Software Model Checking: Theory and Practice

Lecture: *Specification Checking - Temporal Logic*

Objectives

- Understand why temporal logic can be a useful formalism for specifying properties of concurrent/reactive systems.
- Understand the intuition behind Computation Tree Logic (CTL) – the specification logic used e.g., in the well-known SMV model-checker.
- Be able to confidently apply Linear Temporal Logic (LTL) – the specification logic used in e.g., Bogor and SPIN – to specify simple properties of systems.
- Understand the formal semantics of LTL.

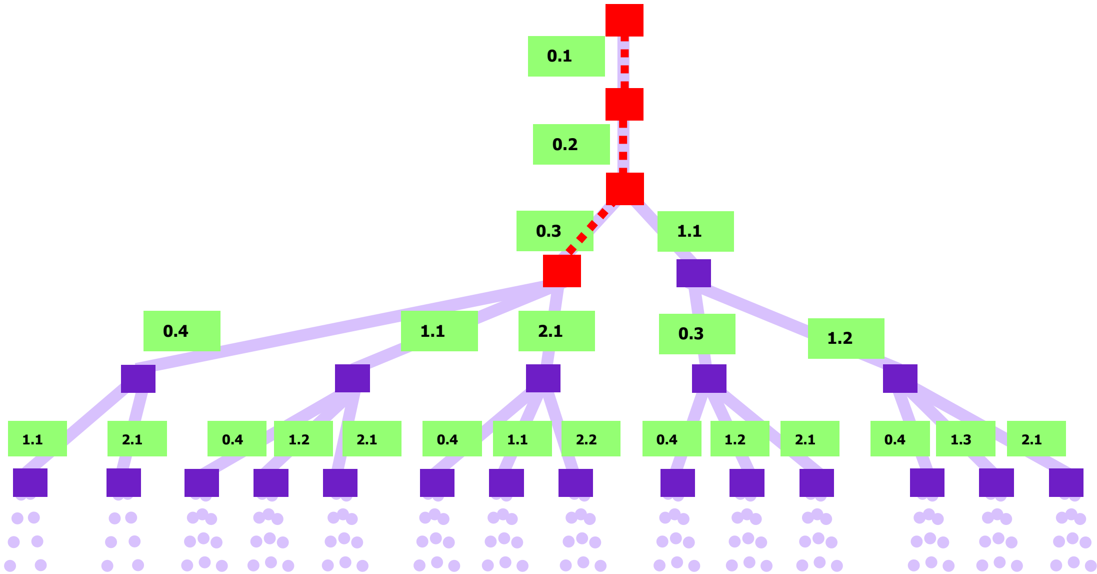
Outline

- CTL by example
- LTL by example
- LTL – formal definition
- Common properties to be stated for concurrent systems and how they can be specified using LTL
- Bogor's support for LTL

Reasoning about Executions

- We’ve seen specifications that are about individual program states
  - e.g., assertions, invariants
- Sometimes we want to reason about the relationship between multiple states
  - Must one state always precede another?
  - Does seeing one state preclude the possibility of subsequently seeing another?
- We need to shift our thinking from states to paths in the state space

Reasoning about Executions



- We want to reason about execution trees
  - tree node = snap shot of the program's state
- Reasoning consists of two layers
  - defining predicates on the program states (control points, variable values)
  - expressing temporal relationships between those predicates

Examples

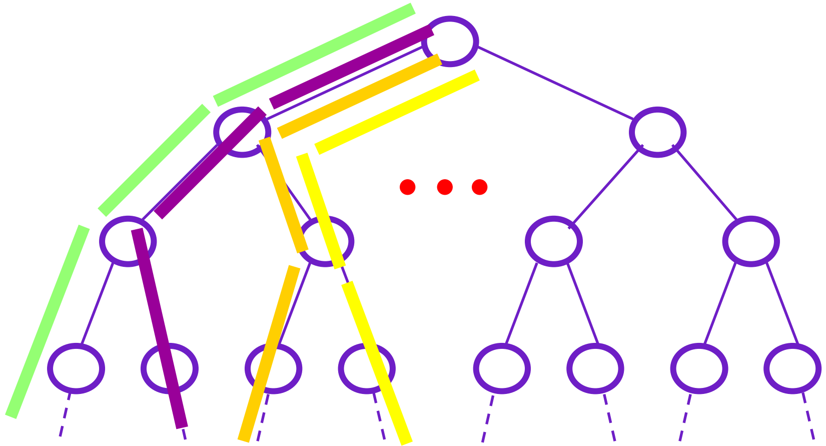
- A use of a variable must be preceded by a definition
- When a file is opened it must subsequently be closed
- You cannot shift from drive to reverse without passing through neutral
- The program will eventually terminate

Why Use Temporal Logic?

