Software Model Checking: Theory and Practice

Lecture: *Specification Checking - Specification Patterns*

Objectives

- Understand the purpose of the temporal specification pattern system
  - encode design knowledge of expert specifiers and make this knowledge accessible through patterns to novice users
  - purpose is *not* to avoid learning the semantics of temporal logics
- Understand the basic temporal classifications and temporal scopes in the pattern system
- Be able to apply the pattern system to realize relatively complex specifications

Outline

- Safety/Liveness classification
- Manna/Pnueli classification
- Temporal Specification Patterns
- Assessment of the Pattern System
- Pointers to other User Friendly Temporal Specification Notations

Motivation

Temporal properties are not always easy to write or read

```
[]((Q & !R & <>R) -> (P -> (!R U (S & !R))) U R)
```

Hint: This a common structure that one would want to use in real systems

Answer:
P triggers S between Q (e.g., end of system initialization) and R (start of system shutdown)

Motivation

Many specifications that people want to write can be specified, e.g., in both CTL and LTL

Example: action Q must respond to action P

CTL: AG(P -> AF Q)

LTL: [](P -> <>Q)

Example: action S precedes P after Q

CTL: A[!Q W (Q & A[!P W S])]

LTL: []!Q | <>(Q & (!P W S))

Motivation

We use Specification Patterns to...

- Capture the experience base of expert designers
- Transfer that experience between practitioners
- Classify properties
  - leverage in implementations
    - e.g., specialize to a particular pattern of properties
  - allow informative communication about properties
    - e.g, “This is a *response property* with an *after* scope.”

Other Classifications

- Safety vs Liveness
  - Independent of a particular formalism
- Practically, it is important to know the difference because...
  - It impacts how we design verification algorithms and tools
    - Some tools only check safety properties (e.g., based on *reachability* algorithms)
  - It impacts how we run tools
    - Different command line options are used for Spin
  - It impacts how we form abstractions
    - Liveness properties often require forms of abstraction that differ from those used in safety properties

Safety Properties

- Informally, a safety property states that
  nothing bad ever happens
- Examples
  - Invariants: “x is always less than 10”
  - Deadlock freedom: “the system never reaches a state where no moves are possible”
  - Mutual exclusion: “the system never reaches a state where two processes are in the critical section”
- As soon as you see the “bad thing”, you know the property is false
- Safety properties can be falsified by a finite-prefix of an execution trace
  - Practically speaking, a Spin error trace for a safety property is a finite list of states beginning with the initial state

