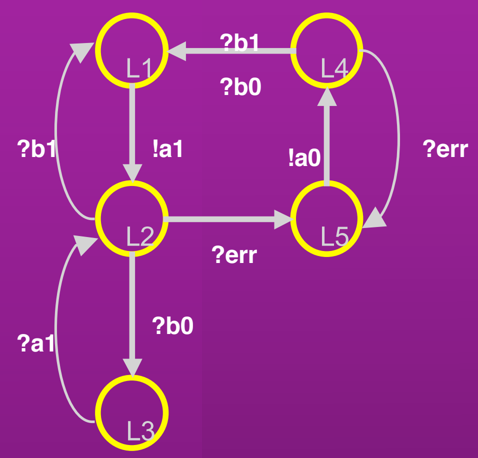
A Crash Course on Temporal Specifications

John Hatcliff *[Kansas State]*

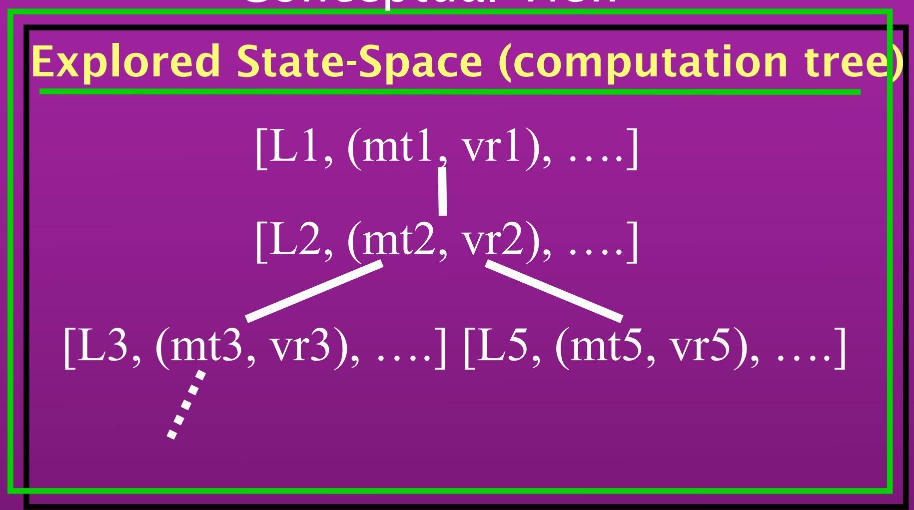
Work on specification patterns by
Matthew Dwyer, Jay Corbett, and George Avrunin

http://www.cis.ksu.edu/santos/bandera

Reasoning about Executions



Conceptual View



- 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

Computational Tree Logic (CTL)

Syntax

Φ ::= P …primitive propositions

| !Φ | Φ && Φ | Φ || Φ | Φ -> Φ …propositional connectives

| AG Φ | EG Φ | AF Φ | EF Φ …temporal operators

| AX Φ | EX Φ | A[Φ U Φ] | E[Φ U Φ]

Semantic Intuition

path quantifier

temporal operator

AG p …along *All* paths p holds *Globally*

EG p …there *Exists* a path where p holds *Globally*

AF p …along *All* paths p holds at some state in the *Future*

EF p …there *Exists* a path where p holds at some state in the *Future*

Computational Tree Logic (CTL)

Syntax

Φ ::= P ...primitive propositions
| !Φ | Φ && Φ | Φ || Φ | Φ -> Φ ...propositional connectives
| AG Φ | EG Φ | AF Φ | EF Φ ...path/temporal operators
| AX Φ | EX Φ | A[Φ U Φ] | E[Φ U Φ]

Semantic Intuition

AX p ...along *All* paths, p holds in the *neXt* state

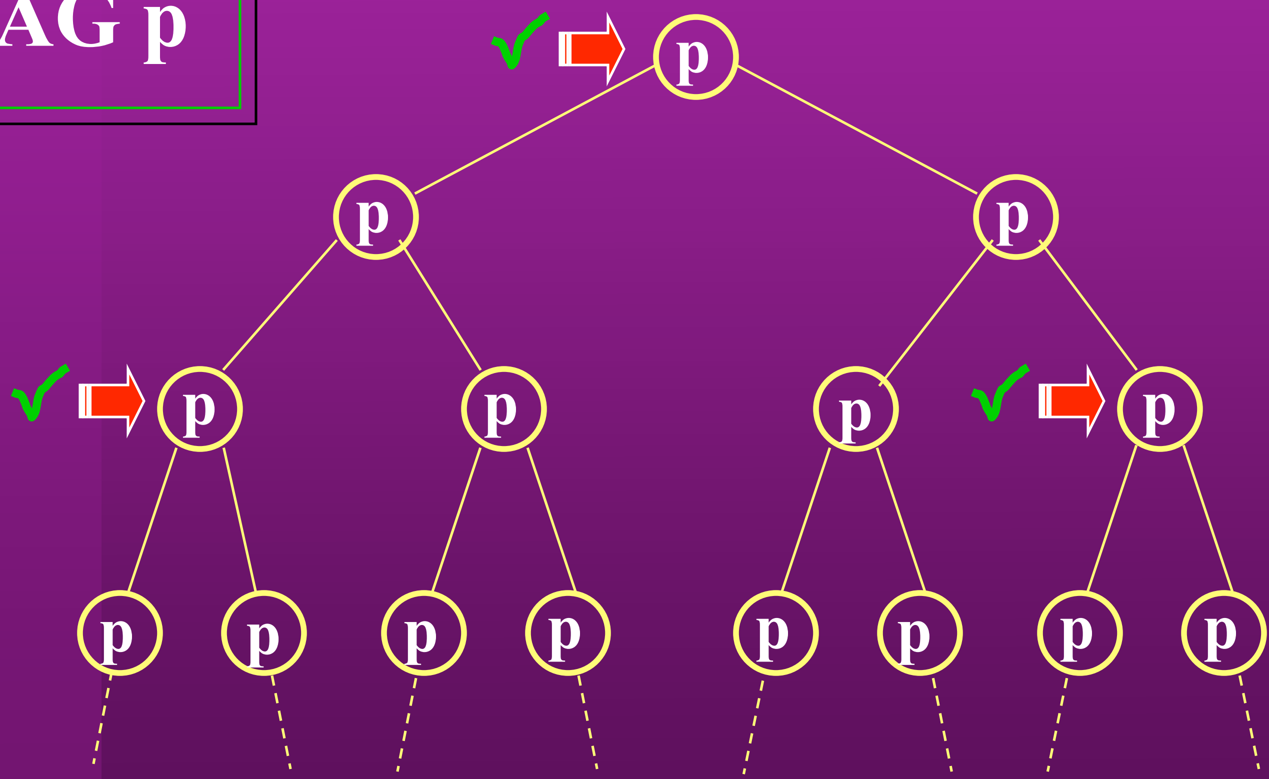
EX p ...there *Exists* a path where p holds in the *neXt* state

