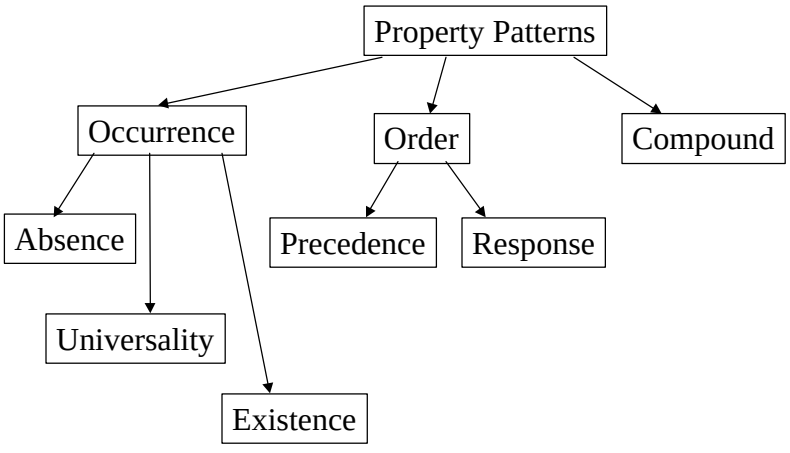
Specification Patterns

- Early taxonomy for property specifications
  - safety properties: nothing bad will ever happen
  - liveness properties: something good will eventually happen

Property Patterns



Relationships

- Note that a **Precedence** property is like a converse of a **Response** property. **Precedence** says that some cause precedes each effect, and **Response** says that some effect follows each cause. They are not equivalent, because **Response** allows effects to occur without causes (**Precedence** similarly allows causes to occur without subsequent effects).

Occurrence Patterns

- **Absence**: A given state/event does not occur within a scope. Also known as **Never**.
- **Existence**: A given state/event must occur within a scope. This pattern is also known as **Future** and **Eventuality**. A variant: **Bounded Existence**: exactly k times, at least k times, at most k times.

Occurrence Patterns

- **Universality**: A given state/event occurs throughout a scope. Also known as **Globally, Always, Henceforth**.

Ordering Patterns

- **Precedence**: A given state/event must always be preceded by a state/event Q within a scope.
- **Response**: A state/event P must always be followed by a state/event Q within a scope. Also known as **Follows** and **Leads-to**. A mixture of **Existence** and **Precedence**.

Some background

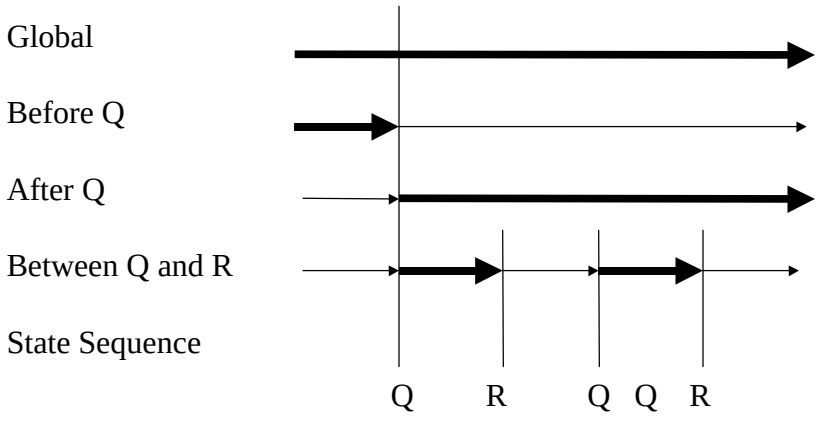
- A *scope* is the extent of a program's execution over which a formula must hold. There are five basic kinds of scopes: global, before, after, between, after-until.

Some background

- *scope*
  - global (the entire program execution),
  - before (the execution up to a given state),
  - after (the execution after a given state)
  - between (any part of the execution from one given state to another given state)
  - after-until (like between even if the second state does not occur)

Some background

- A scope itself should be interpreted as optional; if the scope delimiters are not present in an execution then the specification will be true.



Four Formula Scopes

Specification Pattern System

- **Precedence** Property Pattern: S precedes P. P is the consequent and S is the enabling state/event.
  - Globally
    - A[!P U (S | AG(!P))]: for all paths, P does not hold until S holds or P will never hold

Precedence: Traversal application

- For all traversals which start at an X-object, any visit to a P-object is preceded by a visit to an S-object.
- P uses information produced in S.

Specification Pattern System

- **Precedence** Property Pattern: S precedes P. P is the consequent and S is the enabling state/event.
  - Before R
    - A[!P U (S | R | AG(!P) | AG(!R))]: for all paths, P does not hold until S holds or R holds or P will never hold or R will never hold. When P holds S must have been true earlier if R has not happened.

Precedence: Traversal application

- For all traversals which start at an X-object, any visit to a P-object is preceded by a visit to an S-object provided no R-object has been visited.
- P uses information produced in S or R.

Specification Pattern System

- **Precedence** Property Pattern: S precedes P. P is the consequent and S is the enabling state/event.
  - After Q
    - A[!Q U (AG(!Q) | (Q & A[!P U (S | AG(!P))]))] : for all paths, Q does not hold until Q never holds or Q holds and for all paths P does not hold until S holds or P will never hold.

Precedence: Traversal application

- For all traversals which start at an X-object, any visit to a P-object is preceded by a visit to an S-object provided a Q-object has been visited first.
- Q-object initializes information used by S-object and P-object. S-object computes information used by P-object.

CTL formulas for Absence

- P is false
  - **Globally: AG(!P)**

CTL formulas for Absence

- P is false
  - **Before R: A[!P U (R or AG(!R))]**
  - P is false until R holds or until R will never hold

Absence: Traversal application

- For all traversals which start at an X-object, there can be no visit to a P-object while R is false (e.g., before an R-object is visited).
- While R is false, P can not participate in collaboration.

CTL formulas for Absence

- P is false
  - **After Q: AG(Q => AG(!P))**
  - **For all paths the following condition holds at every state: If Q holds at a state then for all paths from that state !P holds globally.**

Absence: Traversal application

- For all traversals which start at an X-object, after visiting a Q-object we will never visit a P-object.

CTL formulas for Absence

- P is false
  - **Between Q and R: A G(Q => A[!P U (R or A G (!R))])**
  - **Globally, if Q holds at a state s then P is false until R holds or R is false globally from s.**

CTL formulas for Response

- S responds to P: (P is the cause, S the effect)
  - **AFTER Q: AG(Q=>AG(P=>AF(S))) : Globally, if Q holds, then if P holds, eventually S will hold.**

CTL formulas for Response

- S responds to P: (P is the cause, S the effect)
  - **GLOBALLY : AG(P=>AF(S)): Globally, if P holds then S will eventually hold.**

CTL formulas for Response

- S responds to P: (P is the cause, S the effect)
  - **BEFORE R: A[(P=>A[!R U ((S and !R) or AG(!R))]) U (R or AG(!R))]**
  - **Amazing how complex it is to express BEFORE.**
  - **Until R holds or R never holds, if P holds then for all paths until (S and !R) holds or R never holds, not R holds.**

1-2 Response Chain Property Pattern

- Intent: 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.

1-2 Response Chain Property Pattern

- S,T responds to P:
  - Globally
    - AG(P -> AF(S & AX(AF(T))))
  - Before R
    - A[(P -> A[!R U (S & !R & A[!R U T])]) U (R | AG(!R))]
  - After Q
    - AG(Q -> AG(P -> AF(S & AX(AF(T)))))