Are embedded systems transactional?

- Robot asked to navigate to a goal point

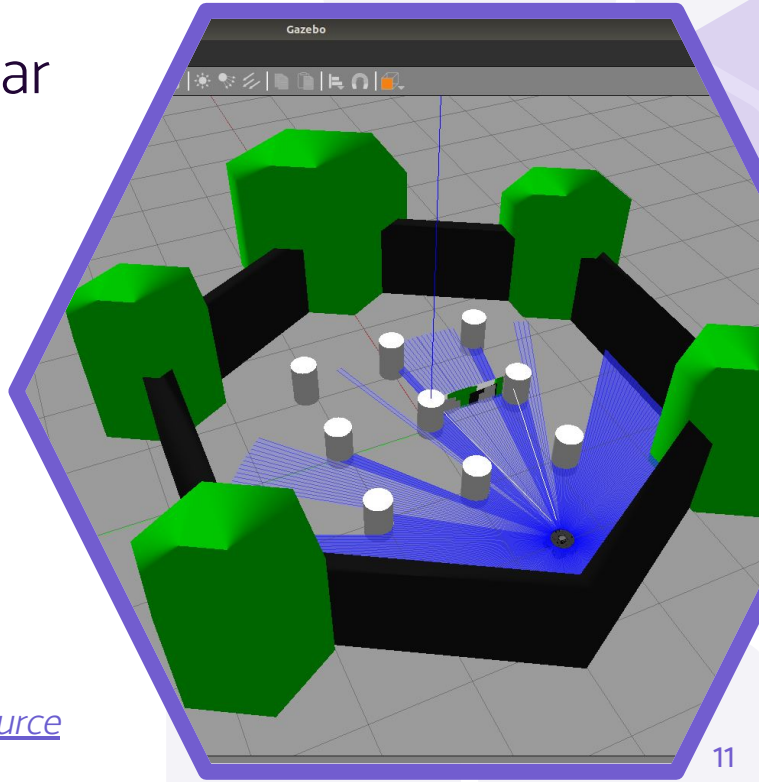


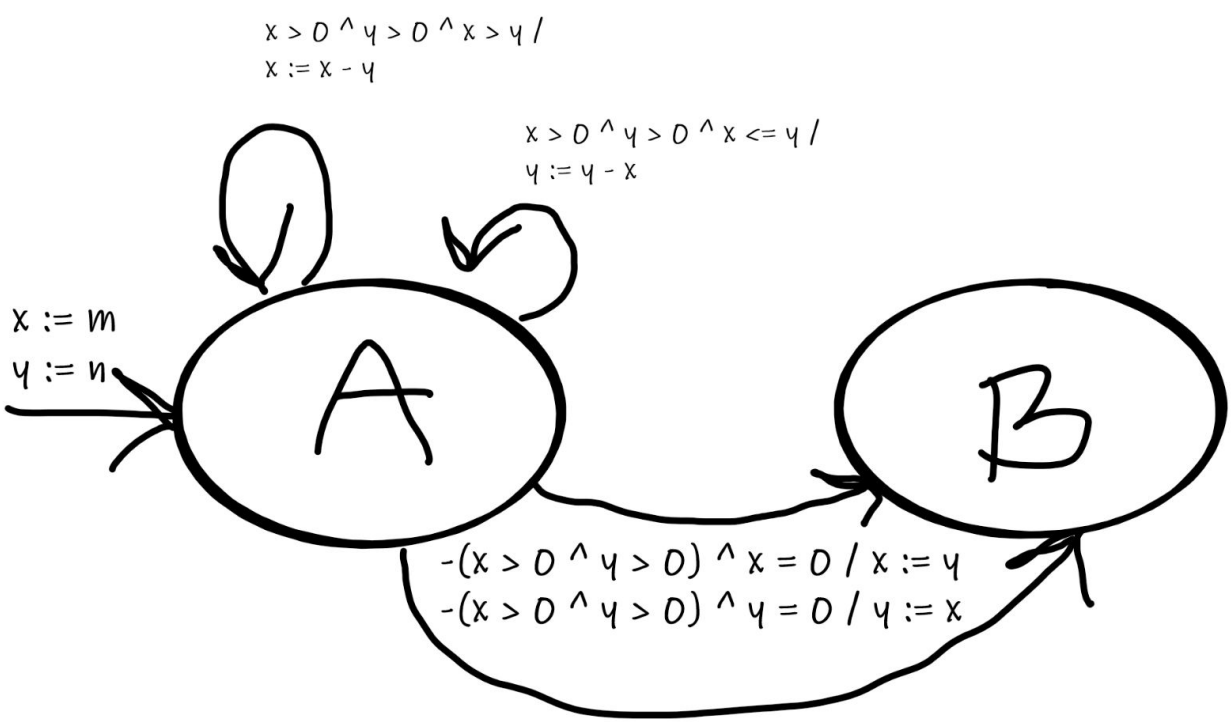
Image source

