Software Model Checking

Linear Temporal Logic

Moonzoo Kim

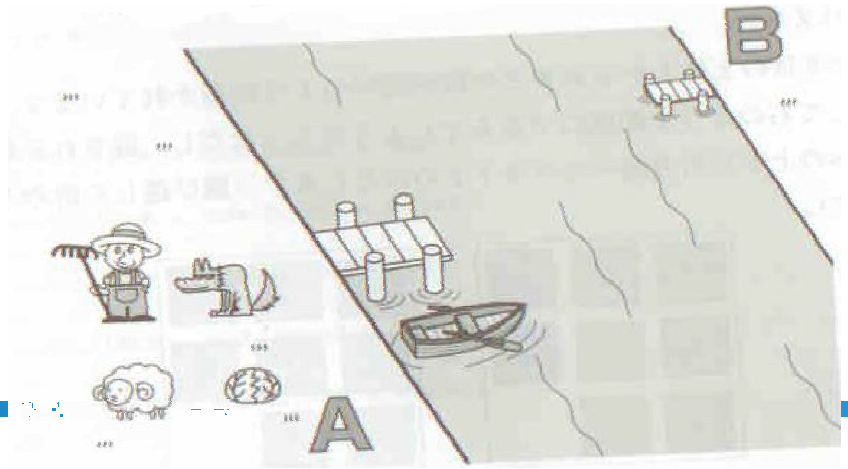
CS Dept. KAIST

Comments on #3 of HW#1

- Do not blindly repeat patterns you know!
  - Squeeze out your specification to reduce # of state and transition
- Comparison
  - 7 smart people in the class modeled Peterson's mutual exclusion protocol including a process **`NumCrit`**
    - # of states: 70, # of transition: 130
  - One smarter person darely removed **`NumCrit`**
    - # of states: 35, # of transition:70

HW#2 due Oct 10

- Wolf, ram, and cabbage problem
  - Goal: move wolf, ram and cabbage to B Restriction:
    - Boat can deliver only 1 object with a person
    - Ram eats cabbage and wolf eats ram if a person is not with them
  - To do:
    - 1. Model this system in CCS
      - When all of wolf, ram, and cabbage are delivered to B, the system should generate “finish” action and halt
    - 2. Show that there exists a solution to the problem using simulation feature of CWB-NC
    - 3. Show that there exists a solution using GCTL* model checking capability
    - 4. Show that there exists a solution using proper equivalence semantics



HW#2 (cont.)

- RW problem of lecture 5
  - RW system shown in lecture 5 does not faithfully model Java thread behavior. Point out difference between the model used in RW system and real Java thread behavior
    - 1.Read “Threads” chapter of “The Java Programming Language” by K.Arnold and J.Gosling
      - Optionally RW explanation in “Concurrent Programming in Java” by D.Lea
    - 2. Analyze given RW specification in CCS
    - 3. Explain difference between the model and real Java threads
    - Hints: carefully looking at thread suspension, scheduling, wait, and notify mechanism
  - Modify the model to reflect real Java thread behavior

semantics

given a state sequence (from a run σ):

$s_0, s_1, s_2, s_3 \ \ldots$

and a set of propositional symbols: `p,q,…` such that

$\forall i, (i \geq 0)$ **and** $\forall p,\ s_i \models p$ is defined

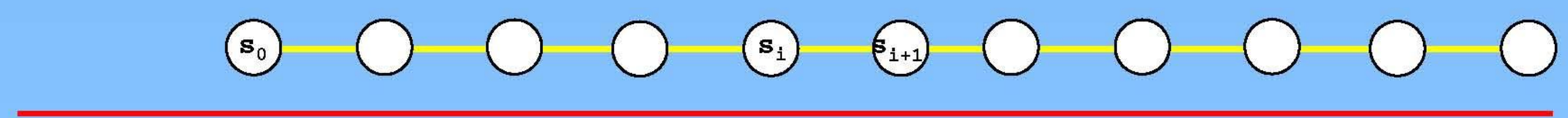
we can define the semantics of the temporal logic formulae:

`[]f, <>f, Xf, and e U f`

$\sigma \models f$	iff	$s_0 \models f$

`i.e., the property holds for the remainder of run σ, starting at position` s_0

$s_i \models \ []f$	iff	$\forall j, (j >= i): s_j \models f$
$s_i \models \ <>f$	iff	$\exists j, (j >= i): s_j \models f$
$s_i \models Xf$	iff	$s_{i+1} \models f$



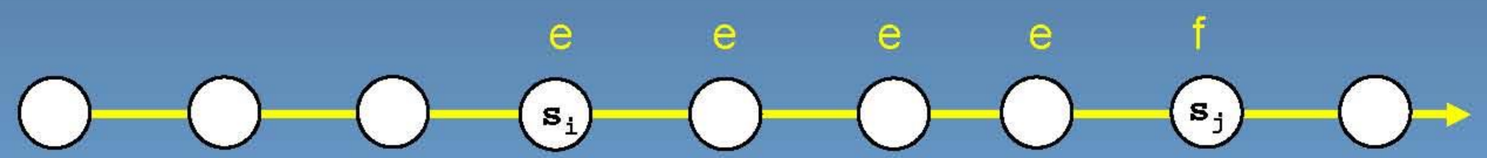
weak and strong until

(cf. book p. 135-136)

weak until

$s_i \models e \; U \; f$ iff

$s_i \models f \vee (s_i \models e \wedge s_{i+1} \models (e \; U \; f))$



strong until (Spin)

$s_i \models e \; U \; f$ iff

$\exists j, (j >= i): s_j \models f$ and

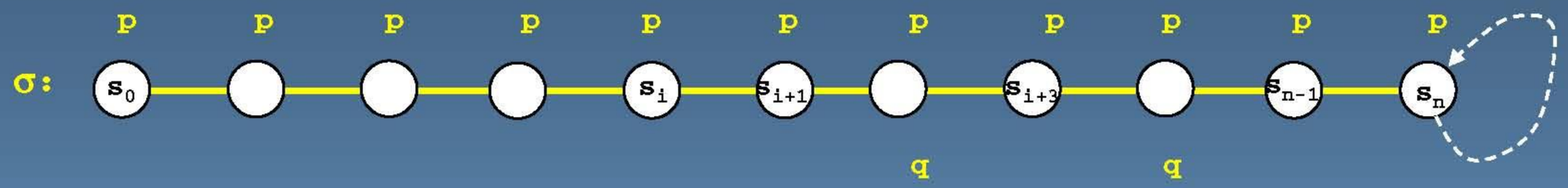
$\forall k, (i <= k < j): s_k \models e$

equivalences:

$(e \; U \; f) == (e \; U \; f) \wedge (<> \; f)$

$(e \; U \; f) == (e \; U \; f) \vee ([] \; e)$

examples



[]p is satified at all locations in σ

<>p is satisfied at all locations in σ

[]<>p is satisfied at all locations in σ

<>q is satisfied at all locations except s_{n-1} and s_n

Xq is satisfied at s_{i+1} and at s_{i+3}

pUq (strong until) is satisfied at all locations except s_{n-1} and s_n

<>(pUq) (strong until) is satisfied at all locations except s_{n-1} and s_n

<>(pUq) (weak until) is satisfied at all locations

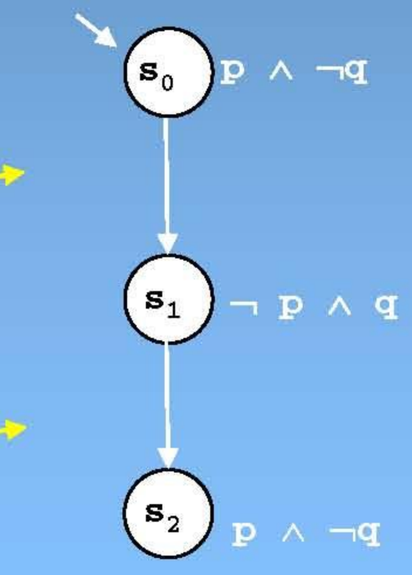
[]<>(pUq) (weak until) is satisfied at all locations

in model checking we are typically only interested in whether a temporal logic formula is satisfied for all runs of the system, starting in the initial system state (that is: at s_0)

equivalences

(cf. book p. 137)