A[p U q] ...along *All* paths, p holds *Until* q holds

E[p U q] ...there *Exists* a path where p holds *Until* q holds

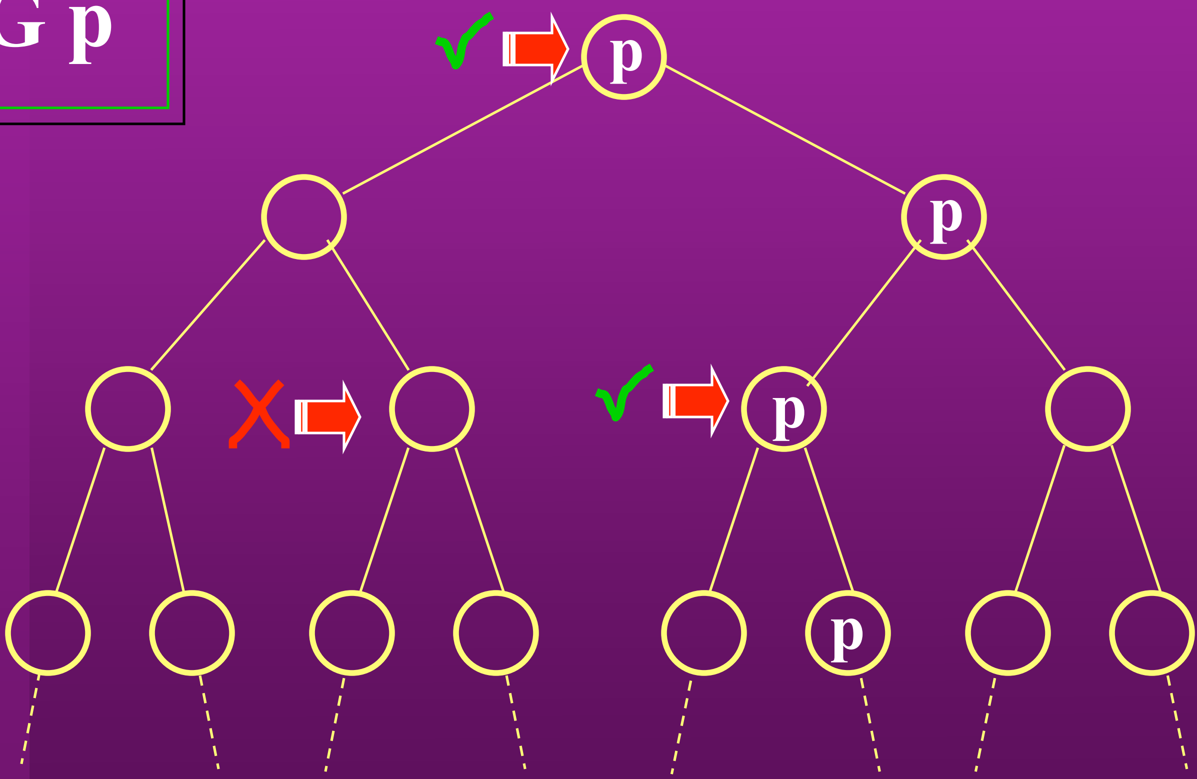
Computation Tree Logic

AG p



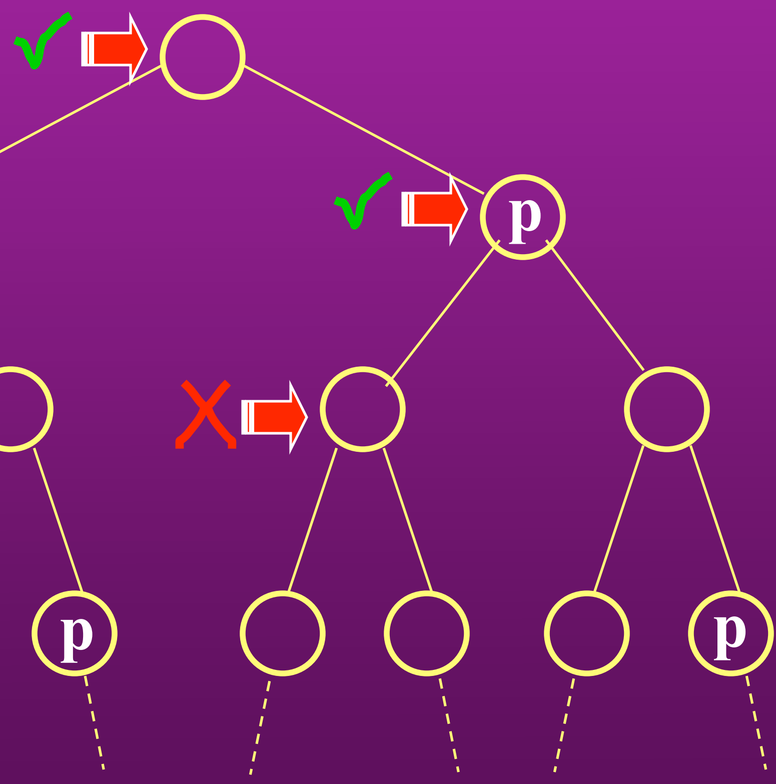
Computation Tree Logic