Formalizing invariants

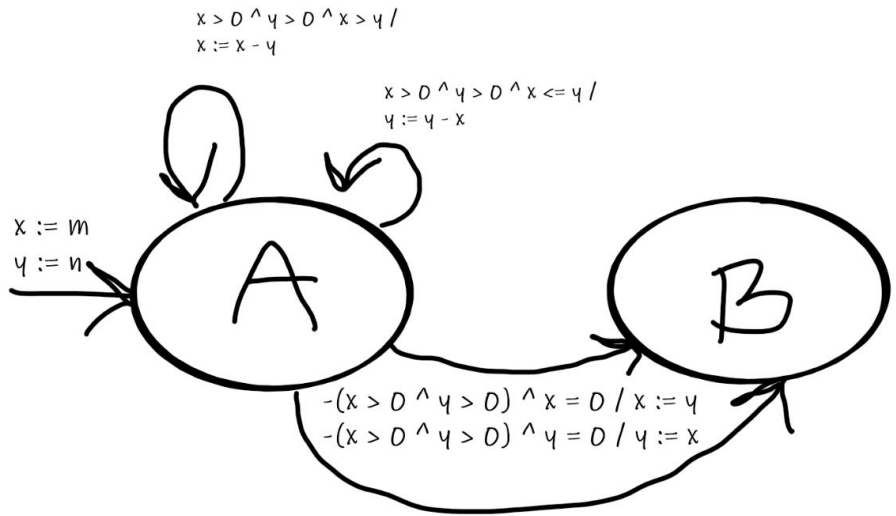
...back to FSMs!

Board discussion:

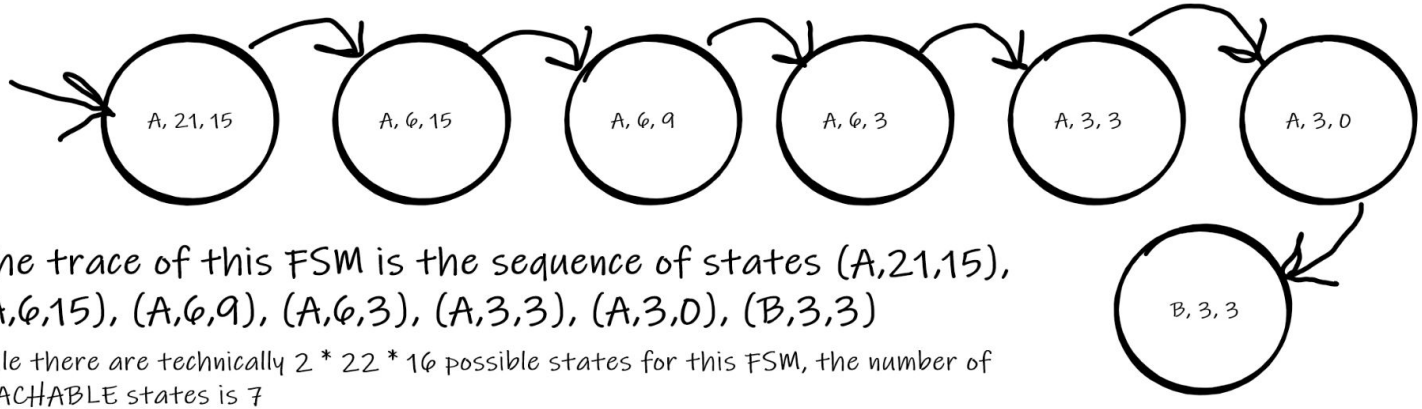
- Traces
- Defining states for an ESM
- Reachability