Liveness Properties

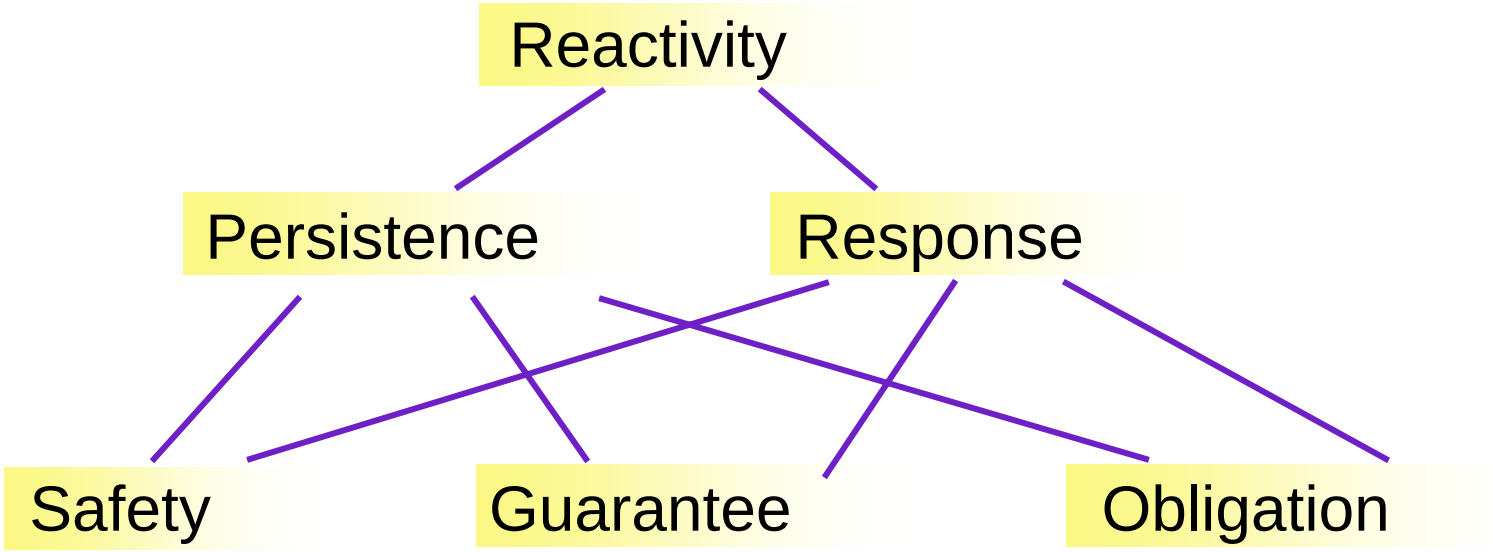
- Informally, a liveness property states that
  something good will eventually happen
- Examples
  - Termination: “the system eventually terminates”
  - Response properties: “if action X occurs then eventually action Y will occur”
- Need to keep looking for the “good thing” forever
- Liveness properties can be falsified by an infinite-suffix of an execution trace
  - Practically speaking, a Spin error trace for a liveness property is a finite list of states beginning with the initial state followed by a *cycle* showing you a loop that can cause you to get stuck and never reach the “good thing”

Assessment

- Safety vs Liveness is an important distinction
- However, it is very coarse
  - Lots of variations within safety and liveness
  - A finer classification might be more useful

Manna & Pnueli Classification

Classification based on syntactic structure of formula



Manna & Pnueli Classification

Canonical Forms

- Safety: [] p
- Guarantee: <> p
- Obligation: [] q || <> p
- Response: [] <> p
- Persistence: <> [] p
- Reactivity: []<>p || <>[]q

Assessment

- The Manna-Pnueli classification is reasonable
- However, their classification is based on the structure of formula, and we would like to avoid having engineers begin their reasoning by reasoning about the structure of formula
- A classification based on the *semantics* of properties instead of syntax might be more useful for non-experts

Specification Pattern System

- http://patterns.projects.cis.ksu.edu/
- Developed by Dwyer, Avrunin, Corbett.
- A pattern system for presenting, codifying, and reusing property specifications for finite-state verification (e.g., model-checking).
- Developed by examining over 500 temporal specifications collected from the literature.
- Organized into a hierarchy based on the semantics of the requirement.

The Specification Pattern System

- A property specification pattern...
  - ...is a generalized description of a commonly occurring requirement on the permissible state/event sequences in a finite-state model of a system.
  - ...describes the essential structure of some aspect of a system's behavior and provides expressions of this behavior in a range of formalisms.

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)]) | 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)]) | AG(!R)) W R])`

