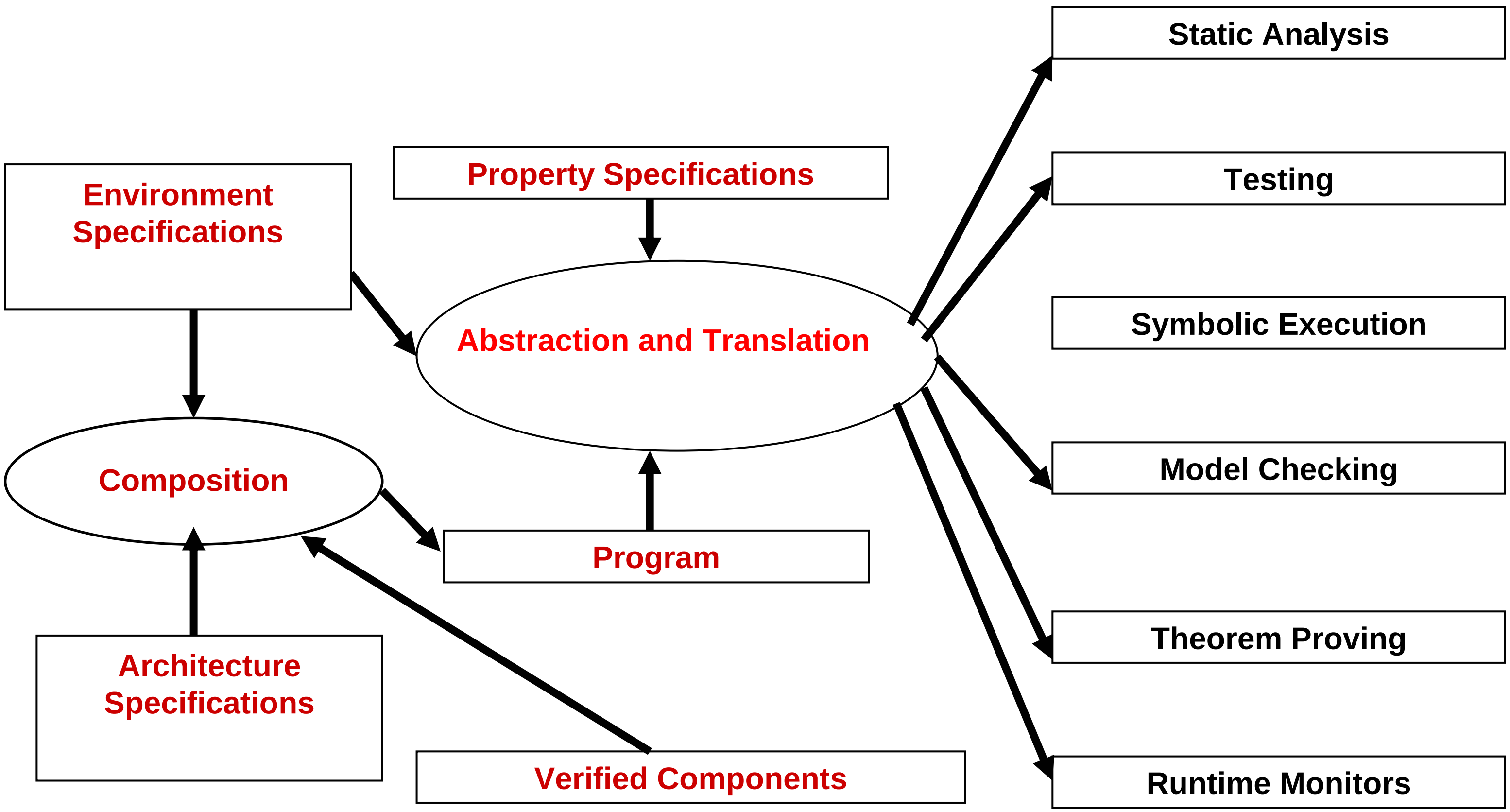
Property Specifications

Jim Browne

Table of Contents

The major part of this lecture is taken from slides by David Garlan

Template for Unification of Verification and Validation



Why to care about specification languages. Reasoning About Executions

- Test Specifications – A test (A test is a specification of an input/output relation.) is a statement about the execution of a (sequential) program for the path taken from a given initial condition. So to test a program we need many tests.
- Desired Specifications – Specifications which make statements about many paths and many initial conditions. If we can establish such a property – we have made a lot of progress.

Property Specifications

- Complete specifications are an alternative statement of the behavior of the implementation in a formal language.
- Partial specifications are statements in a formal language about a program which are deemed critical to meeting requirements.

Property Specifications State of the Art

- There are two main branches of specification languages:
  - Temporal Logics
  - Floyd/Hoare Logics
- There are many different dialects of each language
  - Linear temporal logic (LTL), Computational tree logic (CTL)
  - Java Modeling Language (JML), etc.

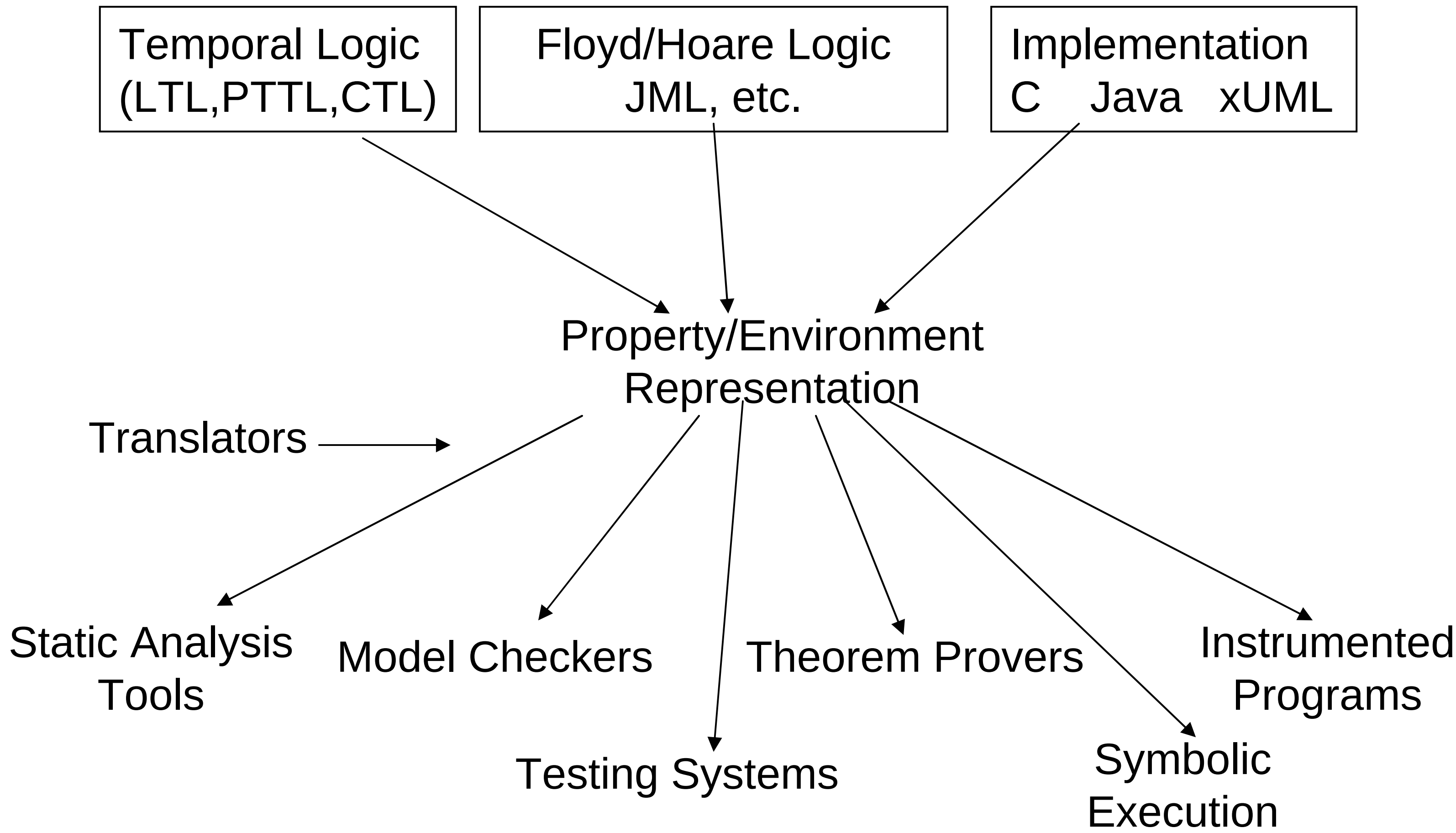
Unified Property Specification Language

- Unified Property Specification Language (UPSL) - Single language for expressing properties which is readily translatable to the input property language of any tool. Example – Accellera PSL for timed systems is readily translatable to multiple model checkers and simulation-based testing systems

Equivalence/Translation Among Specification Languages

- Equivalence and translation – Write in one specification language and translate to other representations
  - Can use existing languages
  - Specific translations may be incomplete.
  - Many translators are needed
  - Possible solution – common intermediate language for property specification languages.