This FSM is for a *fixed* pair of positive numbers m and n (m and n are constants, not inputs) Remember that variables are a shorthand way of describing state. If x and y are assumed to range from 0 to m and 0 to n, respectively, the number of actual states in this FSM is 2*(m +1)*(n+1)



Represent each state as a tuple ({A or B, x, y). Then for m = 21 and y = 15, the FSM actually looks like:



The trace of this FSM is the sequence of states (A,21,15), (A,6,15), (A,6,9), (A,6,3), (A,3,3), (A,3,0), (B,3,3)

While there are technically 2 * 22 * 16 possible states for this FSM, the number of REACHABLE states is 7

Propositional logic

Composed of terms (“a”, “b”, “c”), where a term can be:

p(x), q(x), r(x,y): propositions (evaluate to either true or false)

x > 0

x + y = 2

robot x has not hit obstacle y

a ^ b : a and b (true if term a is true and term b is true)

a v b: a or b (true if term a is true or term b is true or both)

-a: not a (true if term a is false)

a => b: a implies b (true if term b is true or if term a is false)

Formal definition of an invariant

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

**For our class, think of a transition system as an FSM*

[Alur, chapter 3]

Inductive invariants

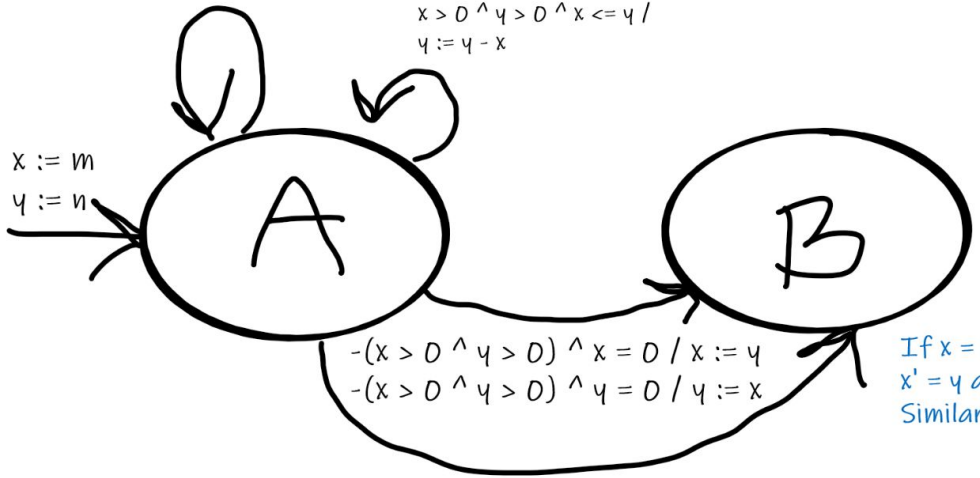
A property *p* of a transition system *S* is an *inductive invariant of S* if:

1. The initial state *s* satisfies *p*, and
2. If a state *s* satisfies *p*, and (*s*, *t*) is a transition, then the state *t* also satisfies *p*

(Board discussion)

x > y (due to transition)
x - y > 0
x' = x - y > 0
y' = y > 0 (due to transition)

y >= x (due to transition)
y - x >= 0
y' = y - x >= 0
x' = x > 0 (due to transition)



If x = 0, then y > 0 due to inductive assumption
x' = y and y' = y, so both are positive
Similar for lower transition

Inductively prove that (x >= 0 ^ y >= 0) ^ (x > 0 v y > 0) (x and y are non-negative and at least one of them is non-zero)

Base case: yes (m > 0 and n > 0)
Inductive case: assume property holds for x and y, show that each of the four transitions causes it to keep holding for new values of x and y, x' and y'

Proving non-inductive invariants

To establish that a property *p* is an invariant of the transition system *S*, find a property *q* that:

1. *q* is an inductive invariant of *S*, and
2. the property *q* implies the property *p* (that is, a state satisfying *q* is guaranteed to satisfy *p*)

(Board discussion)

Non-inductive invariants

Prove B => x > 0 ∧ y > 0 inductively?