After Q until R: `AG(Q & !R -> A[(P -> A[!R U (S & !R)]) W R])`

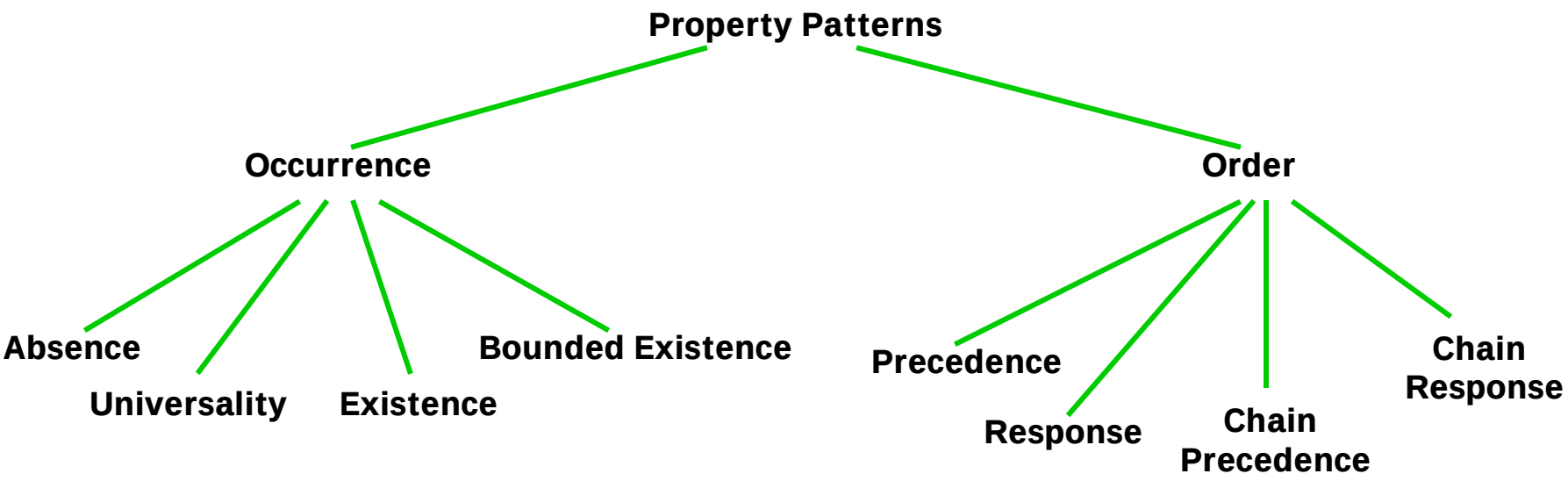
Examples and Known Uses:

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...

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 state/event does not occur within a given scope

Existence:

A given state/event must occur within a given scope

Bounded Existence:

A given state/event must occur *k* times within a given scope

- variants: a least *k* times, at most *k* times

Universality

A given state/event must occur throughout a given 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