Property Specification and Evaluation



- Properties = Knowledge of Component/System Behavior
    - A property can usually be defined as a state machine.
    - Properties are always defined with respect to an environment for the component/system.
- Environment = Set of properties which generates a closed system for execution or verification of a component or system.
    - Environments should be specifiable as set of properties for an executable entity – sets of allowable input/output sequences
    - Environments may be constraints on inputs

What Types of Properties Should Be Specifiable?

Pre-Condition/Post-Condition pairs for units with identifiable semantics.

Occurrence or non-occurrence of specific states or events.

Sequences of states/events/operations which can or cannot occur => paths.

Security properties => information flow and access control.

Performance properties => time to execute a given path, etc.

Representation Issues

1. Syntax should be consistent with programming system for components/systems
2. Language should provide a library of templates for commonly occurring properties. (Equivalent to libraries of components.)
3. Language should support extending the library of templates.
4. Language should practice separation of concerns.

Pre-Condition => Post-Condition

Specify some subset of the state of the system before the execution of a component and some subset of the state after the execution of a component.

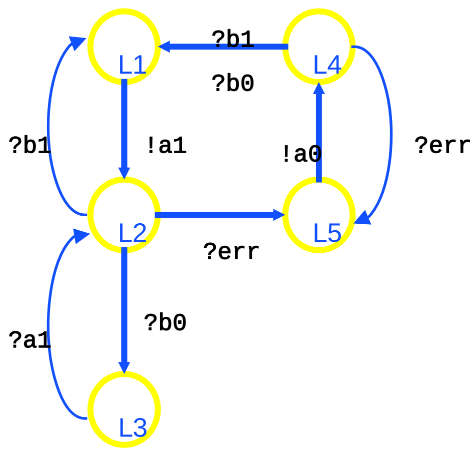
Pre-Condition => Post-Condition pairs can be specified in temporal logics

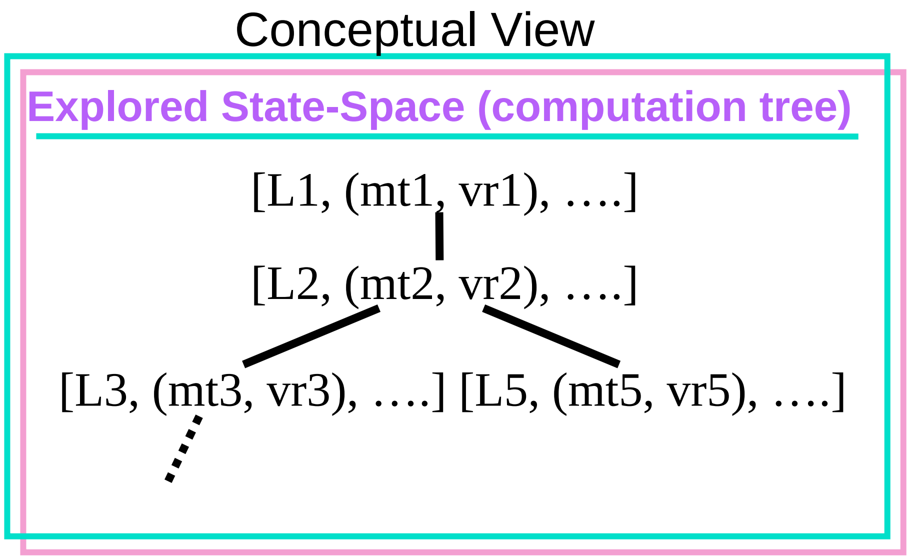
Input/Output Relation is an example of a pre-condition => post-condition

Temporal Logics – Reasoning about Executions

- Specify behaviors along paths (Linear Temporal Logic - LTL)
  - Specify environment such that all paths from all initial conditions are traversed.
- Specify behaviors for all paths on the tree of execution paths traversable from a given set of initial conditions (Computation Tree Logic – CTL)

Reasoning about Executions



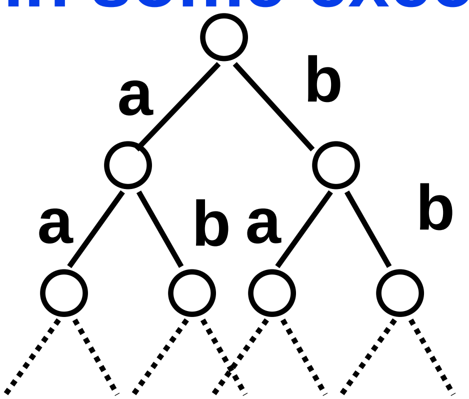


- We want to reason about execution trees
  - > tree node = snap shot of the program's state
- Reasoning consists of two layers
  - > defining predicates on the program states (control points, variable values)
  - > expressing temporal relationships between those predicates

Branching Time Logic

- Branching time logic views a computation as a (possibly infinite) **tree** or dag of **states** connected by atomic **events**
- At each state the outgoing arcs represent the actions leading to the **possible next states** in some execution
- Example:

$$P = (a \rightarrow P) \sqcap (b \rightarrow P)$$



Notation

- **Variant of branching time logic that we will look at is called CTL*, for Computational Tree Logic (star)**
- **In this logic**
  > **A = "for every path"**
  > **E = "there exists a path"**
  > **G = “globally” (similar to □)**
  > **F = “future” (similar to ◊)**

Paths versus States

- A & E refer to paths
  - > A requires that all paths have some property
  - > E requires that at least some path has the property
- G & F refer to states on a path
  - > G requires that all states on the given path have some property
  - > F requires that at least one state on the path has the property

Examples

- **AG p**
  - > For every computation (i.e., path from the root), in every state, p is true
  - > Hence, means the same as □ p
- **EG p**
  - > There exists a computation (path) for which p is always true
  - > Note, unlike LTL not all executions need have this property

Examples

- **AF p**
  - > **For every path, eventually state p is true**
  - > **Hence, means the same as ◊ p**
  - > **Therefore, p is *inevitable***
- **EF p**
  - > **There is some path for which p is eventually true**
  - > **I.e. p is "reachable"**
  - > **Therefore, p will hold *potentially***

More Examples

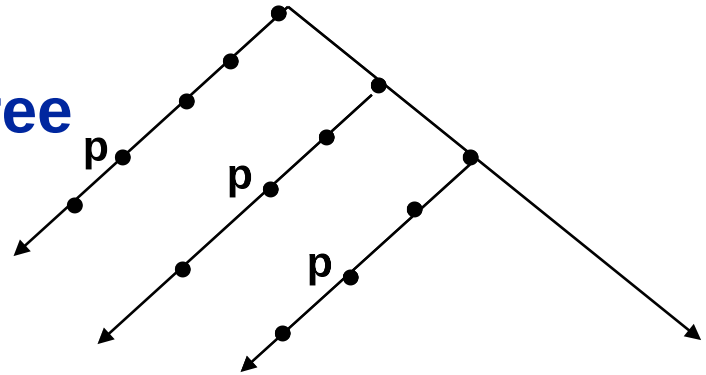
- **EFAG p**
  - > For some computation (E), there is a state (F), such that for all paths from that state (A), globally (G) p is true
- **AGEF halt**
  - > For all computations (A), and for all states in it (G), there is a path (E) along which eventually (F) halt occurs
- **EGEF p**
  - > For some computation (E), for all states in that computation (G), there is a path (E) in which p is eventually (F) true

Other Operators for States

- **Can also have next and until**
  - > **represented as X and U respectively**
  - > **AX p means that for all next states, p will hold**
  - > **E[p U q] means that for some path there is a state where q holds and p holds in all states up to that state**

More Examples

- Show that EGEF p is the same as EGF p or provide a counter example to illustrate why not
  - > EGEF p means that there is a path such that from all states, there is a path such that p is eventually true
  - > EGF p means that there is a path such that from all states, p is eventually true <u>in that path</u>
  - > Consider the following tree
    - First one is true
    - Second one is not

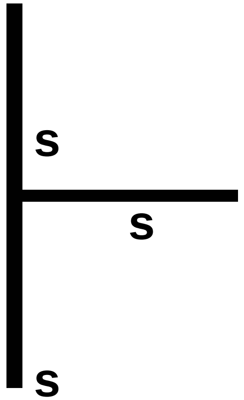


CTL

- **In some versions the symbols are required to occur in pairs of the form**
  - > **AG, AF, EG, EF**
  - > **Called CTL (no star)**
  - > **Important restriction for tools such as model checkers**

Traffic Controller

- Consider a traffic controller on a north-south highway with a road off to the east



- Each road has a sensor that goes to true when a car crosses it
- For simplicity, no north or south bound car will turn

