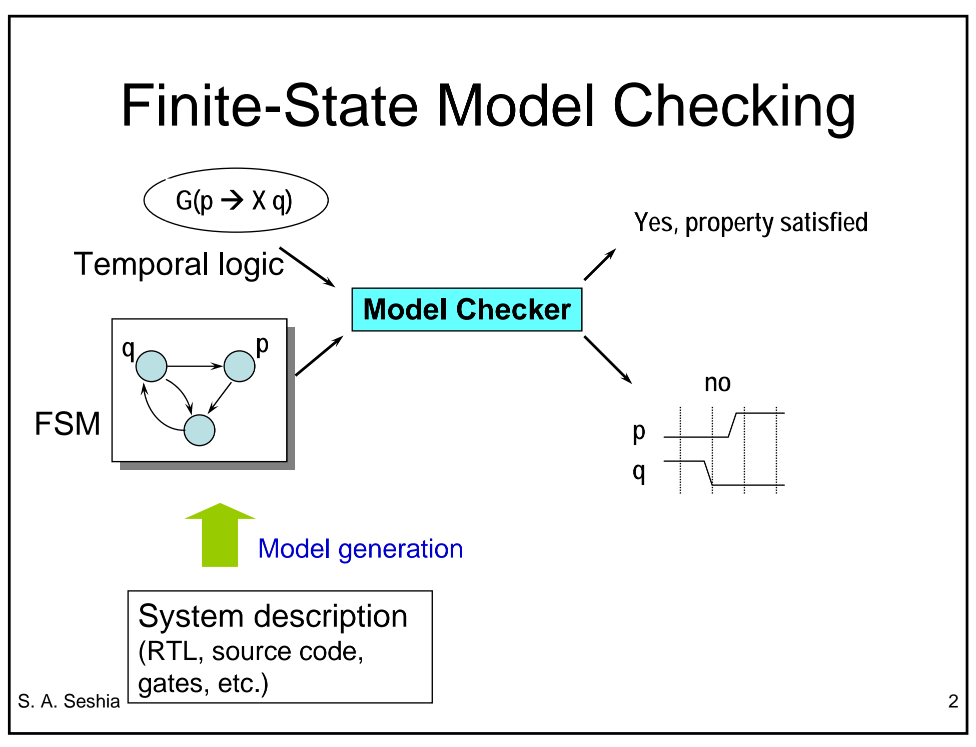
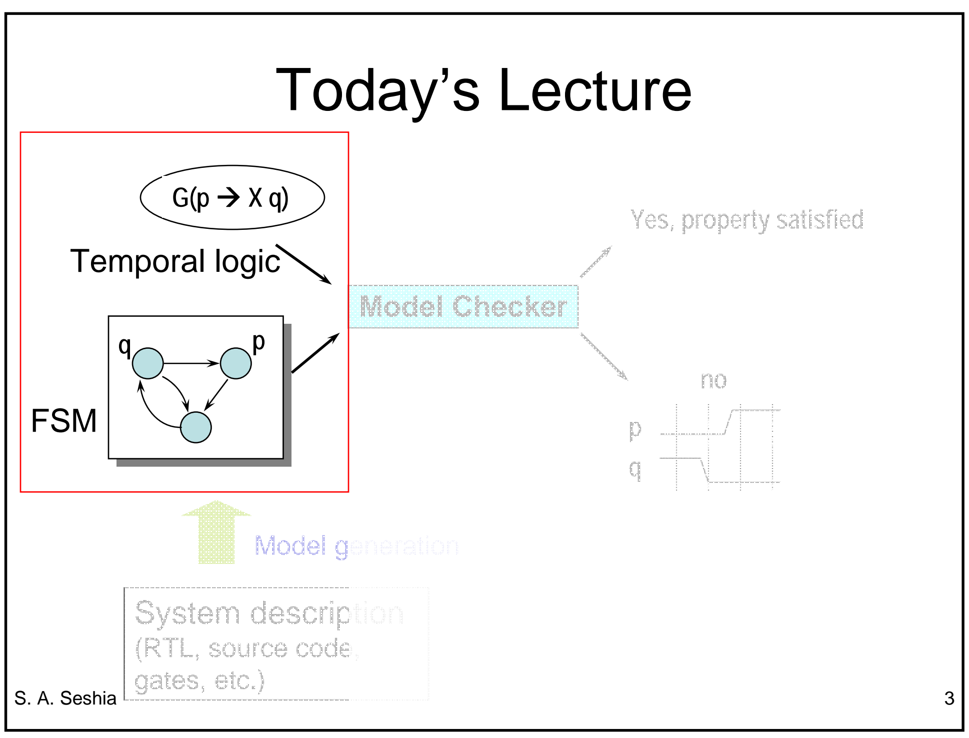
EECS 219C: Computer-Aided Verification