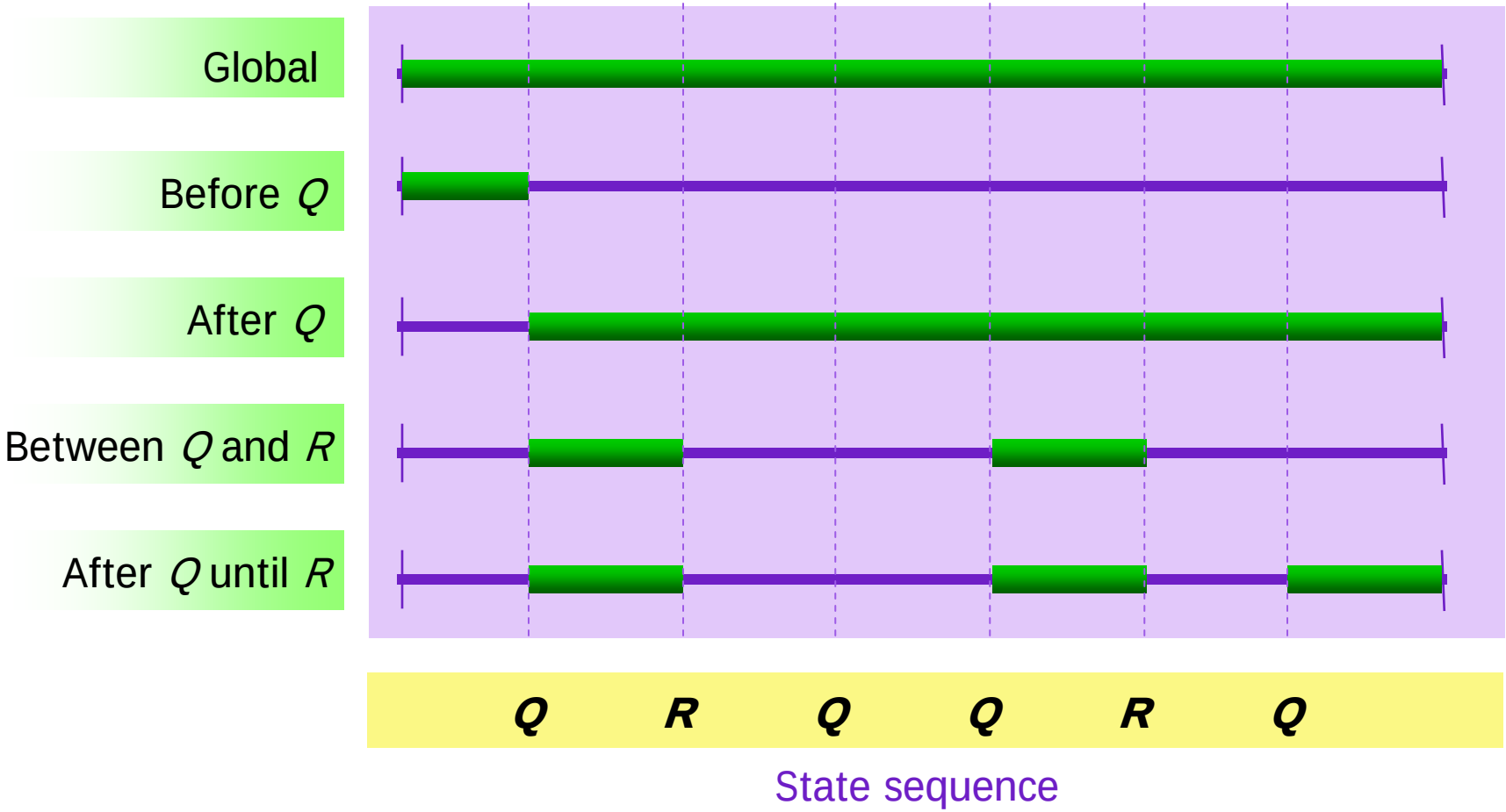
Chain Precedence

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



For You To Do…

- Pause the lecture...
- Using the pattern system, identify the (a) propositions, (b) base temporal pattern and (c) scope, and use the pattern web-pages to construct the corresponding LTL property for each of the seven requirements listed on the following slides. Express the temporal property by filling in the holes of the pattern with the identified propositions. Below is a completed example...

Requirement:

When a client A makes a method call to a server B, it will eventually receive the result of its call if the server is OK.

Answer:

Propositions: ***clientASendB, clientAreceiveB***

Pattern & Scope: **"*response" pattern with "global" scope***

Property: ***{clientAreceiveB} responds to {clientASendB} globally***

LTL: ***[](clientASendB -> <>clientAreceiveB)***

For You To Do…

Requirement 1:

Between an enqueue(d1) and empty(true) there must be a dequeue(d1)

Requirement 2:

It is always the case that when the req_search_state is not enabled, then the req_close_state shall not be closed and will remain not closed until the req_search_state is enabled.

For You To Do…

Requirement 3:

After OpeningNetworkConnection, an ErrorMessage will pop up in response to a NetworkError

Requirement 4:

Every time the form is patron_view it must have been preceeded by a corresponding request_view.

For You To Do…

Requirement 5:

Checkout is 0 until the Status of the book is charged or hold.

Requirement 6:

Only one of the 3-counting semaphore's four semaphore place's may be occupied at any one time.

...end of “For You To Do” requirements.

For You To Do (Answers)

Requirement 1:

Between an enqueue(d1) and empty(true) there must be a dequeue(d1)

Answer:

Propositions: ***enqueue(d1), emtpy(true), dequeue(d1)***

Pattern & Scope: **"*existence" pattern with "between" scope***

Property: ***{dequeue(d1)} exists between {enqueue(d1)} and empty(true)***

LTL: ***[](enqueue(d1) & !empty(true)***
-> (!empty(true) W (dequeue(d1) & !empty(true))))

For You To Do (Answers)

Requirement 2:

It is always the case that when the req_search_state is not enabled, then the req_close_state shall not be closed and will remain not closed until the req_search_state is enabled.