Traffic Controller

- **To reason about them, we name the sensors**
  - > **N (north)**
  - > **S (south)**
  - > **E (east)**
- **We also name the output signals at each end of the intersection**
  - > **N-go (cars from the north can go)**
  - > **S-go (cars from the south can go)**
  - > **E-go (cars from the east can go)**

Safety Property

- **If cars from the east have a go-signal, then no other car can have a go-signal**

$$AG \neg (E\text{-}go \wedge (N\text{-}go \vee S\text{-}go))$$

Liveness properties

- If a sensor registers a car, then the car will be able to go through the intersection

$$AG\ (\neg \text{N-go} \wedge N \rightarrow AF\ \text{N-go})$$
$$AG\ (\neg \text{S-go} \wedge S \rightarrow AF\ \text{S-go})$$
$$AG\ (\neg \text{E-go} \wedge E \rightarrow AF\ \text{E-go})$$

- If the above are true, then the controller is free of deadlock

Efficiency

- **Since north and south bound cars can safely pass by each other we can state a possibility**

$$\mathbf{EF\ (N\text{-}go \wedge S\text{-}go)}$$

Fairness

- **We can't have a car stop in the intersection**

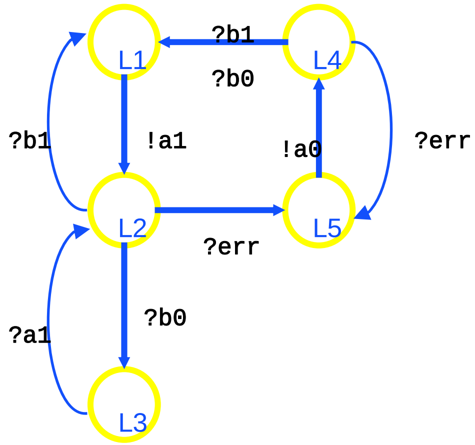
$$AG \neg (N\text{-}go \wedge N)$$
$$AG \neg (S\text{-}go \wedge S)$$
$$AG \neg (E\text{-}go \wedge E)$$

Yet More Temporal Logics

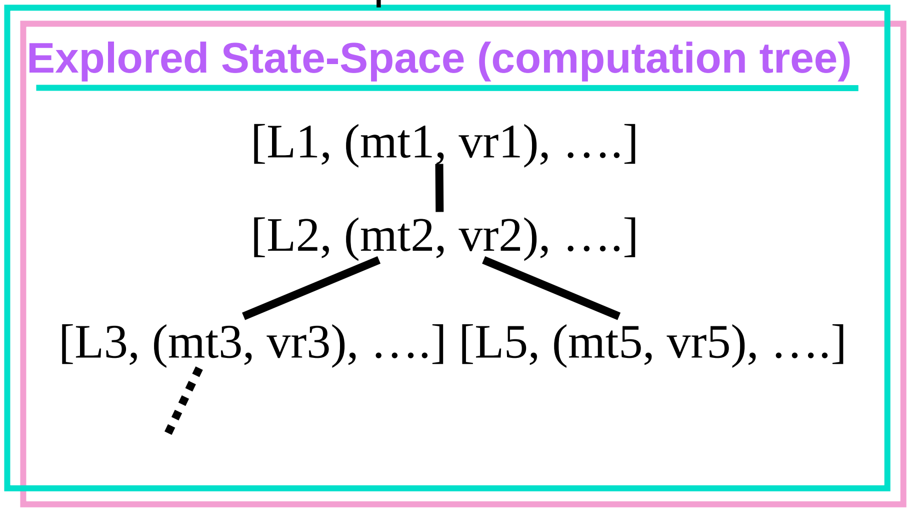
- The logic we've used so far is concerned with instances of state
  - > assertions about a future state(s)
  - > predicate is applied to each selected state
- What about contiguous collections of states?

- Interval temporal logic
  - > assertions over intervals of time
  - > have to worry about overlapping intervals

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

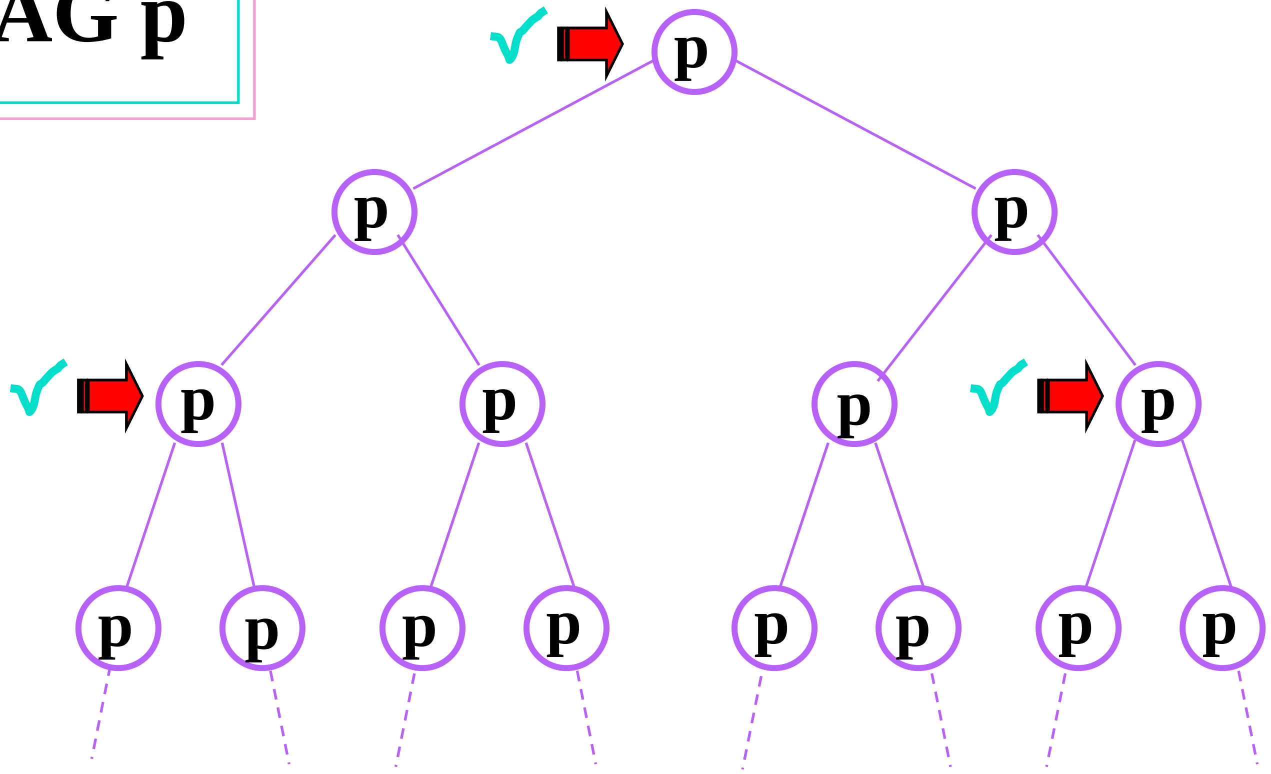
`EX p` …there *Exists* a path where p holds in the *neXt* state