EG p



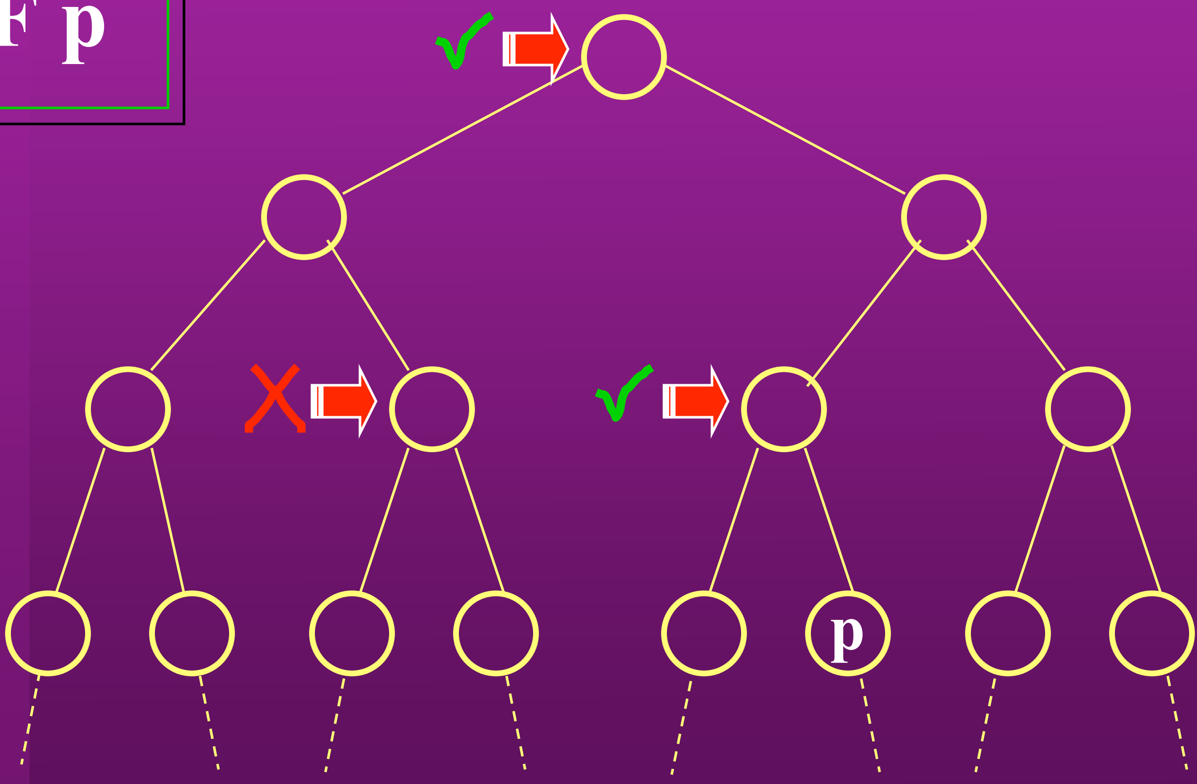
Computation Tree Logic

AF p



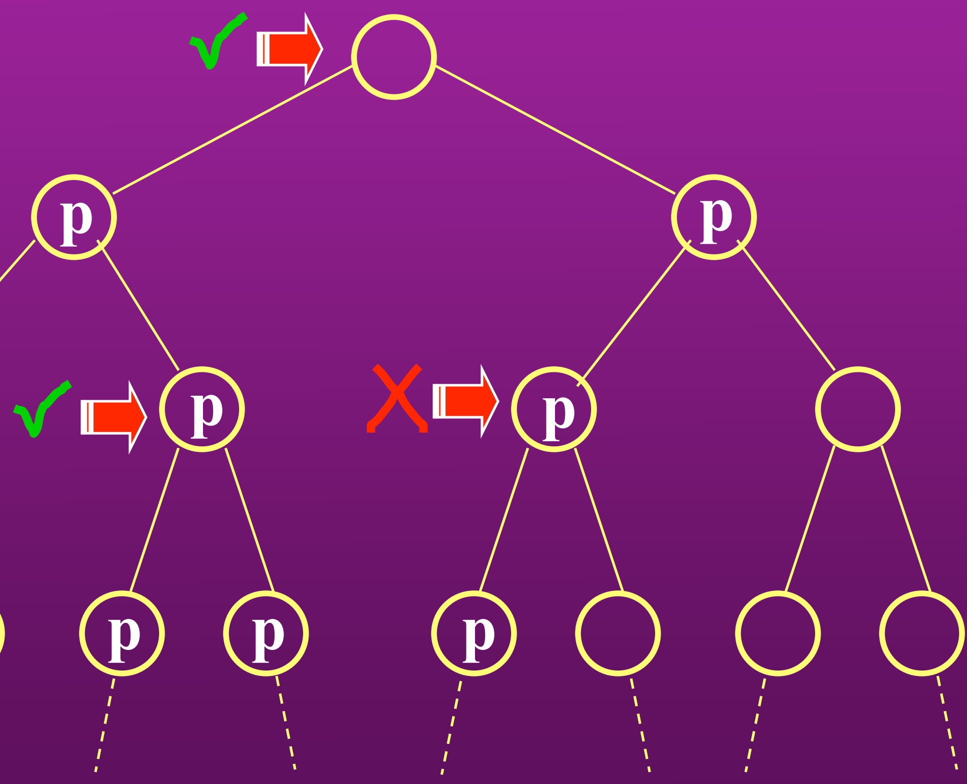
Computation Tree Logic

EF p



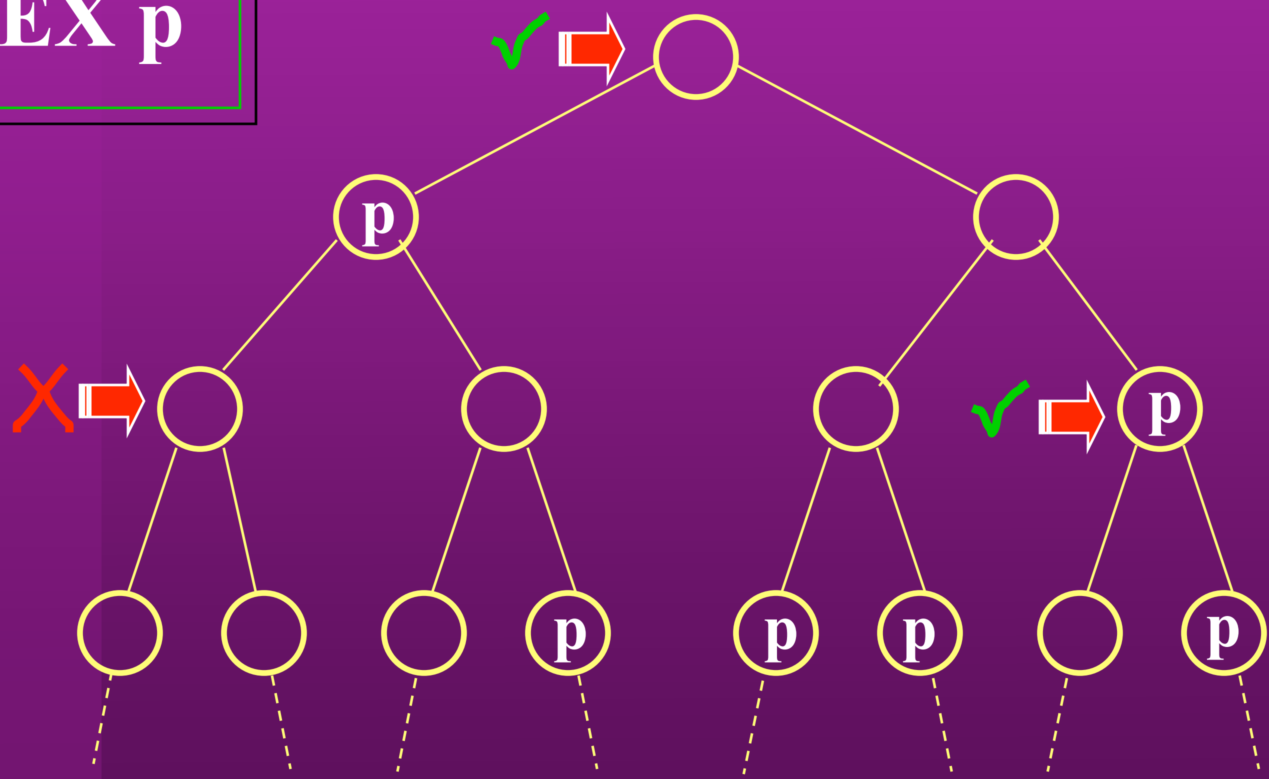
Computation Tree Logic

AX p



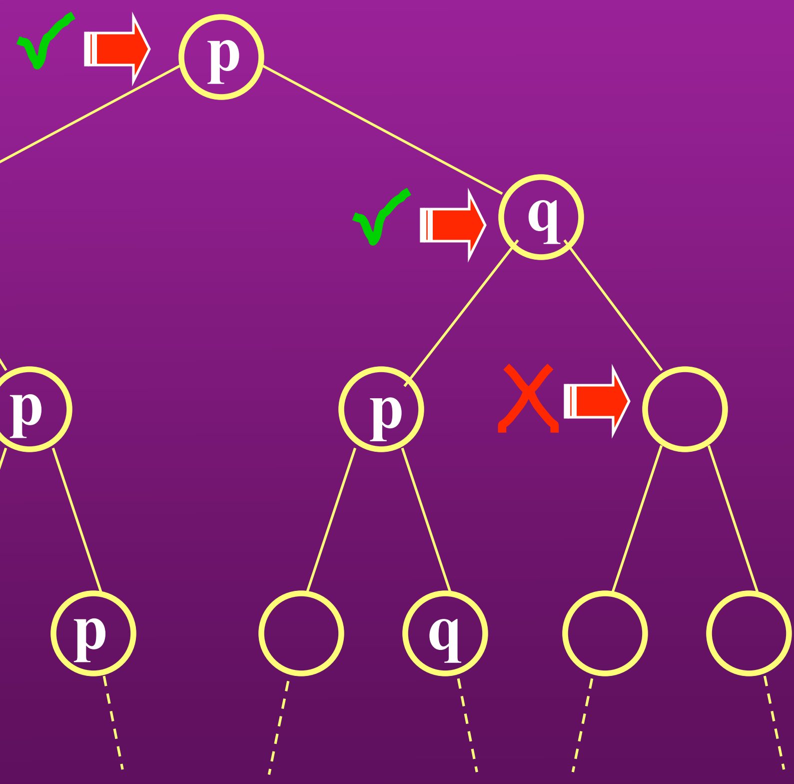