Cannot do this

Need to bring along the previous property we proved, which will imply this one

How would you deal with this invariant?

10) If the system is on, the control knob hasn't changed for at least 2450 ms, and the current temperature has been lower than the desired temperature for at least 2450 ms, the AC status is off as sent by status_message.

Stateful invariants

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

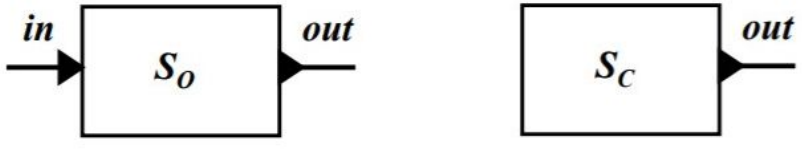
- inputs of *M* are a subset of the inputs and outputs of system
- Some subset *E* of the states of *M* are designated as “error” states
- The behavior of *M* is designed such that if the sequence of inputs to *M* leads *M* to an error state in *E*, this is an invariant violation

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?

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

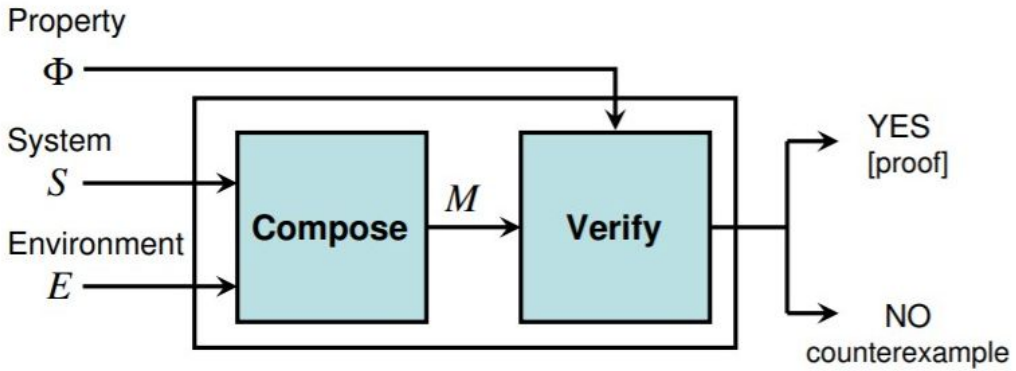
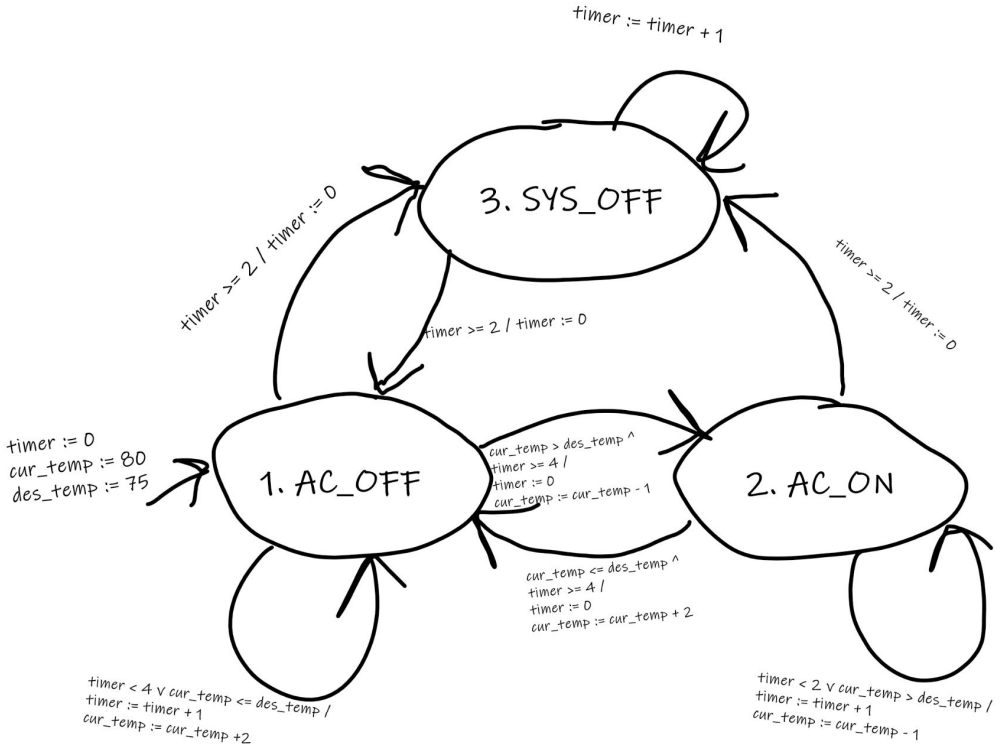