`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

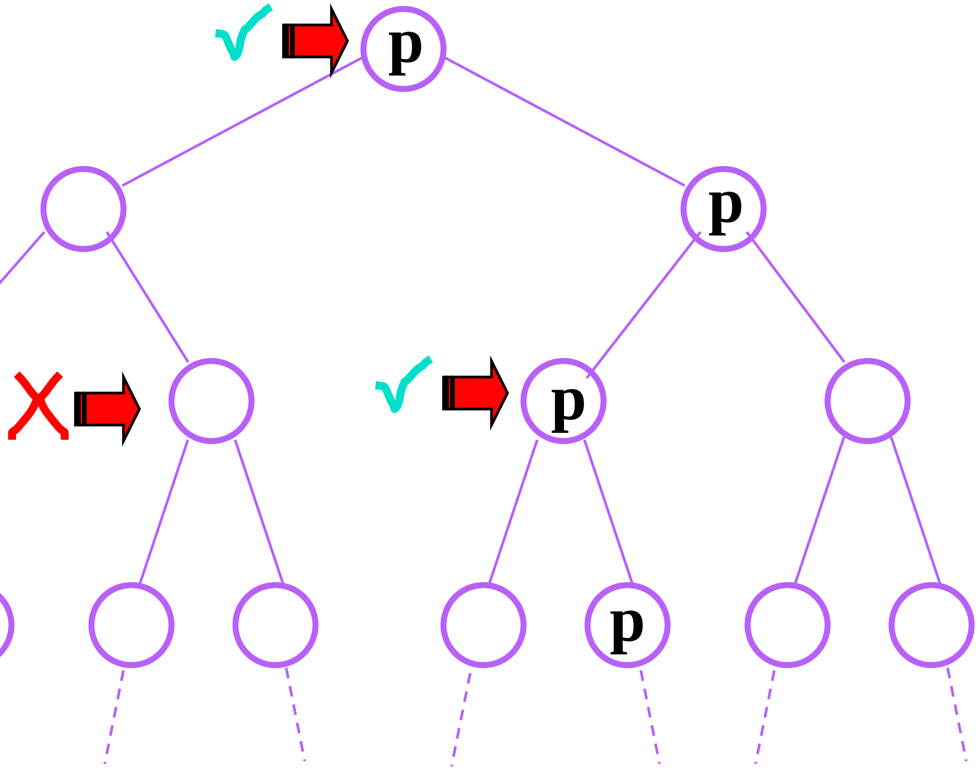
Computation Tree Logic

AG p



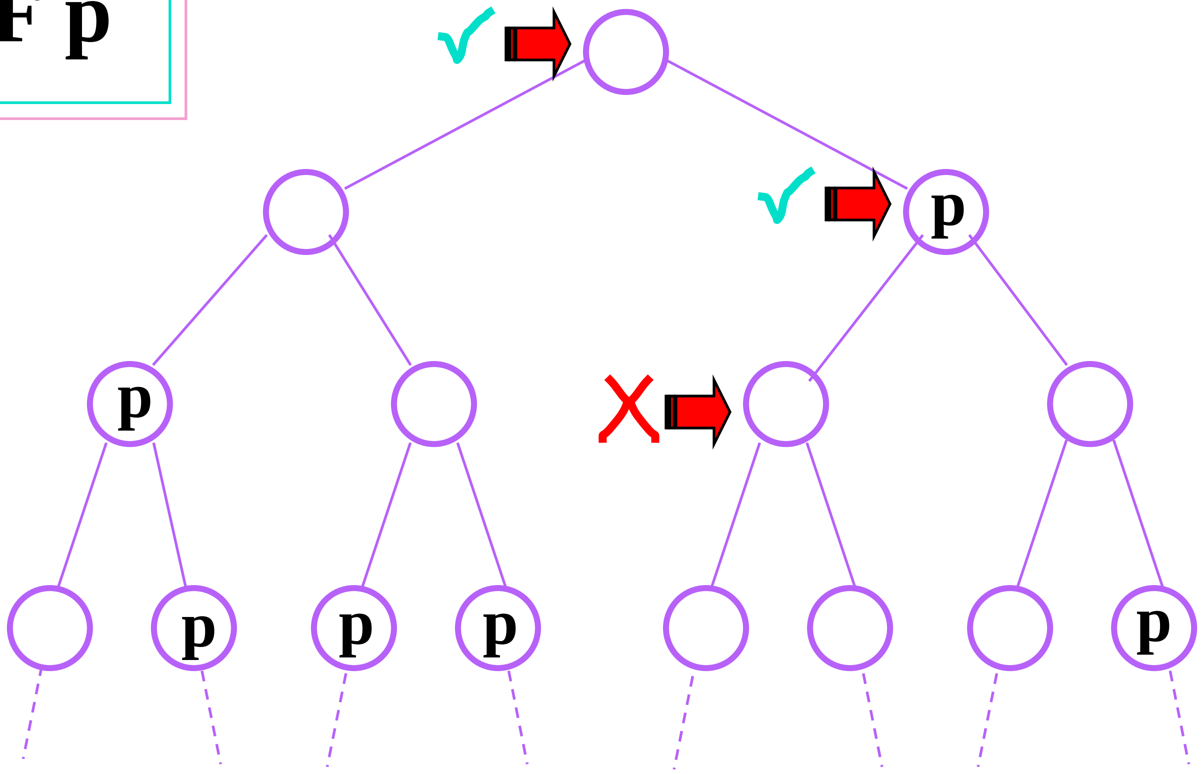
Computation Tree Logic

EG p



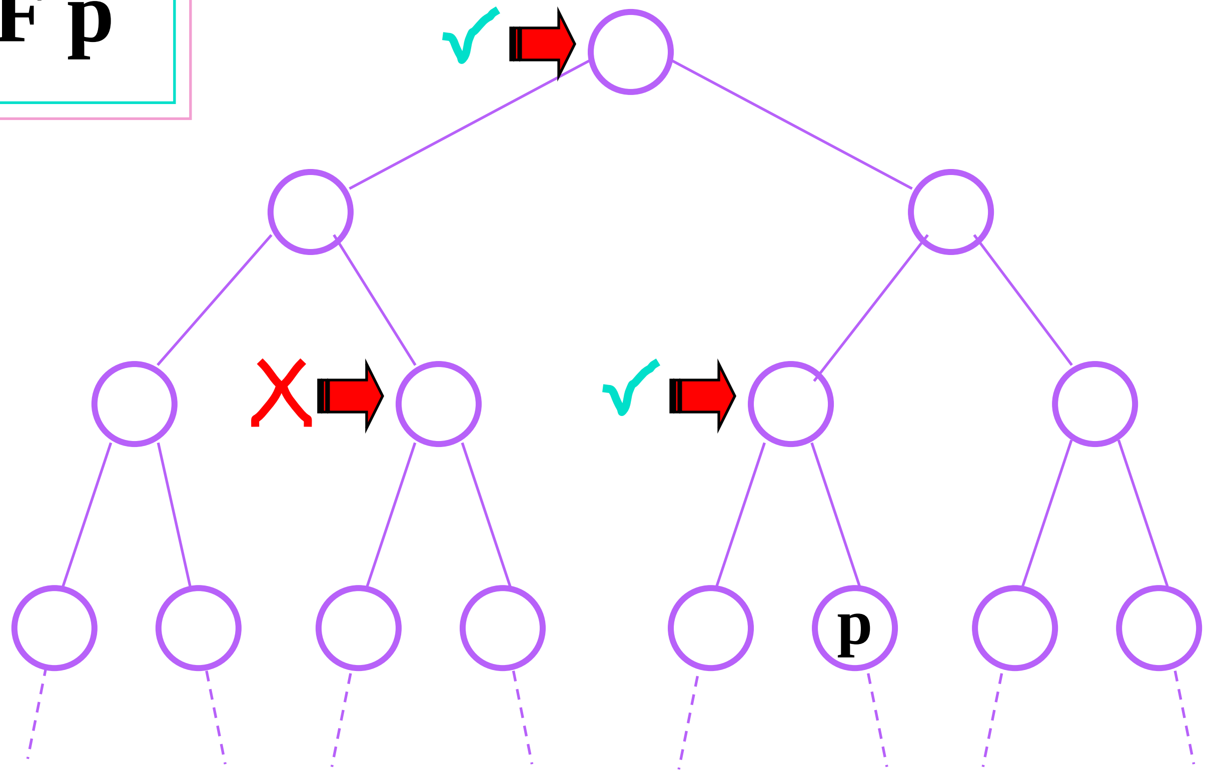
Computation Tree Logic

AF p



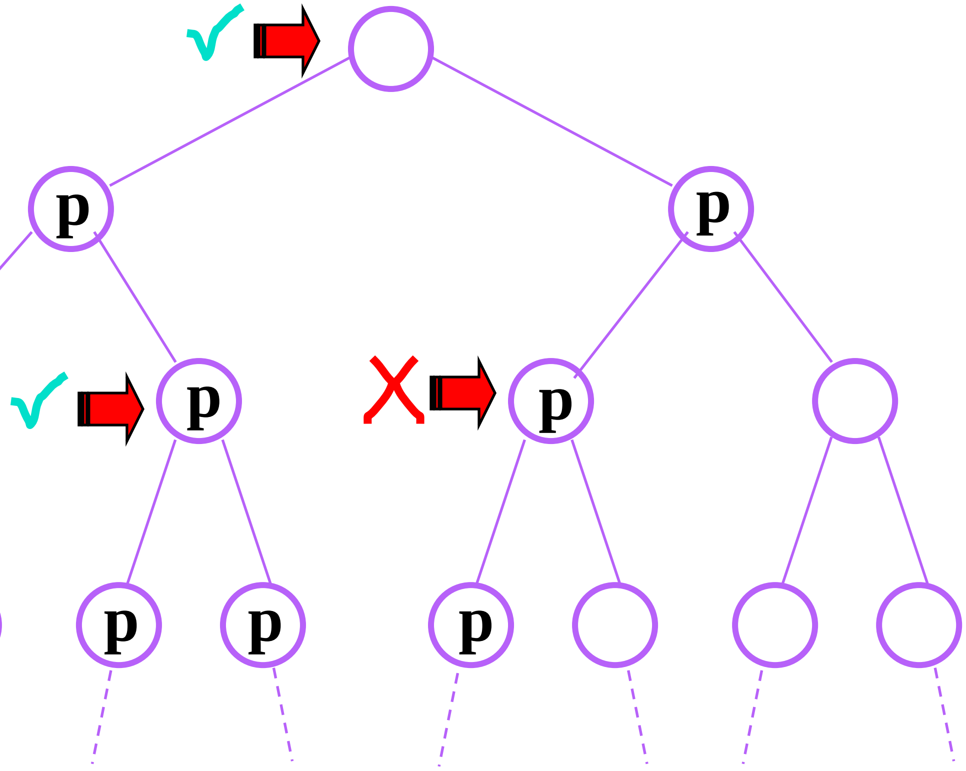
Computation Tree Logic

EF p



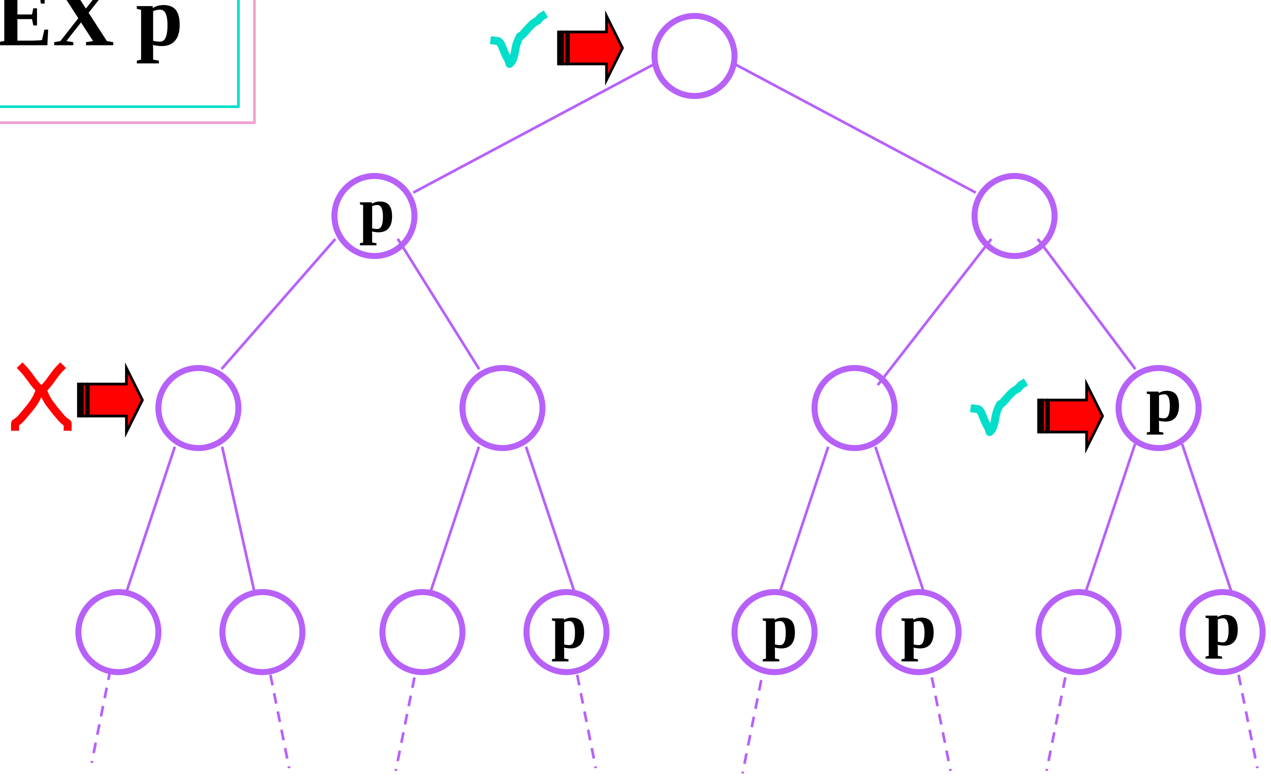
Computation Tree Logic

AX p



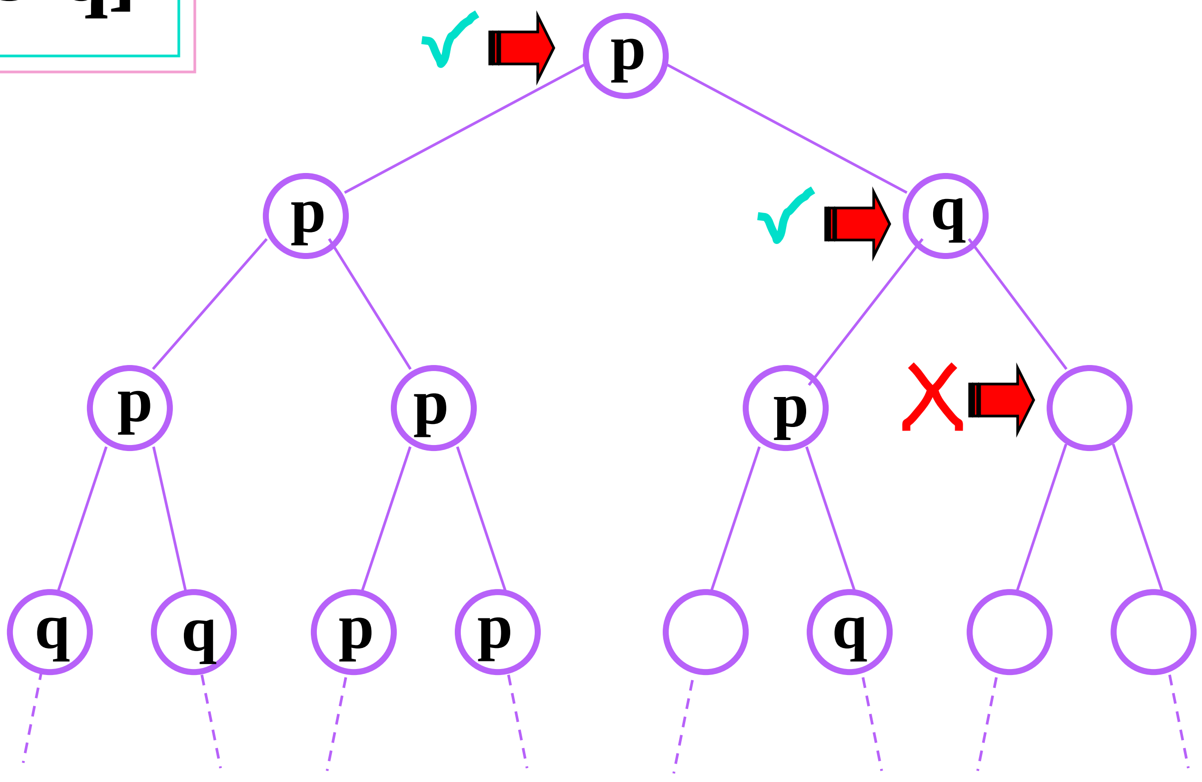
Computation Tree Logic

EX p



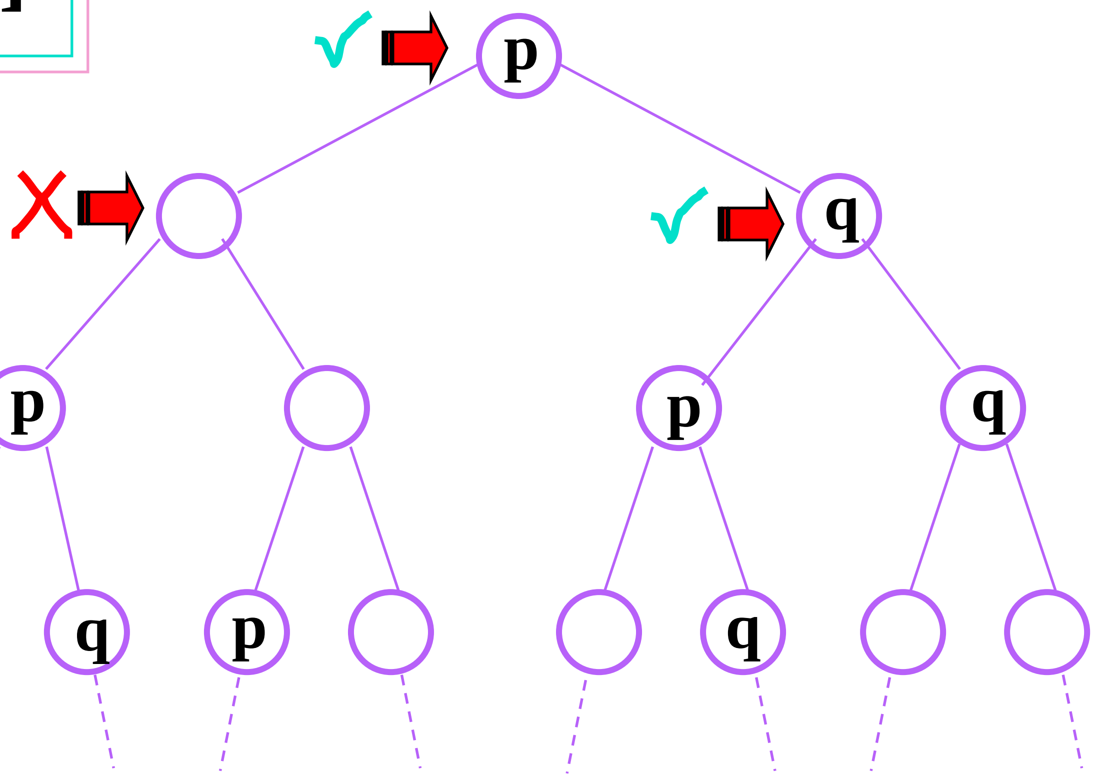
Computation Tree Logic

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