Computation Tree Logic

EX p



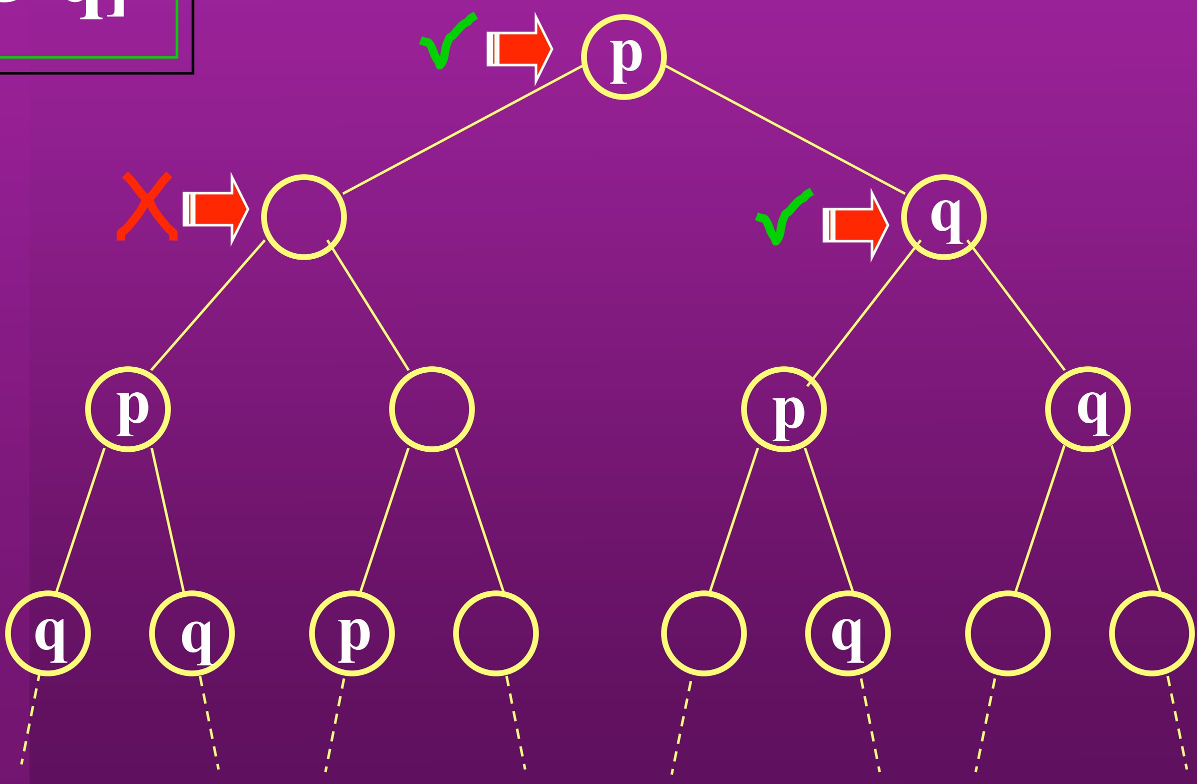
Computation Tree Logic

A[p U q]



Computation Tree Logic

E[p U q]



Example CTL Specifications

- For any state, a request (for some resource) will eventually be acknowledged

AG(requested -> AF acknowledged)

- From any state, it is possible to get to a restart state

AG(EF restart)

- An upwards travelling elevator at the second floor does not changes its direction when it has passengers waiting to go to the fifth floor