- Requirements of concurrent, distributed, and reactive systems are often phrased as constraints on *sequences of events or states* or constraints on *execution paths.*
- Temporal logic provides a formal, expressive, and compact notation for realizing such requirements.
- The temporal logics we consider are also strongly tied to various computational frameworks (e.g., automata theory) which provides a foundation for building verification tools.

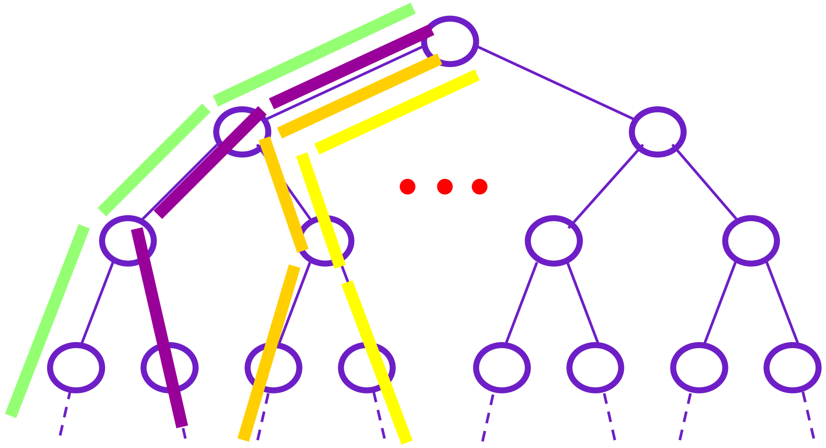
Linear Time Logic

Restrict path quantification to *"ALL" (no "EXISTS")*

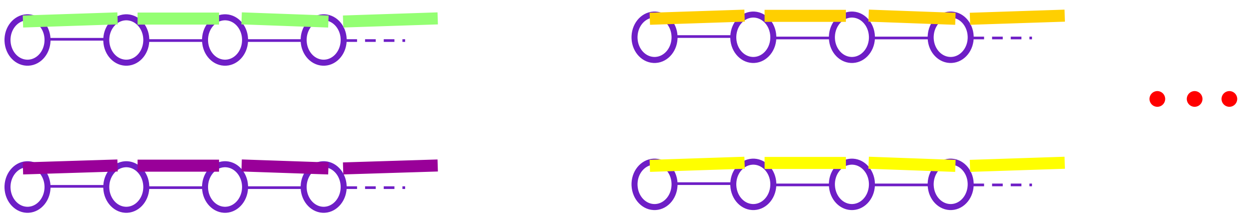


Linear Time Logic

Restrict path quantification to *"ALL" (no "EXISTS")*



Reason in terms of branching traces instead of branching trees



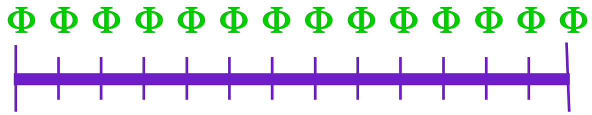
Linear Time Logic (LTL)

Syntax

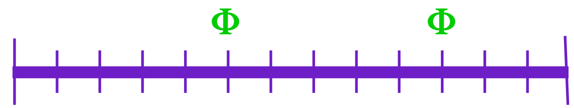
```
Φ ::=  P                                          ...primitive propositions
   | !Φ  | Φ && Φ | Φ || Φ | Φ -> Φ               ...propositional connectives
   | []Φ | <>Φ    | Φ U Φ  | X Φ                  ...temporal operators
```

Semantic Intuition

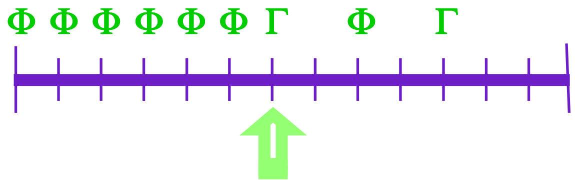
[]Φ ...*always* Φ



<>Φ ...*eventually* Φ



Φ U Γ ...Φ *until* Γ



Modal vs. Temporal Logic

Modal Logic	Temporal Logic (LTL)
(G, R)	(G,<)
Kripke Structures	Temporal Structures
World $g \in G$	Time point $g \in G$
[]F <>F	[]F (always in the future) <>F (sometimes in the future) XF (next time point) F U G (until) …