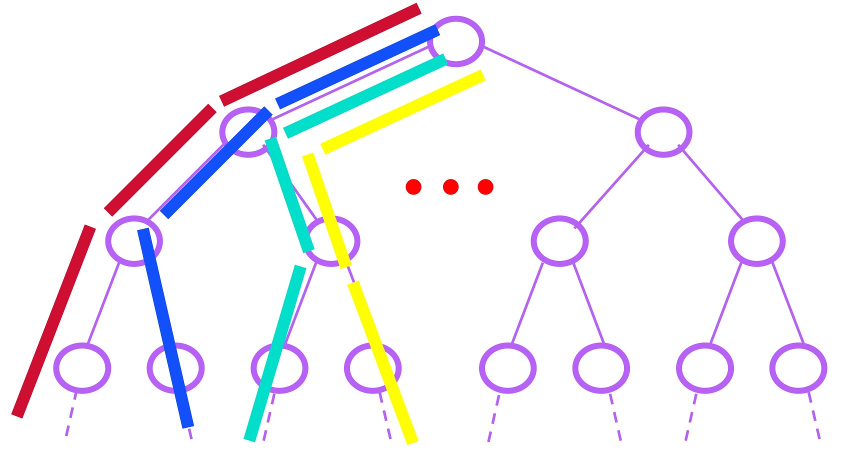
AG((floor=2 && direction=up && button5pressed)
-> A[direction=up U floor=5])

CTL Notes

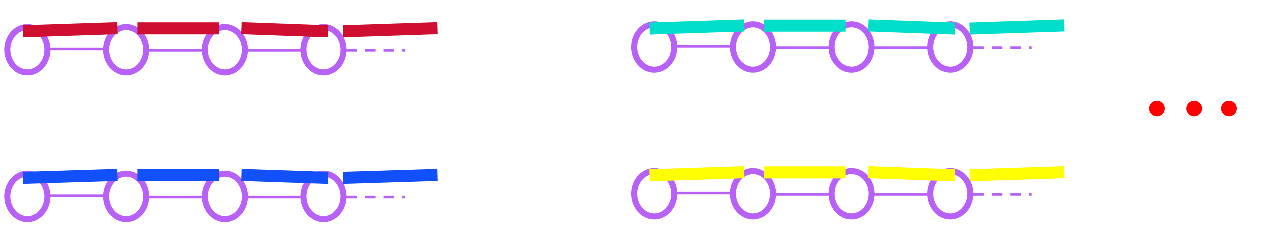
- Invented by E. Clarke and E. A. Emerson (early 1980's)
- Specification language for Symbolic Model Verifier (SMV) model-checker
- SMV is a *symbolic* model-checker instead of an *explicit-state* model-checker
- Symbolic model-checking uses Binary Decision Diagrams (BDDs) to represent boolean functions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



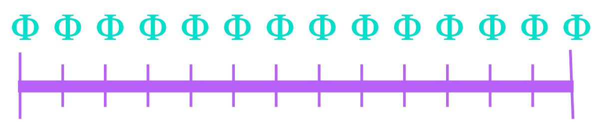
Linear Temporal Logic (LTL)