- `[]` p ↔ (p U false) — weak until
- <>p ↔ (true U p) — strong until
- !`[]`p ↔ <>!p
  - if p is not invariantly true, then eventually p becomes false
- !<>p ↔ `[]`!p
  - if p does not eventually become true, it is invariantly false
- `[]`p && `[]`q ↔ `[]` (p && q)
  - note though: (`[]` p || `[]` q) → `[]` (p || q)
  - but: (`[]`p || `[]` q) ← `[]` (p || q)
- <>p || <> q ↔ <> (p || q)
  - note though: (<> p && <> q) ← <> (p && q)
  - but: (<> p && <> q) → <> (p && q)



some standard LTL formulae

`[]` p	always p	invariance
<> p	eventually p	guarantee
p -> (<> q)	p implies eventually q	response
p -> (q U r)	p implies q until r	precedence
`[]`<> p	always, eventually p	recurrence (progress)
<>`[]` p	eventually, always p	stability (non-progress)
(<> p) -> (<> q)	eventually p implies eventually q	correlation

non-progress

acceptance

dual types of properties

in every run where p eventually becomes true q also eventually becomes true (though not necessarily in that order)

the earlier informally stated sample properties

(vugraph 12 lecture 11)

- p is invariantly true

  `[] p`

- p *eventually* becomes invariantly true

  `<>[] p`

- p *always eventually* becomes false at least once more

  `[]<>!p`

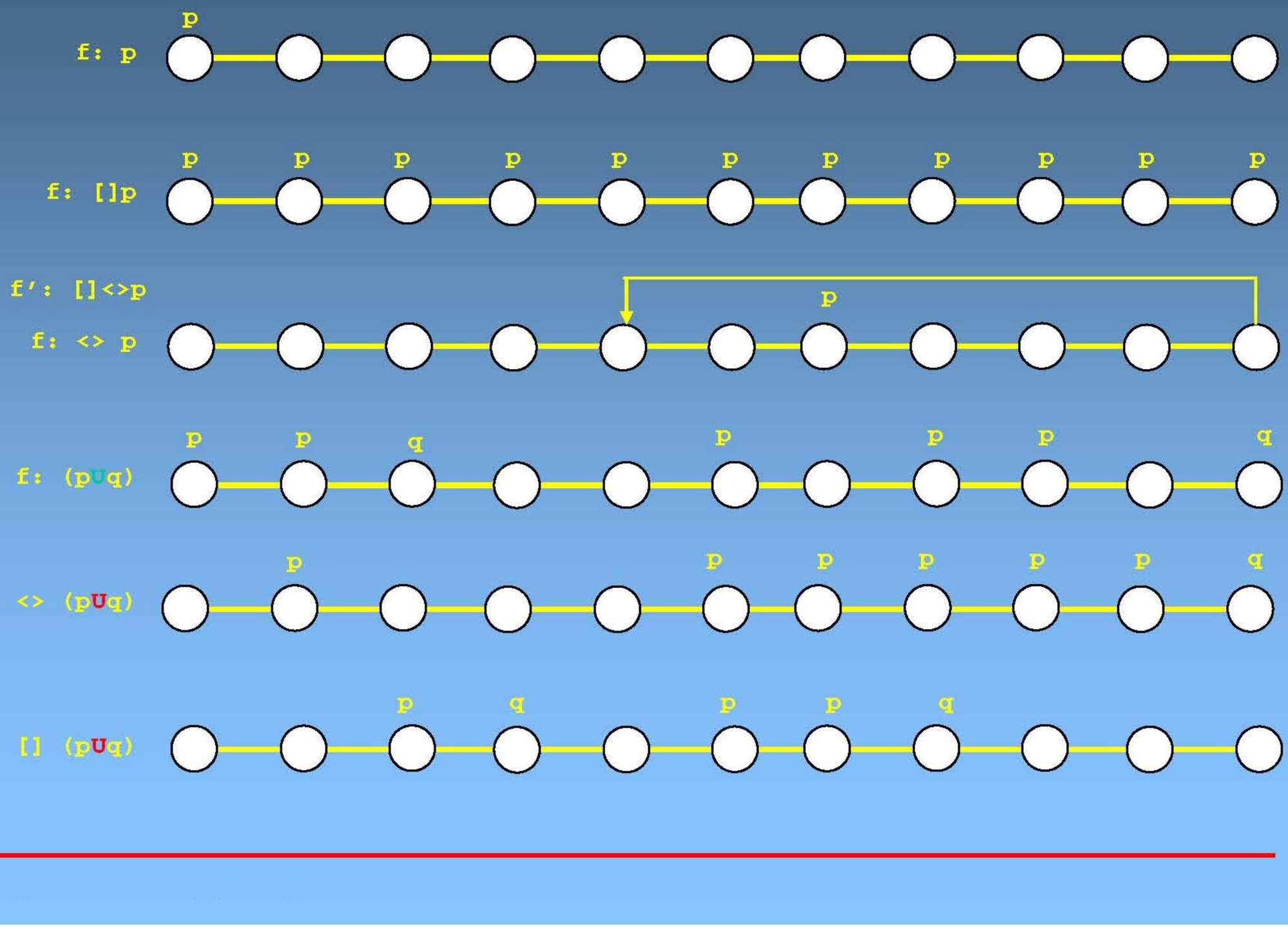
- p *always* implies ¬q