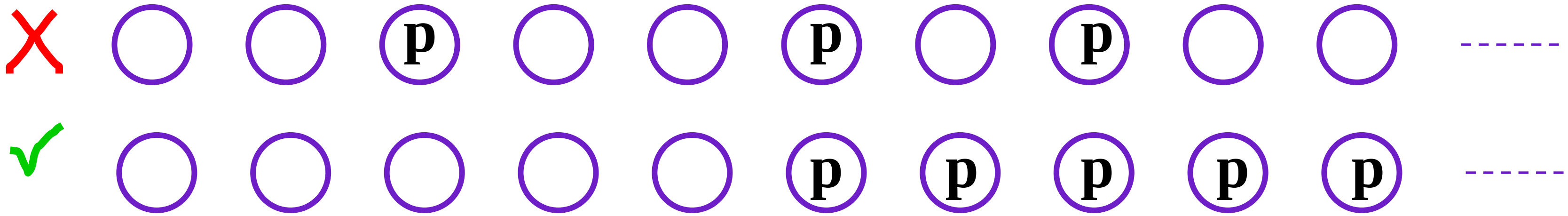
Linear Time Logic

[]<>p

- “Along all paths, it must be the case that globally (I.e., in each state we come to) eventually p will hold”
- Expresses a form of fairness
  - p must occur infinitely often along the path
  - To check Φ under the assumption of fair traces, check []<>p -> Φ

Linear Time Logic

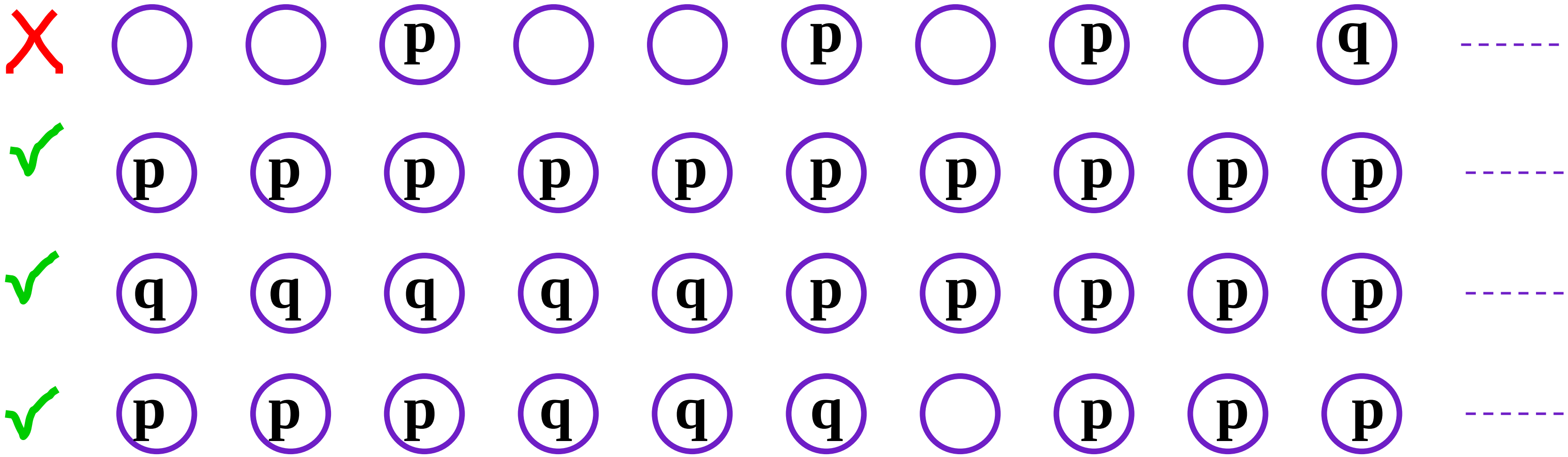
<>[]p



- "Along all paths, eventually it is the case that p holds at each state)" (i.e., "eventually permanently p")
- "Any path contains only finitely many !p states"

Linear Time Logic

p W q = **[]p || (p U q)**



- "p unless q", or "p waiting for q", or "p weak-until q"

Semantics for LTL

- Semantics of LTL is given with respect to a (usually infinite) path or trace
  - $\pi = s_1\ s_2\ s_3\ \ldots$
- We write π_i for the suffix starting at s_i, e.g.,
  - $\pi_3 = s_3\ s_4\ s_5\ \ldots$
- A system satisfies an LTL formula f if each path through the system satisfies f.