Syntax

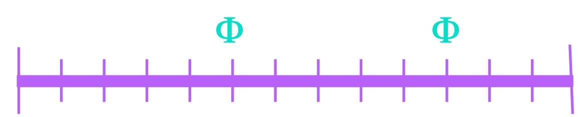
```
Φ ::=  P                                          …primitive propositions
    | !Φ  | Φ && Φ | Φ || Φ | Φ -> Φ             …propositional connectives
    | []Φ | <>Φ    | Φ U Φ  | X Φ                  …temporal operators
```

Semantic Intuition

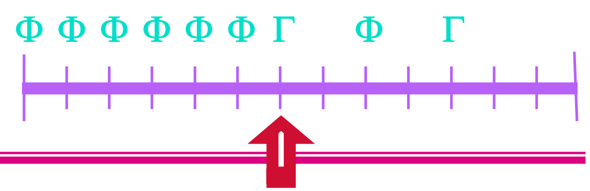
[]Φ …*always* Φ



<>Φ …*eventually* Φ



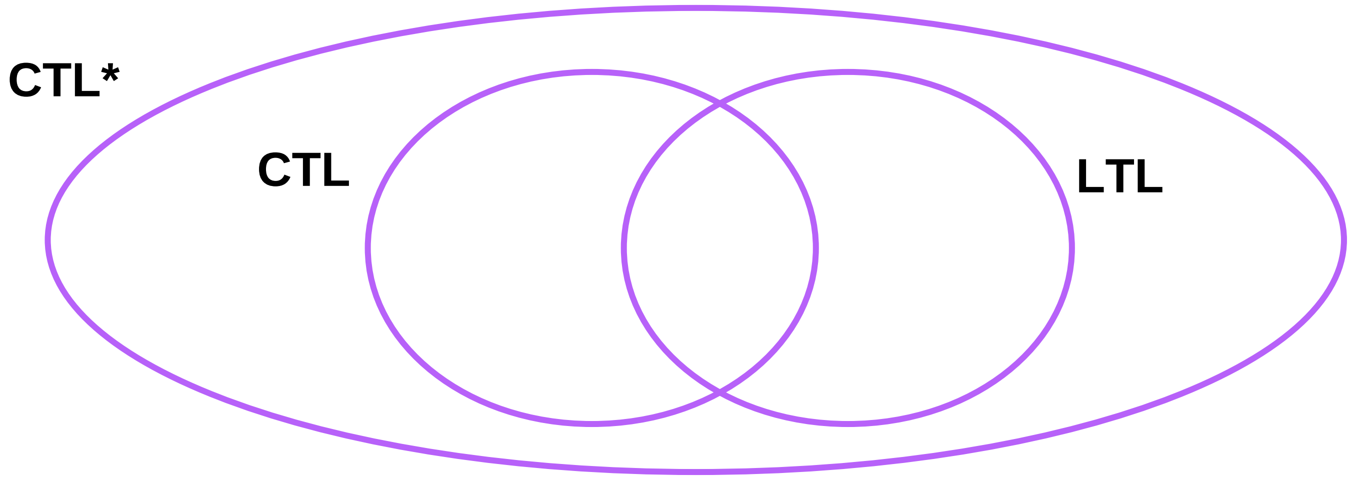
Φ U Γ …Φ *until* Γ



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

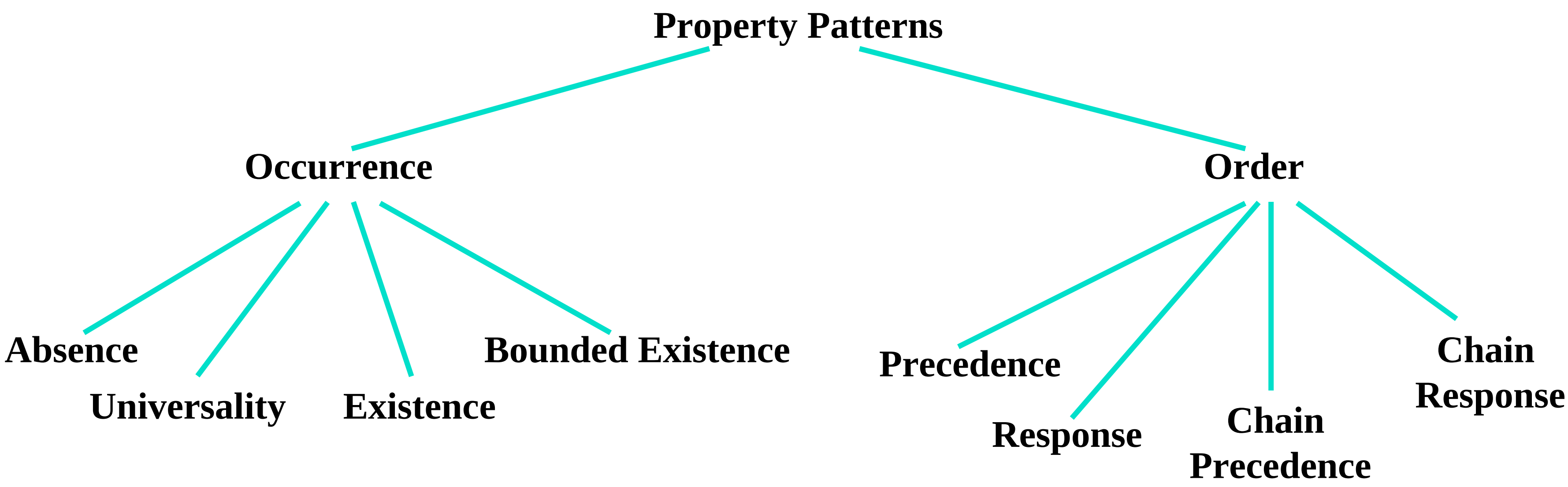
Example: **action Q must respond to action P**