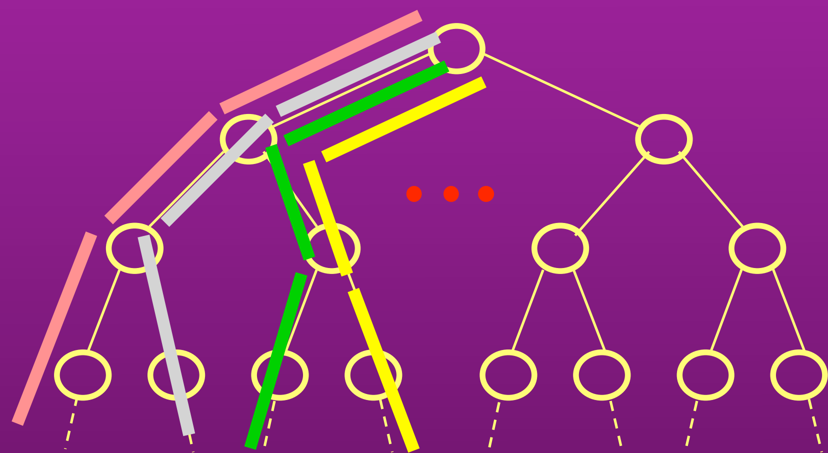
AG((floor=2 && direction=up && button5pressed)
-> A[direction=up U floor=5])

CTL Notes

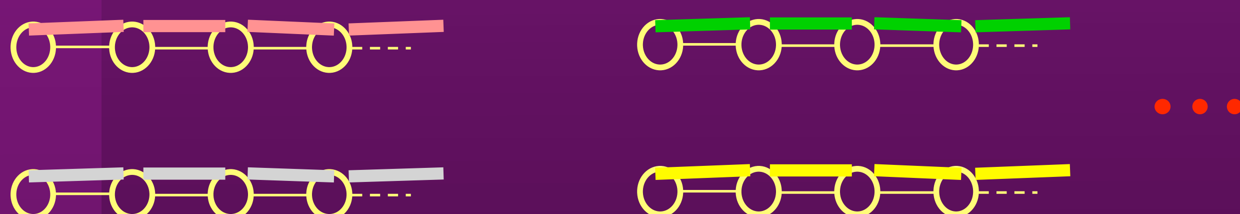
- Invented by E. Clarke and E. A. Emerson (early 1980's)
- Specification language for Symbolic Model Verifier (SMV) model-checker
- SMV is a *symbolic* model-checker instead of an *explicit-state* model-checker
- Symbolic model-checking uses Binary Decision Diagrams (BDDs) to represent boolean functions (both transition system and specification

Linear Temporal Logic

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



Reason in terms of linear *traces* instead of branching *trees*



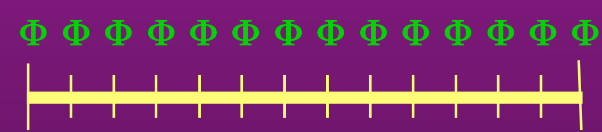
Linear Temporal Logic (LTL)

Syntax

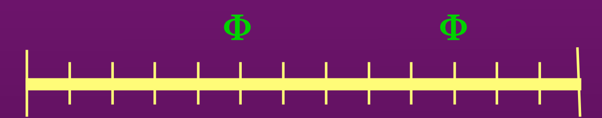
Φ ::= P …primitive propositions
  | !Φ | Φ && Φ | Φ || Φ | Φ -> Φ …propositional connectives
  | []Φ | <>Φ | Φ U Φ | X Φ …temporal operators

Semantic Intuition

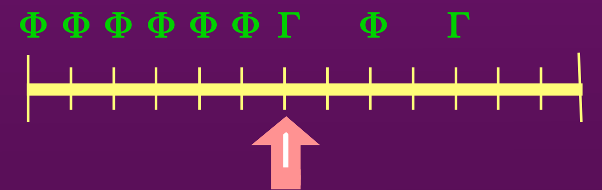
[]Φ …*always* Φ



<>Φ …*eventually* Φ



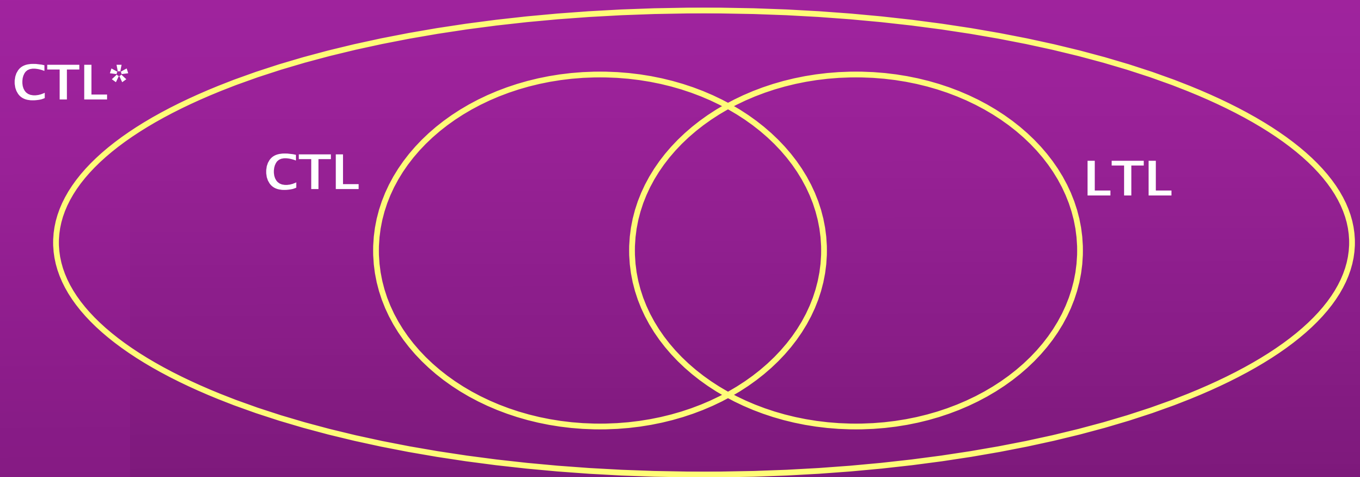
Φ U Γ …Φ *until* Γ



LTL Notes

- Invented by Prior (1960's), and first use 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