Semantics of LTL

- For primitive propositions p:

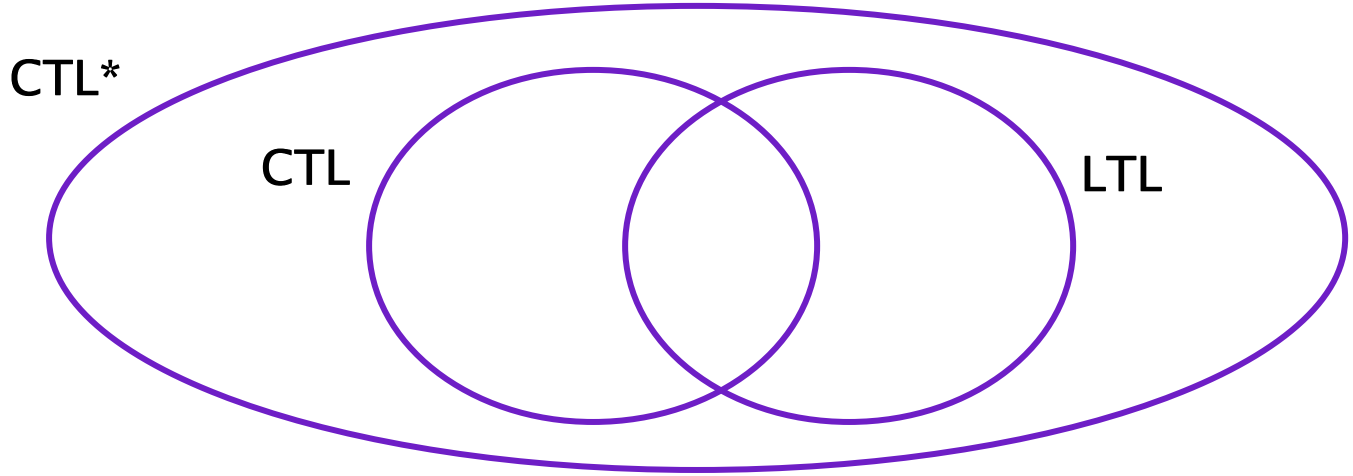
  $\pi \models p \Leftrightarrow s_1 \models p \qquad \pi \models !p \Leftrightarrow s_1 \models !p$
- $\pi \models f \wedge g \Leftrightarrow \pi \models f$ and $\pi \models g$
- $\pi \models f \vee g \Leftrightarrow \pi \models f$ or $\pi \models g$
- $\pi \models Xf \Leftrightarrow \pi_2 \models f$
- $\pi \models \langle\rangle f \Leftrightarrow \exists i \geq 1.\ \pi_i \models f$
- $\pi \models []f \Leftrightarrow \forall i \geq 1.\ \pi_i \models f$
- $\pi \models (f\ U\ g) \Leftrightarrow \exists i >= 1.\ \pi_i \models g$

  and $\forall j : 1 \leq j < i.\ \pi_j \models f$

LTL Notes

- Invented by Prior (1960's), and first used to reason about concurrent systems by A. Pnueli, Z. Manna, etc.
- LTL model-checkers are usually explicit-state checkers due to connection between LTL and automata theory
- Most popular LTL-based checker is SPIN (G. Holzman)

Comparing LTL and CTL



- CTL is not strictly more expression than LTL (and vice versa)
- CTL* invented by Emerson and Halpern in 1986 to unify CTL and LTL
- We believe that almost all properties that one wants to express about software lie in intersection of LTL and CTL

A classic distinction …

- Safety properties
  - "nothing bad ever happens"
  - are violated by a *finite* path prefix that ends in a bad thing
  - are fundamentally about the *history* of a computation up to a point
- Liveness properties
  - "something good eventually happens"
  - are violated by *infinite* path suffixes on which the good thing never happens
  - are fundamentally about the *future* of a computation from a point onward

Examples

- A use of a variable must be preceded by a definition
- When a file is opened it must subsequently be closed
- You cannot shift from drive to reverse without passing through neutral
- No pair of adjacent dining philosophers can be eating at the same time
- The program will eventually terminate
- The program is free of deadlock

Examples

- A use of a variable must be preceded by a definition -- Safety
- When a file is opened it must subsequently be closed -- Liveness
- You cannot shift from drive to reverse without passing through neutral -- Safety
- No pair of adjacent dining philosophers can be eating at the same time -- Safety
- The program will eventually terminate -- Liveness
- The program is free of deadlock -- Safety