  `[] (p -> !q)`

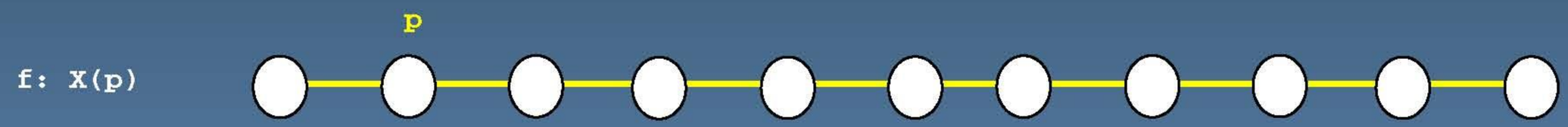
- p *always* implies *eventually* q

  `[] (p -> <> q)`

visualizing LTL formulae



the simplest operator: X



- the next operator X is part of LTL, but should be viewed with some suspicion
  - it makes a statement about what should be true in all possible *immediately* following states of a run
  - in distributed systems, this notion of 'next' is ambiguous
  - since it is unknown how statements are interleaved in time, it is unwise to build a proof that depends on specific scheduling decisions
    - the 'next' action could come from any one of a set of active processes – and could depend on relative speeds of execution
  - the only *safe* assumptions one can make in building correctness arguments about executions in distributed systems are those based on longer-term *fairness*

interpreting formulae...

LTL: (<>(b1 && (!b2 U b2))) -> []!a3

1. suppose b1 never becomes true
   (p->q) means (!p ∨ q)
   the formula is *satisfied!*

2. b1 becomes true, but not b2
   the formula is *satisfied!*

3. b1 becomes true, then b2
   but not a3
   the formula is *satisfied*

4. b1 becomes true, then b2, then a3
   the formula is *not satisfied*
   i.e., the property is violated

another example

LTL: (<>b1) -> (<>b2)