Answer:

Propositions: ***req_search_state_enabled, req_close_state_closed***

Pattern & Scope: ***"absence" pattern with "after-until" scope***

Property: ***{req_close_state_closed} is absent after {!req_search_state_enabled} until {req_search_state_enabled}***

LTL: ***[](!req_search_state_enabled***
-> (!req_close_state_closed W req_search_state_enabled))

For You To Do (Answers)

Requirement 3:

After OpeningNetworkConnection, an ErrorMessage will pop up in response to a NetworkError

Answer:

Propositions: ***OpeningNetworkConnection, ErrorMessage, NetworkError***

Pattern & Scope: ***"response" pattern with "after" scope***

Property: ***{ErrorMessage} responds to {NetworkError} after {OpeningNetworkConnection}***

LTL: ***[](OpeningNetworkConnection -> [](NetworkError -> <>ErrorMessage))***

For You To Do (Answers)

Requirement 4:

Every time the form is patron_view it must have been preceded by a corresponding request_view.

Answer:

Propositions: ***form==patron_view, form==request_view***

Pattern & Scope: **"*precedence" pattern with "global" scope***

Property: ***{form==request_view} precedes {form==patron_view} globally***

LTL: ***!form==patron_view W form==request_view***

For You To Do (Answers)

Requirement 5:

Checkout is 0 until the Status of the book is 'charged' or 'hold' (the last two actions need not occur).

Answer:

Propositions: ***status==charged, status==hold, checkOut==0***

Pattern & Scope: **"*precedence" pattern with "global" scope***

Property: ***{status==charged | status==hold} precedes {!checkOut==0} globally***

LTL: **(*checkOut==0) W (status==charged | status==hold)***

For You To Do (Answers)

Requirement 6:

Only one of the 3-counting semaphore's four semaphore place's may be occupied at any one time.

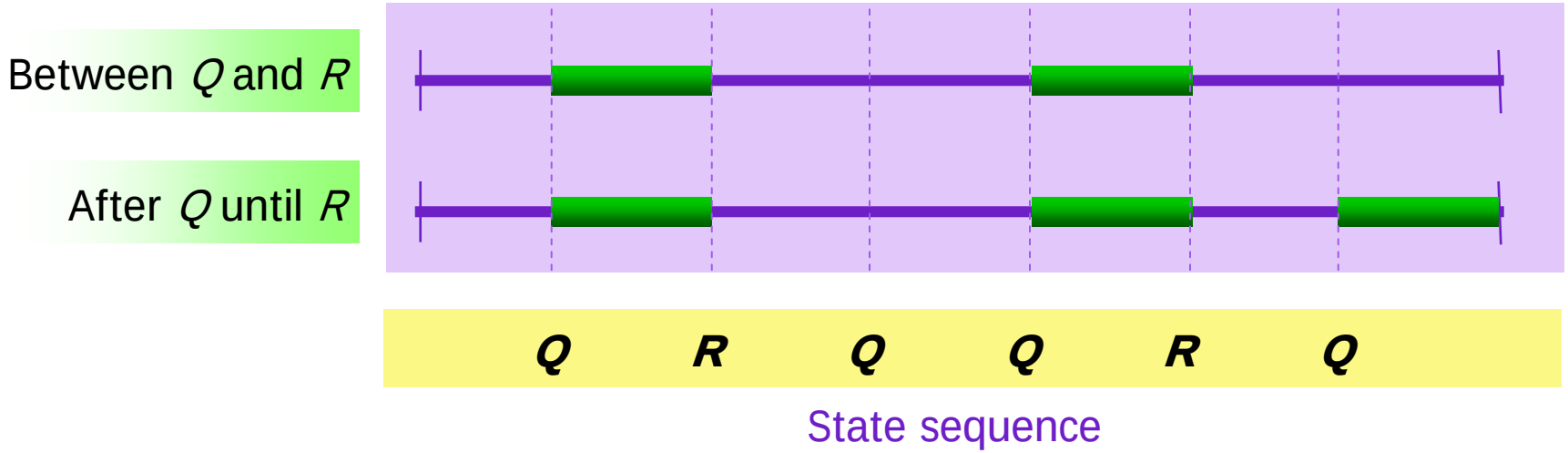
Answer:

Propositions: ***place0, place1, place2, place3***

Pattern & Scope: **"*universal" pattern with "global" scope***

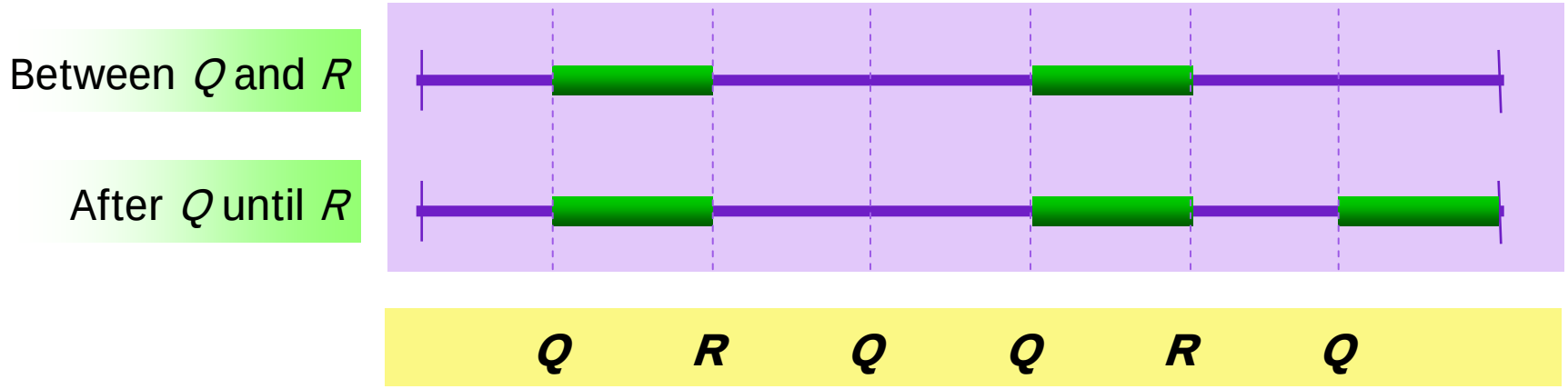
Property: ***{(place0 && !place1 && !place2 && !place3) ||***
(!place0 && place1 && !place2 && !place3) ||
(!place0 && !place1 && place2 && !place3) ||
(!place0 && !place1 && place2 && !place3)} is universal globally

Between vs After-Until



- Note that the *Between* scope only requires the pattern to hold if a matching R exists for the Q.
- In contrast, *After-Until* requires the pattern to hold after every Q until an R is seen (and the matching R need not occur).

Between vs After-Until



LTL mappings for the Existence Pattern

- {P} exists between {Q} and {R}

  [](Q & !R -> (!R W (P & !R)))