Intro. to Model Checking: Models and Properties

Sanjit A. Seshia

EECS, UC Berkeley





2 Kinds of Systems

1. Open
2. Closed

- What's the difference between the two?

S. A. Seshia 4

Verifying Closed Systems

- Assumes we have models of
  - System
  - Environment (a “good enough” one)
- Overall model is the composition of the system with its environment
- This will be the topic for most of this course

Questions addressed in this lecture

- What is a model?
- How to compose two models together?
- How to express properties of a model?

Modeling Finite-State Machines

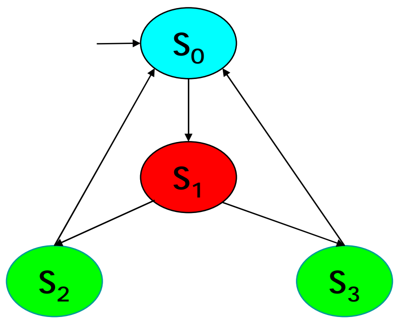
- Remember, it's a closed system – i.e., no inputs
- Common representation:
  - (S, S_0, R)
- Why do we need a *transition relation* and not just a transition function?
- Representation in practice:
  - (V, S_0, R)

Kripke Structure

- Alternative way of representing closed finite-state models

  (S, S_0, R, L)
  - S → set of states
  - S_0 → set of initial states
  - R → transition relation
    - must be *total:* for every state s, there exists state s', s.t. R(s,s')
  - L → labeling function
    - labels a state with a set of "atomic propositions" (think of these as "colors")

Example of Kripke Structure



Why should we use Kripke structures?

Why Kripke Structures?

- Representation is independent of state-encoding
- Captures notion of "observability" to relate to actual executions
  - an observer might not be able to read all state variables

Composition

- Typically the overall system is specified as a set of modules, and the environment
  - Assume we have a Kripke structure for each
- There are two basic ways of constructing the overall Kripke structure
  - Synchronous composition
  - Asynchronous composition

How to Compose?

- Synchronous Composition
  - All components in the system change their state variables simultaneously
- Asynchronous Composition
  - At each time point, one component changes its state
- Which form of composition exhibits more behaviors?

Synchronous Product

- Given two Kripke structures
  - $M1 = (S1, s1_0, R1, L1)$
  - $M2 = (S2, s2_0, R2, L2)$
- Sync. Product is $M = (S, s_0, R, L)$
  - $S \subseteq S1 \times S2$
  - $s_0 = (s1_0, s2_0)$
  - $R = R1 \wedge R2$
  - $L(s1, s2) = (L1(s1), L2(s2))$

Asynchronous Product

(interleaving semantics)

- Given two Kripke structures
  - $M1 = (S1, s1_0, R1, L1)$
  - $M2 = (S2, s2_0, R2, L2)$
- Async. Product is $M = (S, s_0, R, L)$
  - $S \subseteq S1 \times S2$
  - $s_0 = (s1_0, s2_0)$
  - $R(s, s') = \ (R1(s1,s1') \wedge s2' = s2) \vee \ (R2(s2,s2') \wedge s1' = s1)$
  - $L(s1, s2) = (L1(s1), L2(s2))$

Example: Interrupt-Driven S/W

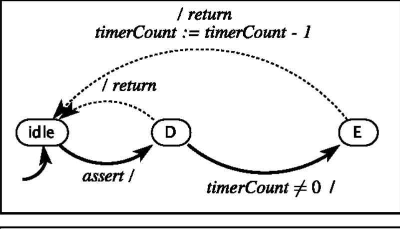
```
    volatile uint timerCount = 0;
    void ISR(void) {
D →    … disable interrupts
E →    if(timerCount != 0) {
          timerCount--;
       }
       … enable interrupts
    }
    int main(void) {
       // initialization code
       SysTickIntRegister(&ISR);
       … // other init
       timerCount = 2000;
A →    while(timerCount != 0) {
B →     … code to run for 2 seconds
       }
C → … whatever comes next
    }
```

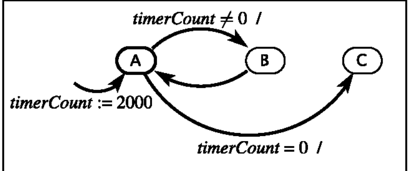
A key question: Assuming interrupt can occur infinitely often, is position C always reached?