1. b1 never becomes true

   formula satisfied

2. b1 and b2 both become true

   formula satisfied

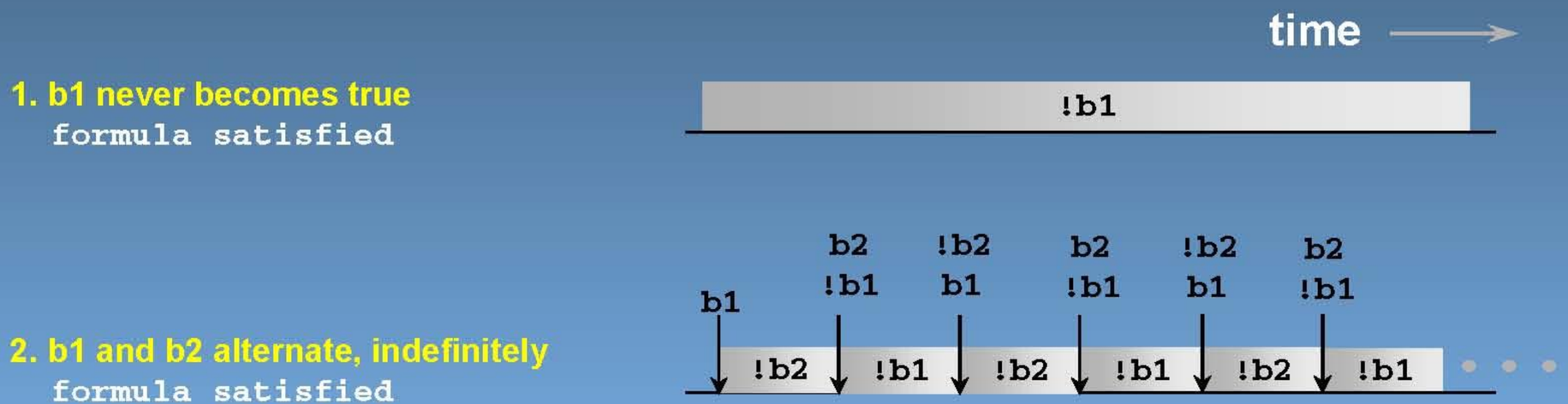
3. b1 becomes true but not b2

   formula *not* satisfied

   the property is *violated*

prefix the last formula with "[]"

LTL: [](<>b1) -> (<>b2))



where intuition can fail...

e.g., expressing the property: "p implies q"

- p -> q
  - not that there are no temporal operators ([], <>, U) in this formula -- it is a propositional formula (a *state* property) that will apply *only* to the *initial state* of each run...
  - the formula is immediately satisfied if (!p || q) is *true* in the initial system state – and the rest of the run is irrelevant

- []p -> q
  - beware of precedence rules...
  - as written this is parsed as ([]p) -> (q)
  - if p is not invariantly true, the formula is vacuously satisfied *(by the definition of ->,* "->" is *not* a temporal operator!*)*
  - if p is invariant, then the formula is satisfied *if q holds in the initial system state...*

expressing properties in LTL

"p implies q"

- [](p -> q)
  - note: there is still no temporal relation between p and q
  - this formula is satisfied if in every reachable state the propositional formula (!p || q) holds

- [](p -> <> q)
  - this would still be satisfied if p and q become true simultaneously, in one step (repeatedly)
  - doesn't capture the notion that somehow the truth of p *causes*, sometime later, the truth of q

expressing properties in LTL

"p implies q"