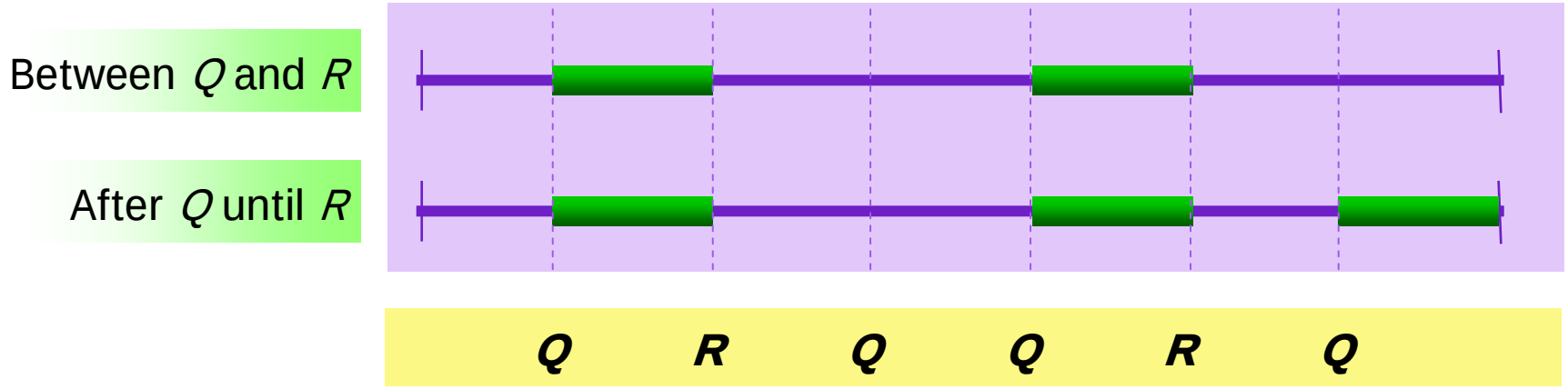
- {P} exists after {Q} until {R}

  [](Q & !R -> (!R U (P & !R)))

Simply says R cannot occur until a P occurs (without R)

Requires P to always occur after Q

Open vs Closed Intervals



Consider: *{P} is universal between {Q} and {R}*

- Does this scope declaration allow P to occur at the state where Q first becomes true (e.g., is the Q/R interval closed on the left)?
- Does this scope declaration allow P to occur at the state where R becomes true (e.g., is the Q/R interval closed on the right)?

Open vs Closed Intervals

Consider: *{P} is universal between {Q} and {R}*

- Is the interval open/closed on the left/right?

LTL Mapping:

[]((Q & !R & <>R) -> (P U R))

- This requires P to occur in the same state where Q becomes true (interval is closed on left).
- This does not require P to occur in the state where R becomes true due to the semantics of the “Until” operator (interval is open on right).
  - note: P is *allowed* to occur when R becomes true

Assessment

- Although the patterns are a useful guide to constructing temporal logic, they are not an excuse for not learning LTL, CTL, etc.
- The English description of particular patterns/scope is ambiguous, and you need to be able to look at the LTL, CTL, etc. to determine the exact meaning in some circumstances.

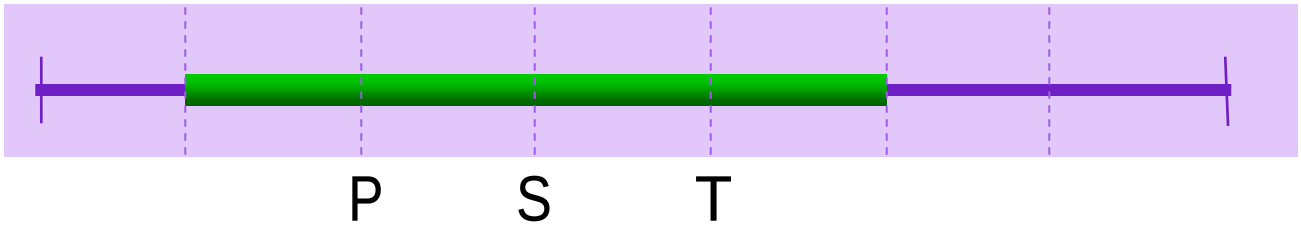
For You To Do…

- Pause the lecture...
- Answer the following questions:
  - Consider the specification {P} is universal after {Q}. By examining the LTL mapping, determine if P should hold at the first state where Q holds if this specification is satisfied.
  - Consider the specification {P} is universal before {Q}. By examining the LTL mapping, determine if P should hold at the first state where Q holds if this specification is satisfied.
  - Consider the specification {P} is universal between {S} and {T}.
    - According to the LTL mapping, when the specification is satisfied, must P occur when T first becomes true?
    - According to the LTL mapping, when the specification is satisfied, can S occur without T?
    - According to the LTL mapping, when the specification is satisfied, if there is no matching T for an S, is P required to hold after the S with no matching T?
  - Repeat the questions immediately above for the specification {P} is universal after {S} until {T}.

Response Chain

Intent: 1-stimulus, 2-response chain

To describe a relationship between a stimulus event (P) and a sequence of two response events (S,T) in which the occurrence of the stimulus event must be followed by an occurrence of the sequence of response events within the scope. In state-based formalisms, the states satisfying the response must be distinct (i.e., S and T must be true in different states to count as a response), but the response may be satisfied by the same state as the stimulus (i.e., P and S may be true in the same state).