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

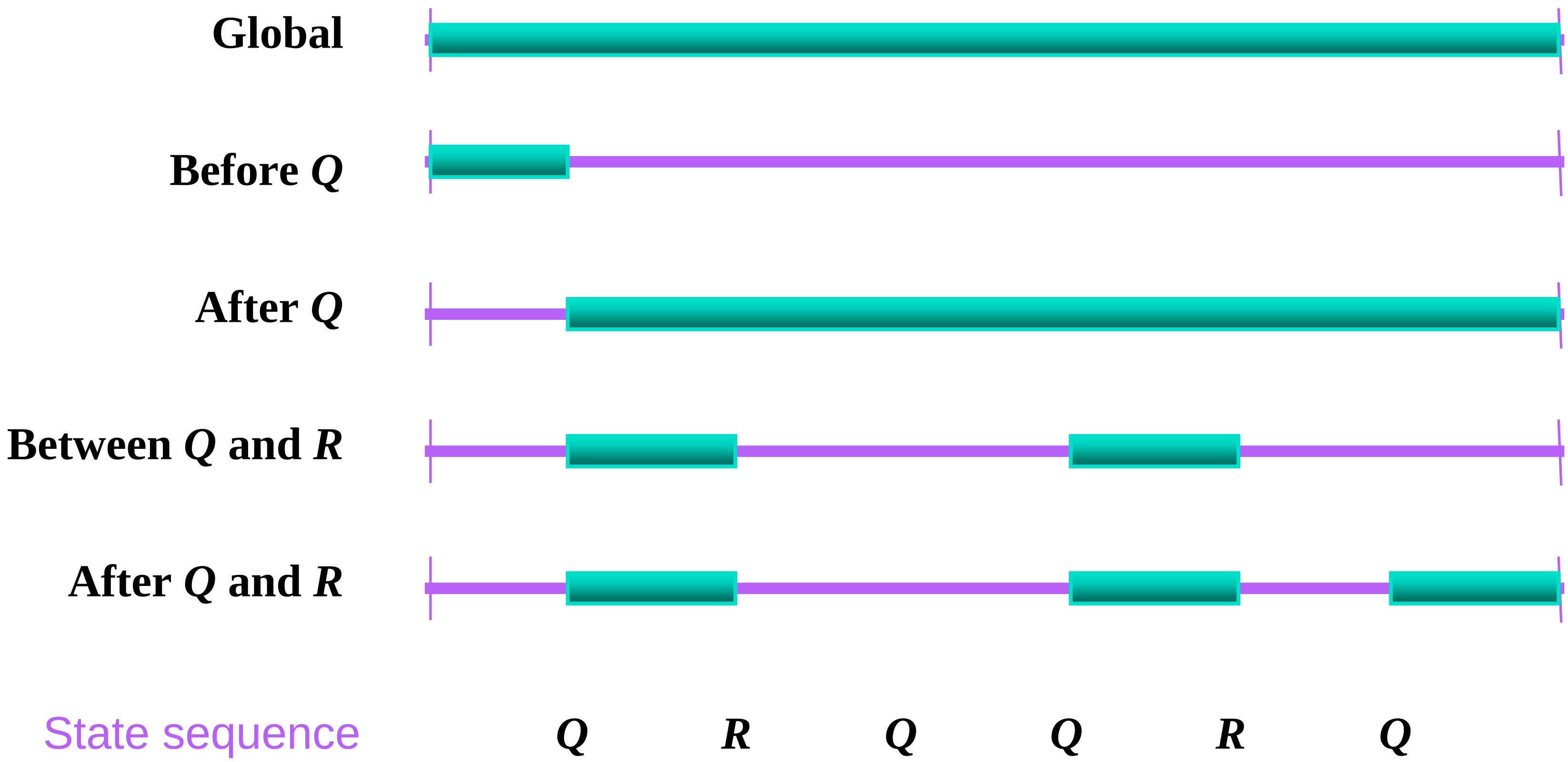
Occurrence Patterns

- 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)]) | 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...

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

Assertions

- **Basics**
- **JML**

Verification Conditions,

Hoare Logics,

Assertations/Specifications

Assertions/Specifications

- Precise, formal specifications concerning the behavior of some unit of code
- Usually written in a language separate from programming language.
- Used for documentation, verification, runtime monitoring, testing

Assertions - Types

Invariants (from Wikipedia) - A predicate that will always keep its truth value throughout a specific sequence of operations, is called (an) **invariant** to that sequence.

State Invariants

Loop Invariants

Pre-conditions/Post-Conditions - Pre- and post-conditions are constraints that define a contract that an implementation of the operation has to fulfill. A precondition must hold when an operation is called, a postcondition must be true when the operation returns.

-

Invariants

- **Definition**
  > **An *invariant* is a property that is always true of an object's state (when control is not inside the object's methods).**
- **Invariants allow you to define:**
  > **Acceptable states of an object, and**
  > **Consistency of an object's state.**

```
//@ public invariant !name.equals("") && weight >= 0;
```

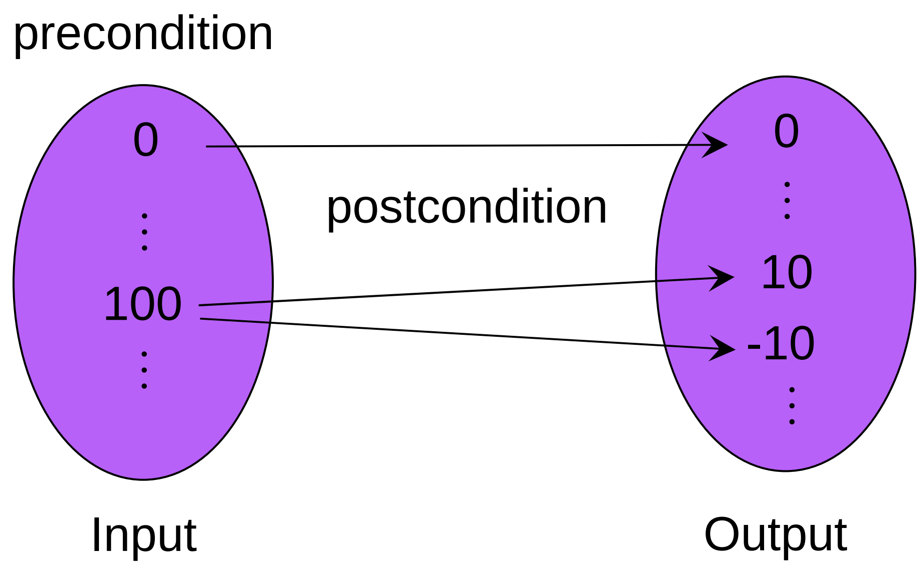
Pre and Postconditions

- **Definition**
  - > **A method or function *precondition* says what must be true to call it.**
  - > **A method or function *normal postcondition* says what is true when it returns normally (i.e., without throwing an exception).**
  - > **A method or function *exceptional postcondition* says what is true when a method throws an exception.**