State machine model

```
    volatile uint timerCount = 0;
    void ISR(void) {
D →    … disable interrupts
E →    if(timerCount != 0) {
          timerCount--;
       }
       … enable interrupts
    }
    int main(void) {
       // initialization code
       SysTickIntRegister(&ISR);
       … // other init
       timerCount = 2000;
A →    while(timerCount != 0) {
B →     … code to run for 2 seconds
       }
C → … whatever comes next
    }
```

variables: *timerCount*: uint
input: *assert*: pure
output: *return*: pure





Which form of composition is the right thing to do here?

Recap

- We're verifying closed systems
- Modeled as Kripke structures (S, S_0, R, L)
  - Represents the product of the "system" with its "environment"

Specifying Properties

- Ideally, want a complete specification
  - Implementation must be equivalent to the specification w.r.t. observable state
- In practice, only have partial specifications
  - Specify some "good" behaviors and some "bad" behaviors

What's a Behavior?

- Define in terms of states and transitions
- A sequence of states, starting with an initial state
  - s_0 s_1 s_2 … such that $R(s_i, s_{i+1})$ is true
- Also called "run", or "(computation) path"
- Trace: sequence of observable parts of states
  - Sequence of state labels

Safety vs. Liveness

- Safety property
  - "something bad must not happen"
  - E.g.: system should not crash
  - finite-length error trace
- Liveness property
  - "something good must happen"
  - E.g.: every packet sent must be received at its destination
  - infinite-length error trace

Examples: Safety or Liveness?

1. "No more than one processor (in a multi-processor system) should have a cache line in write mode"

2. "The grant signal must be asserted at some time after the request signal is asserted"

3. "Every request signal must receive an acknowledge and the request should stay asserted until the acknowledge signal is received"

Temporal Logic

- A logic for specifying properties over time
  - E.g., Behavior of a finite-state system
- We will study *propositional* temporal logic
  - Other temporal logics exist:
    - e.g., real-time temporal logic

Atomic State Property (Label)

A Boolean formula over state variables

We will denote each unique Boolean formula by
• a distinct color
• a name such as p, q, …