Figure 15.2: Formal verification procedure.

[Lee/Seshia, chapter 15]

Simplified closed AC model

Observe use of non-determinism to represent that button can be pressed at any time



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 δ for closed finite-state system M
Output: Set R of reachable states of M

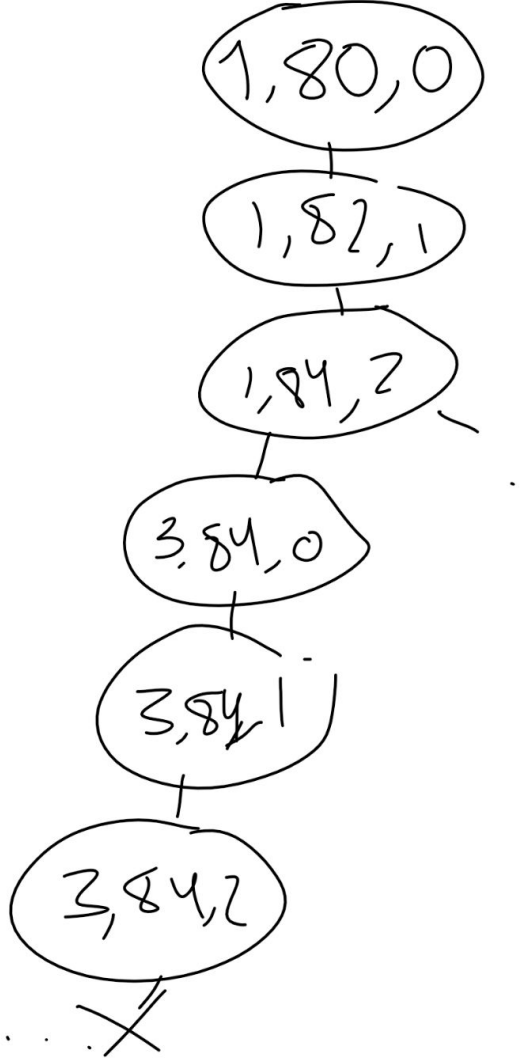
```
1  Initialize: Stack Σ to contain a single state s0; Current set of reached
   states R := {s0}.
2  DFS_Search() {
3  while Stack Σ is not empty do
4  |   Pop the state s at the top of Σ
5  |   Compute δ(s), the set of all states reachable from s in one
   |   transition
6  |   for each s' ∈ δ(s) 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

(timer variable needs to be limited to avoid infinite traversal down path)



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 δ for closed finite-state system M
Output: Set R of reachable states of M

1 **Initialize:** *Stack* Σ *to contain a single state* s_0*; Current set of reached states* $R := \{s_0\}$.
2 **DFS_Search**() {
3 **while** *Stack* Σ *is not empty* **do**
4 Pop the state s at the top of Σ
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 Σ
10 **end**
11 **end**
12 **end**
13 }

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

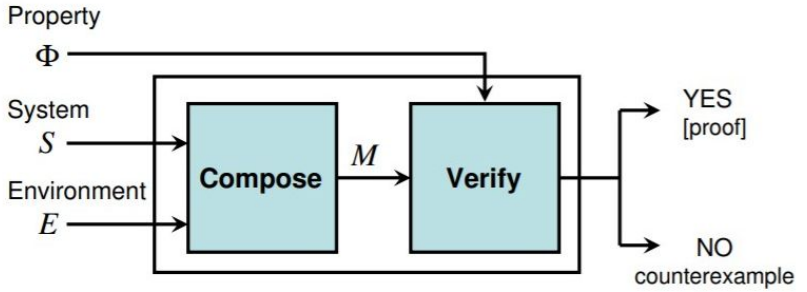


Figure 15.2: Formal verification procedure.

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

Can you come up with a liveness requirement 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

Can we use model checking to check this?

Linear Temporal Logic (LTL)

Assume you have *some* execution trace

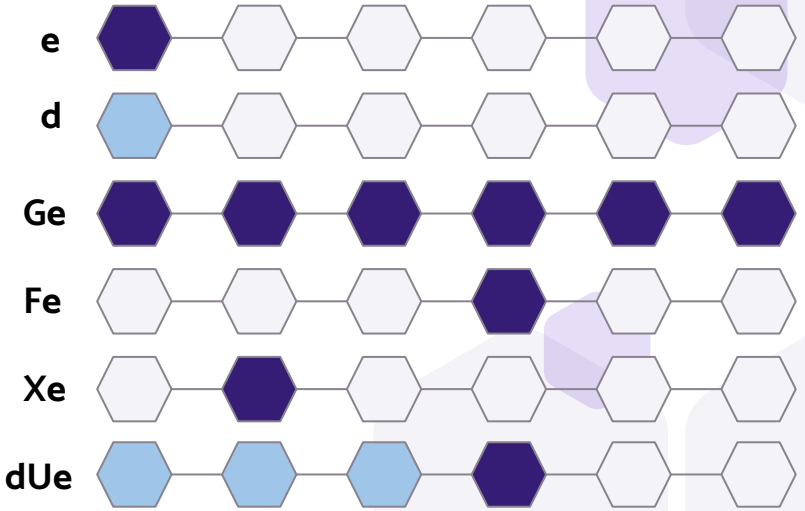