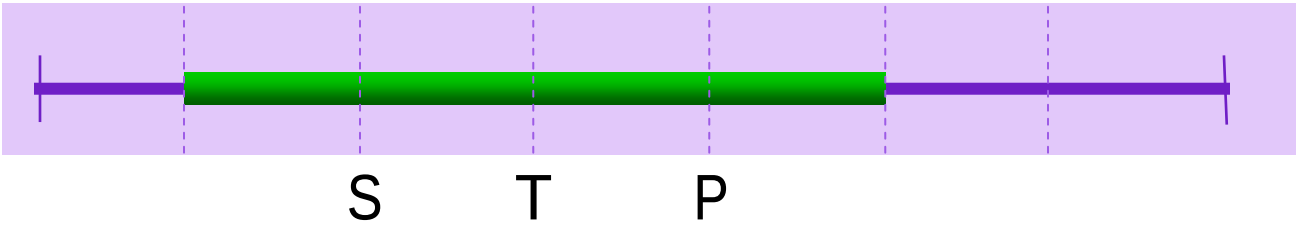


...P triggers S followed by T in the given scope

Response Chain

Intent: 2-stimulus, 1-response chain

To describe a relationship between two stimulus events (S,T) and a response event (P) in which the occurrence of a sequence of the two stimulus events must be followed by an occurrence of the response event. In state-based formalisms, the states satisfying the stimulus must be distinct (i.e., S and T must be true in different states to count as a stimulus), but the response may be satisfied by the same state as the stimulus (i.e., T and P may be true in the same state).



...S followed by T triggers P in the given scope

Response Chain

LTL Mapping: 1-stimulus, 2-response chain

Globally: `[] (P -> <>(S & o<>T))`

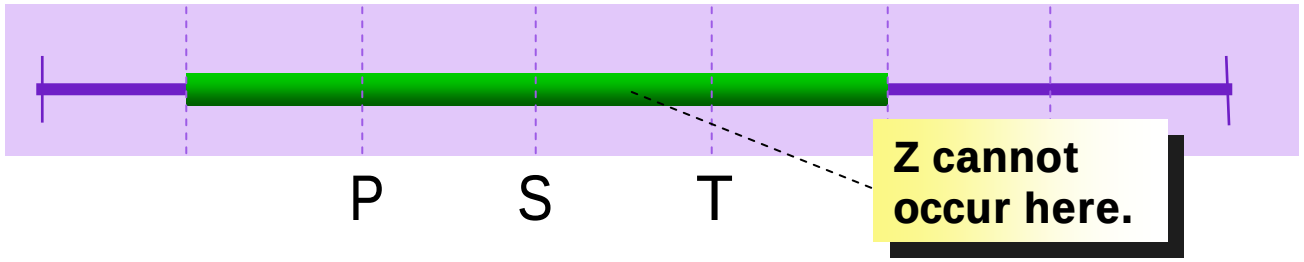
LTL Mapping: 2-stimulus, 1-response chain

Globally: `[] (S & o<> T -> o(<>(T & <> P)))`

Constrained Response Chain

Intent: 1-stimulus, 2-response chain with absence constraint

To describe a relationship between a stimulus event (P) and a sequence of two response events (S,T) in which the occurrence of the stimulus event must be followed by an occurrence of the sequence of response events within the scope. Moreover, Z must not occur between S (inclusive) and T.



...P triggers S followed by T without Z in the given scope

Globally: `[] (P -> <>(S & !Z & o(!Z U T)))`

Constrained Response Example

Requirement:

If between an enqueue of d1 and the initiation of a forward iteration, d1 is not dequeued, then it will eventually be produced by the iteration.

Answer:

Propositions: ***d1-enqueue, init-forward-iteration, d1-dequeued, d1-produced***

Pattern & Scope: constrained 2-stimulus 1-response chain

Property: ***{d1-produced} responds to {enqueue-d1, init-forward-iteration} without {!d1-dequeued}***

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”