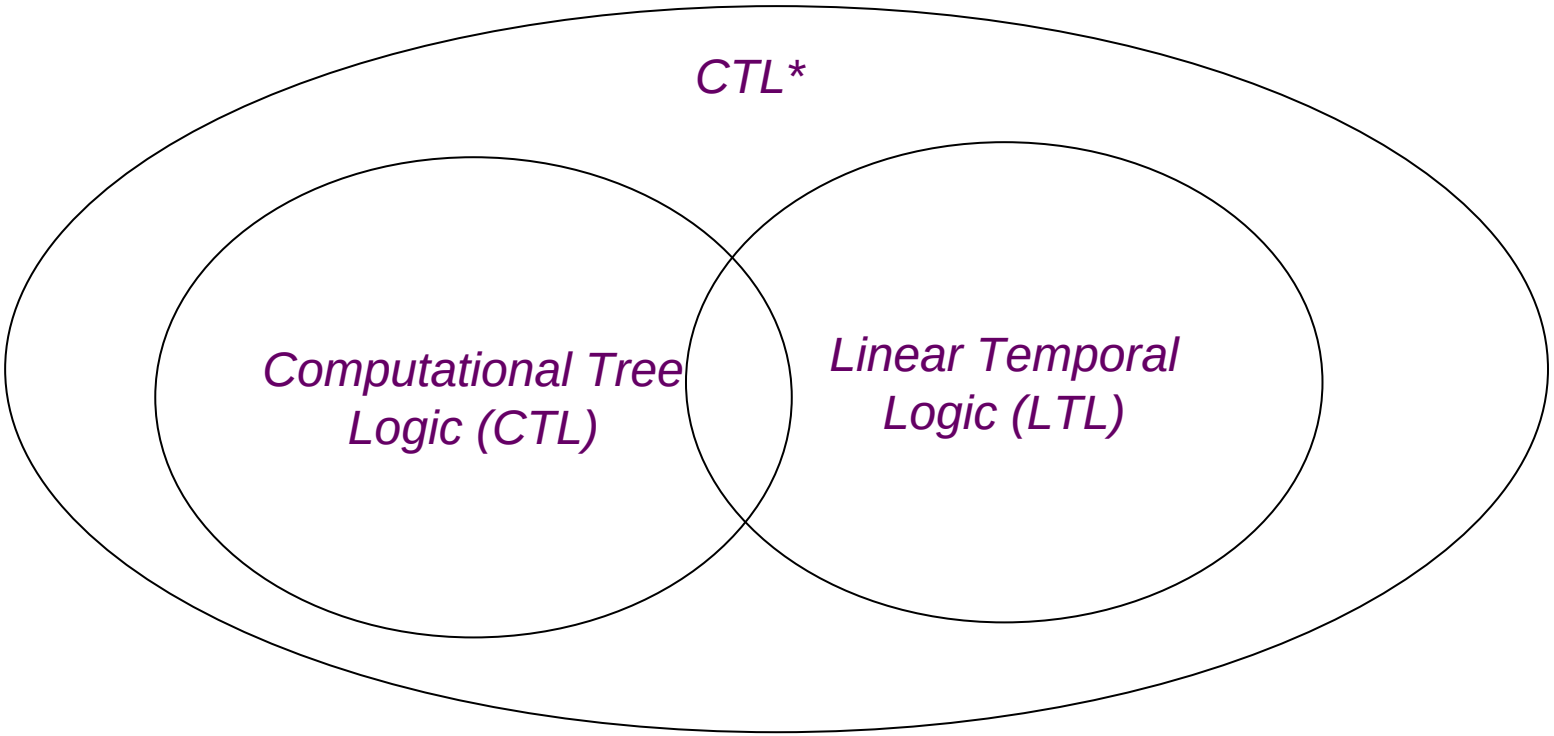
- [](p -> X(<>q))
  - puts one or more steps in between the truth of p and q, but this uses the maligned X operator... (but stutter invariance is maintained in this case)
  - formula is still satisfied if p *never becomes true*, probably not what is meant

- [](p -> X(<>q)) && (<>p)
  - this may actually capture what we intended
  - compare to our first guess of just: (p -> q)

beware of LTL
always double-check your formulae
be especially on guard when a model checker *fails* to find a matching run...