Motivation for Specification Patterns

- Temporal properties are not always easy to write
- Clearly many specifications can be captured in both CTL and LTL

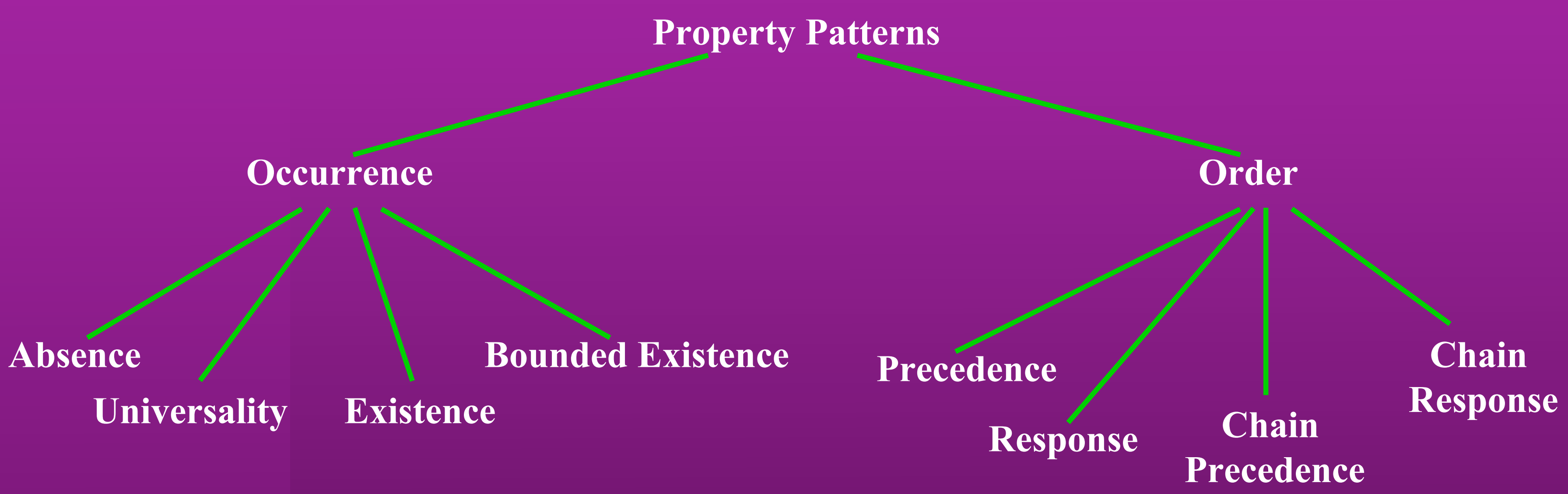
Example: **action Q must respond to action P**

CTL: AG(P -> AF Q) **LTL: [](P -> <>Q)**

We use specification patterns to:

- Capure the experience base of expert designers
- Transfer that experience between practictioners.

Pattern Hierarchy



Classification

- *Occurrence* Patterns:
  - require states/events to occur or not to occur
- *Order* Patterns
  - constrain the order of states/events