req

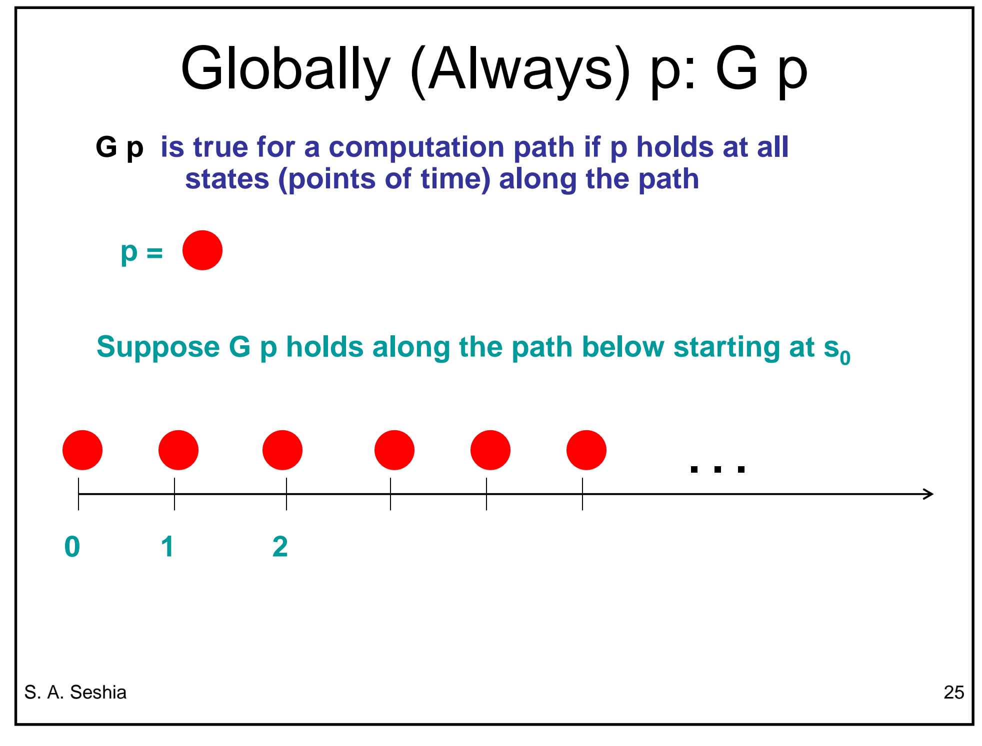
req & !ack

S. A. Seshia

24

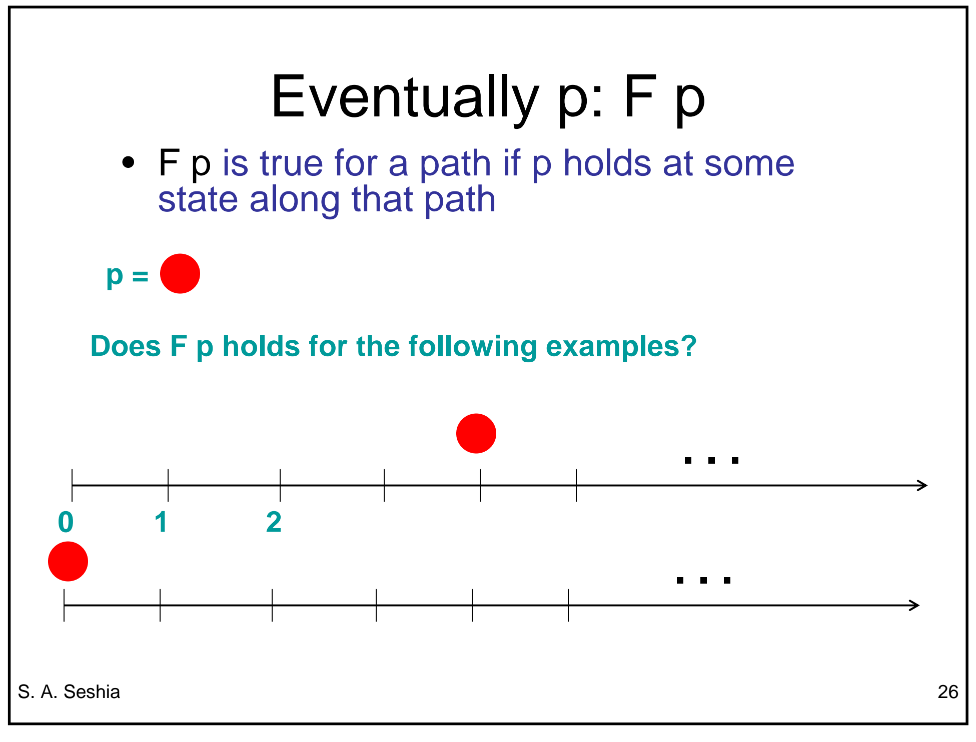
Globally (Always) p: G p

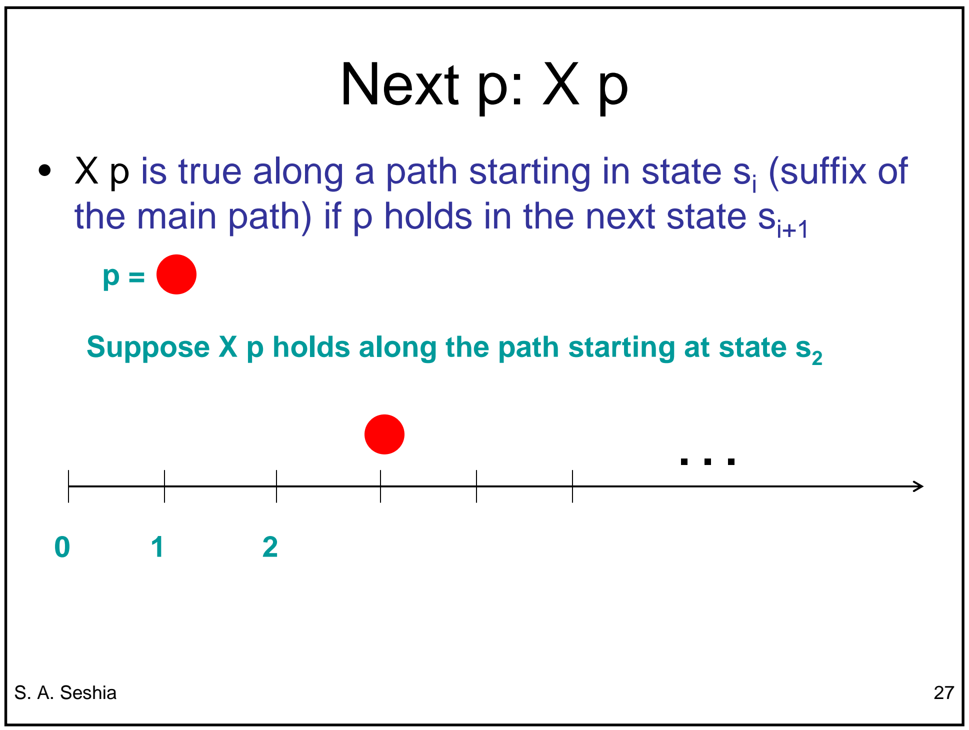
G p is true for a computation path if p holds at all states (points of time) along the path



S. A. Seshia

25


Eventually p: F p
• F p is true for a path if p holds at some state along that path
p =
Does F p holds for the following examples?
. . .
0
1
2
. . .
S. A. Seshia
26


Next p: X p
• X p is true along a path starting in state s_i (suffix of the main path) if p holds in the next state s_{i+1}
p =
Suppose X p holds along the path starting at state s_2
. . .
0
1
2
S. A. Seshia
27

Nesting of Formulas

- p need not be just a Boolean formula.
- It can be a temporal logic formula itself!

p =

"X p holds for all suffixes of a path"