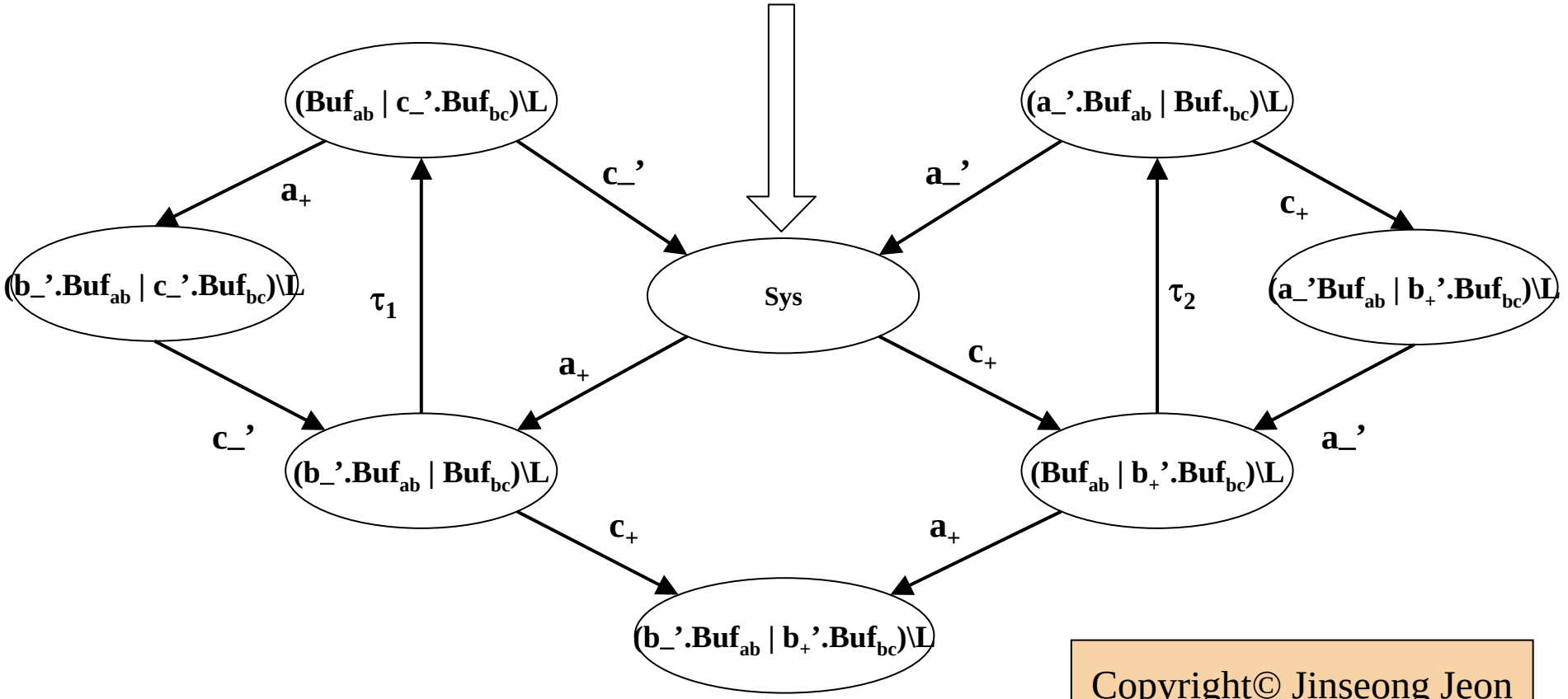
always use Spin to generate the never claim for each LTL formula, and study it to see if it matches your intuition of what you thought it should be...

Relationship among Various Temporal Logics



Combined Buffer Example in Lecture 3

$$Buf_{ab} = a_+.\overline{b_-}.Buf_{ab} + b_+.\overline{a_-}.Buf_{ab}$$
$$Buf_{bc} = Buf_{ab}[f] \qquad where\ [f] \triangleq [c_+/b_+, c_-/b_-, b_-/a_+, b_+/a_-]$$
$$Sys = (Buf_{ab} | Buf_{bc}) \setminus L \qquad where\ L \triangleq \{b_+, b_-\}$$



deadlock