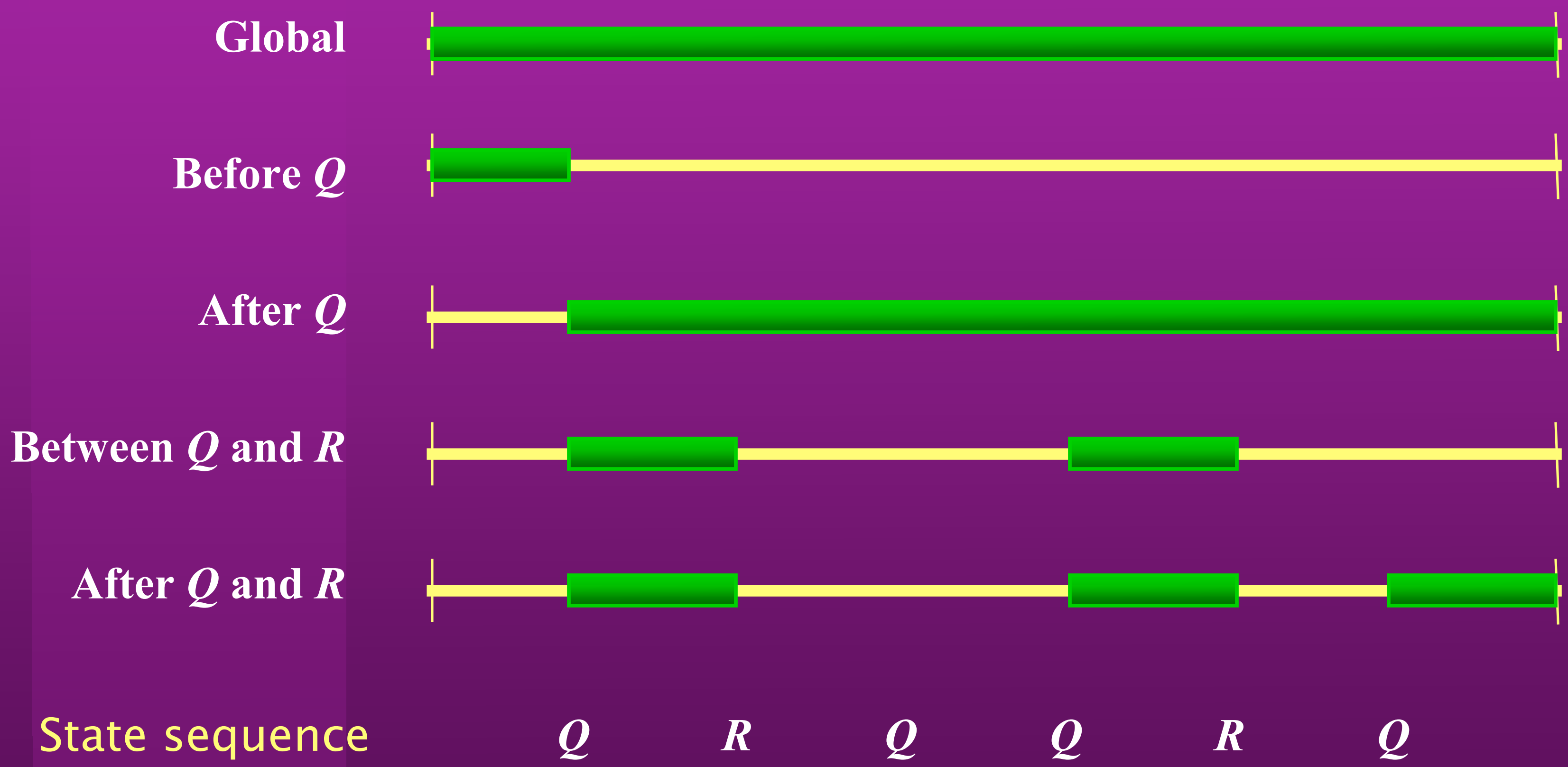
Occurrence Patterns

- Absence: A given state/event does not occur within a scope
- Existence: A given state/event must occur within a scope
- Bounded Existence: A given state/event must occur *k* times within a scope
  - variants: *at least k* times in scope, *at most k* times in scope
- Universality: A given state/event must occur throughout a scope

Order Patterns

- Precedence: A state/event P must always be preceded by a state/event Q within a scope
- Response: A state/event P must always be followed a state/event Q within a scope
- Chain Precedence: A sequence of state/events P1, …, Pn must always be preceded by a sequence of states/events Q1, …, Qm within a scope
- Chain Response: A sequence of state/events P1, …, Pn must always be followed by a sequence of states/events Q1, …, Qm within a scope

Pattern Scopes



The Response Pattern

Intent

To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect. Also known as **Follows** and **Leads-to**.

Mappings: *In these mappings, **P** is the cause and **S** is the effect*

LTL:

Globally:	[](P -> <>S)
Before R:	<>R -> (P -> (!R U (S & !R))) U R
After Q:	[](Q -> [](P -> <>S))
Between Q and R:	[]((Q & !R & <>R) -> (P -> (!R U (S & !R))) U R)
After Q until R:	[](Q & !R -> ((P -> (!R U (S & !R))) W R)

The Response Pattern (continued)

Mappings: *In these mappings, **P** is the cause and **S** is the effect*

CTL:

Globally:	AG(P -> AF(S))
Before R:	A[((P -> A[!R U (S & !R)]) I AG(!R)) W R]
After Q:	A[!Q W (Q & AG(P -> AF(S))]
Between Q and R:	AG(Q & !R -> A[((P -> A[!R U (S & !R)]) I AG(!R)) W R])
After Q until R:	AG(Q & !R -> A[(P -> A[!R U (S & !R)]) W R])

Examples and Known Uses:

Response properties occur quite commonly in specifications of concurrent systems. Perhaps the most common example is in describing a requirement that a resource must be granted after it is requested.

Relationships

Note that a Response property is like a converse of a Precedence property. Precedence says that some cause precedes each effect, and...

Specify Patterns in Bandera

The Bandera Pattern Library is populated by writing pattern macros:

```
pattern {
   name = “Response”
   scope = “Globally”
   parameters = {P, S}
   format = “{P} leads to {S} globally”
   ltl = “[]({P} –> <>{S})”
   ctl = “AG({P} –> AF({S}))”
}
```

Evaluation

- 555 TL specs collected from at least 35 different sources
- 511 (92%) matched one of the patterns
- Of the matches...
  - Response: 245 (48%)
  - Universality: 119 (23%)
  - Absence: 85 (17%)

Questions

- Do patterns facilitate the learning of specification formalisms like CTL and LTL?
- Do patterns allow specifications to be written more quickly?
- Are the specifications generated from patterns more likely to be correct?
- Does the use of the pattern system lead people to write more expressive specifications?

Based on anecdotal evidence, we believe the answer to each of these questions is “yes”

For more information...

- Pattern web pages and papers

  http://www.cis.ksu.edu/santos/spec-patterns