How do we draw this?

How can we write this in temporal logic?

Write down formal definitions of Gp, Fp, Xp

Notation

- Sometimes you'll see alternative notation in the literature:

  G □

  F ◇

  X ○

Examples: What do they mean?

- G F p
- F G p
- G(p → F q)
- F(p → (X X q))

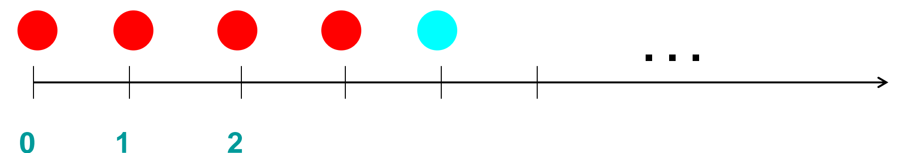
S. A. Seshia 30

p Until q: p U q

- p U q is true along a path starting at s if
  - q is true in some state reachable from s
  - p is true in all states from s until q holds

p = ● q = ●

Suppose p U q holds for the path below



0 1 2

S. A. Seshia 31

Temporal Operators & Relationships

- G, F, X, U: All express properties along paths
- Can you express G p purely in terms of F, p, and Boolean operators ?
- How about G and F in terms of U and Boolean operators?
- What about X in terms of G, F, U, and Boolean operators?