LTL operators are propositional logic operators PLUS:

G (globally/always)

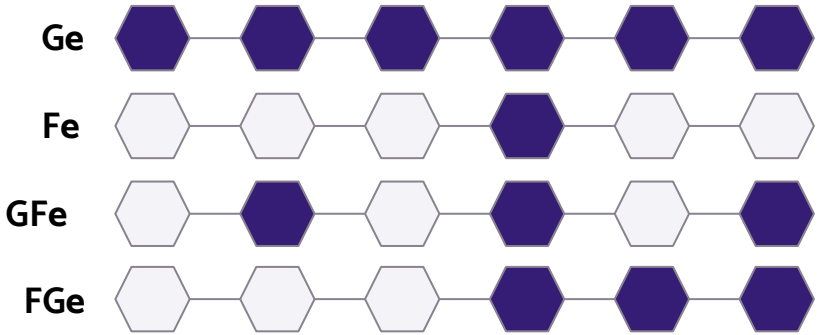
F (eventually/finally)

X (next state)

U (until)



FG vs G, GF vs F, FG vs GF



37

Homework problem discussion

LTL means we can specify liveness properties with F. Can we specify safety properties more easily with LTL, too?

Safety properties with LTL

Use globally to say it holds for every state

Can use “X” to express statefulness/history without a monitor state machine

Limits of LTL

G(even(x) -> ((X-even(x)) ∧ (XXeven(x))

But if you don't know if you started a sequence with an odd number or even number, you cannot write

(even(x) ∧ X-even(x))

Repeatability

A property *p* over the state variables of a transition system *S* is said to be *repeatable* if there exists *some* trace *q* of *S* such that *q* satisfies the recurrence LTL-formula *GFp*.

Buchi automata

Automata which “accept” a given LTL formula

(see Alur’s textbook for examples)

Automated LTL verification

Buchi automata construction can be automated (discussion in optional reading)

Compose Buchi automata of negated property with system and produce a counter-example of repeatability to prove property

- Done using a DFS + cycle detection (“nested DFS”)

More verification techniques

Automated verification

Symbolic model checking: represent a set of states symbolically as a logic formula and does symbolic (algebraic) computation

What about timed/hybrid automata?

Symbolic model checking for a different kind of logic (signal temporal logic)

Assisted proof engines (differential dynamic logic)

Summary: pros/cons of verification?