Examples in Temporal Logic

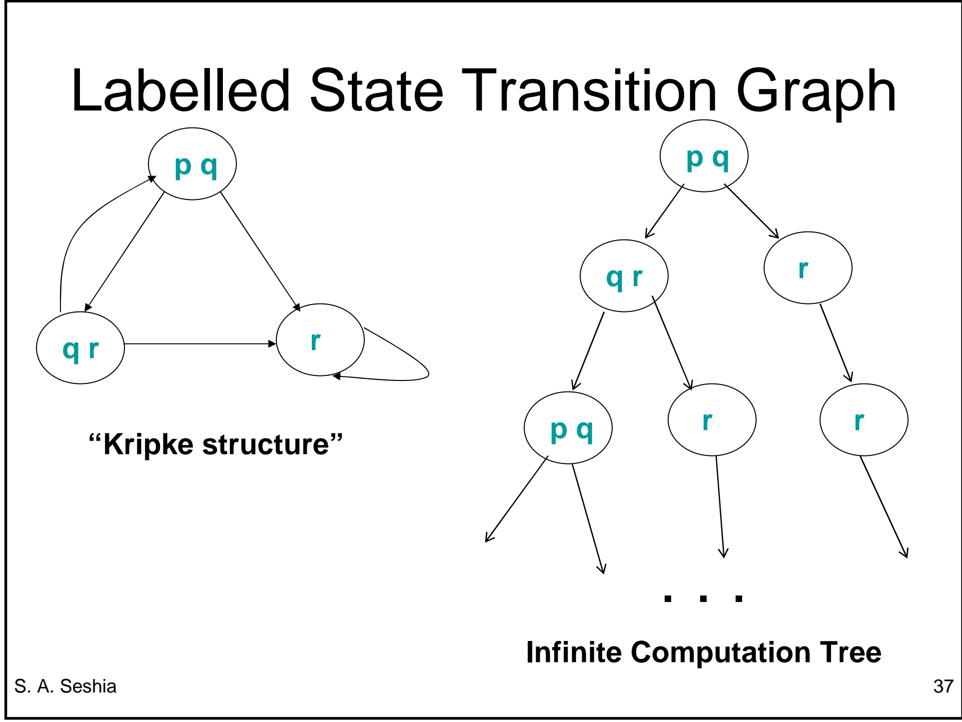
1. "No more than one processor (in a 2-processor system) should have a cache line in write mode"
   - wr_1 / wr_2 are respectively true if processor 1 / 2 has the line in write mode
2. "The grant signal must be asserted at some time after the request signal is asserted"
   - Signals: grant, req
3. "Every request signal must receive an acknowledge and the request should stay asserted until the acknowledge signal is received"
   - Signals: req, ack

Linear Temporal Logic

- What we've seen so far are properties expressed over a single computation path or run
  - LTL

Temporal Logic Flavors

- Linear Temporal Logic
- Computation Tree Logic
  - Properties expressed over a tree of all possible executions
  - Where does this "tree" come from?



Temporal Logic Flavors

- Linear Temporal Logic (LTL)
- Computation Tree Logic (CTL, CTL*)
  - Properties expressed over a tree of all possible executions
  - CTL* gives more expressiveness than LTL
  - CTL is a subset of CTL* that is easier to verify than arbitrary CTL*

Computation Tree Logic (CTL*)

- Introduce two new operators A and E called "Path quantifiers"
  - Corresponding properties hold in states (not paths)
  - A p : Property p holds along all computation paths starting from the state where A p holds
  - E p : Property p holds along at least one path starting from the state where E p holds
- Example:

  "The grant signal must always be asserted some time after the request signal is asserted"

  A G (req → A F grant)

- Notation: A sometimes written as ∀, E as ∃

S. A. Seshia 39

CTL

- Every F, G, X, U must be immediately preceded by either an A or a E
  - E.g., Can't write A (FG p)
- LTL is just like having an "A" on the outside

S. A. Seshia 40

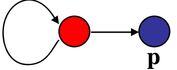
Why CTL?

- Verifying LTL properties turns out to be computationally harder than CTL
- But LTL is more intuitive to write
- Complexity of model checking
  - Exponential in the size of the LTL expression
  - linear for CTL
- For both, model checking is linear in the size of the state graph

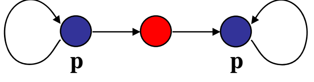
S. A. Seshia 41

CTL as a way to approximate LTL

– AG EF p is weaker than G F p **Useful for finding bugs...**



– AF AG p is stronger than F G p



Useful for verifying correctness...

Why? And what good is this approximation?

S. A. Seshia 42

More CTL

- "From any state, it is possible to get to the reset state along some path"

A G (E F reset)

CTL vs. LTL Summary

- Have different expressive powers
- Overall: LTL is easier for people to understand, hence more commonly used in property specification languages

Some Remarks on Temporal Logic

- The vast majority of properties are safety properties
- Liveness properties are useful abstractions of more complicated safety properties (such as real-time response constraints)

S. A. Seshia 45

(Absence of) Deadlock

- An oft-cited property, especially people building distributed / concurrent systems
- Can you express it in terms of
  - a property of the state graph (graph of all reachable states)?
  - a CTL property?
  - a LTL property?

S. A. Seshia 46

Summary

- What we've done so far:
  - Modeling with Kripke structures
  - Sync/Async Composition
  - Properties in Temporal Logic, LTL, CTL, CTL*