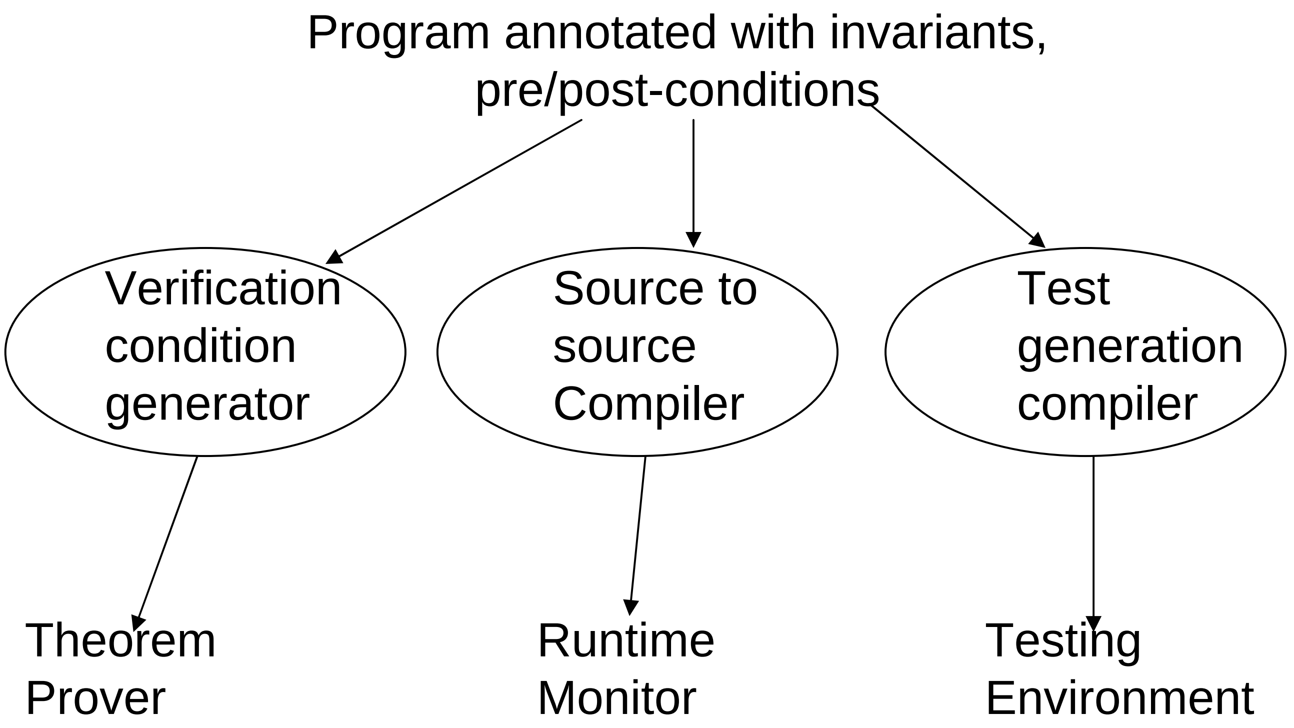
```
//@ signals (IllegalArgumentException e) x < 0;
```

Relational Model of Methods

- **Can think of a method as a relation:**
  Inputs ↔ Outputs



Assertions – How Used



Relationship to Temporal Logic

Temporal logic predicates are same as assertion/specfication predicates.

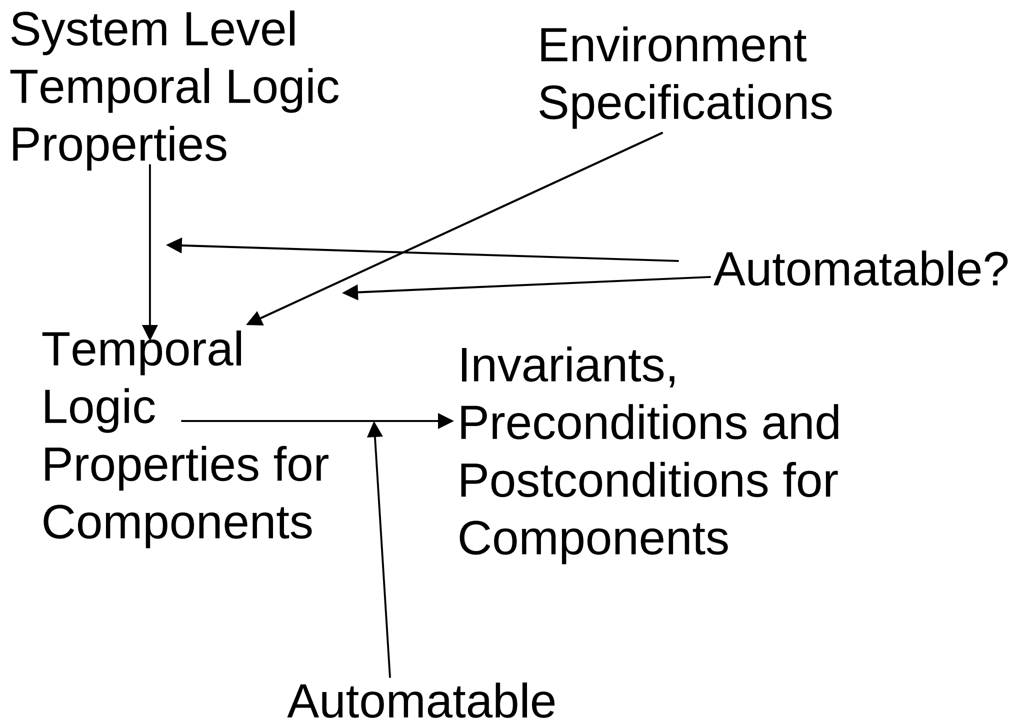
Assertion specifications are local with respect to some code unit (composed by Hoare logic rules)

Temporal logic predicates apply to states during execution of some code unit and are defined on paths or structures of paths

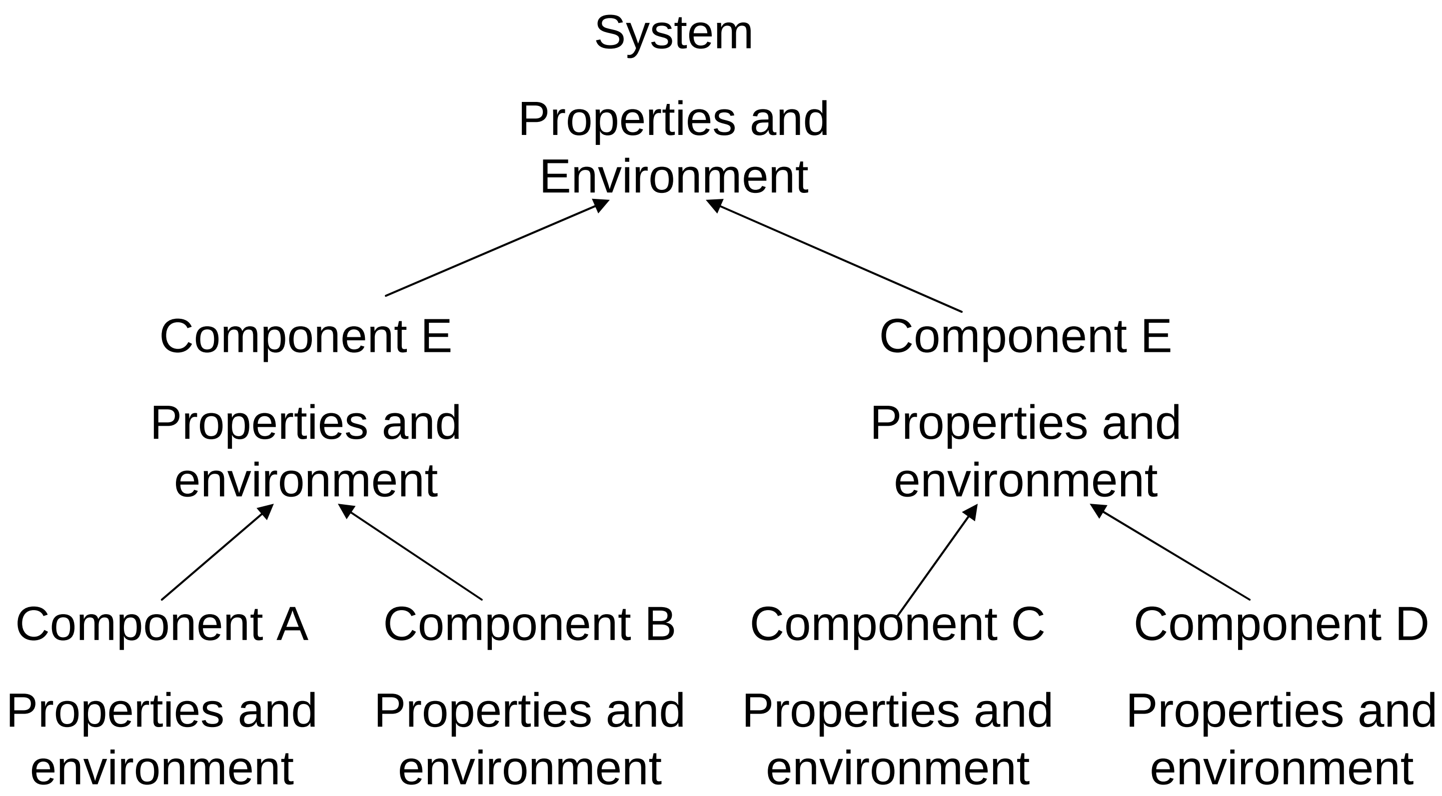
Relationship to Temporal Logic

- Temporal logic properties for code units can be composed into properties for larger code units
- System level temporal logic can be decomposed into component level properties.
- Component level temporal logic properties can be translated into invariants, preconditions and postconditions

Relationship to Temporal Logic



Temporal Logic Composition



Decomposition of I/P/P Specifications

System Level Preconditions == Environment Specifications

↓ ← Automatable?

Component Level Preconditions

↓

Component Level Invariants and Postconditions

Composition of I/P/P Specifications

System Level Preconditions == Environment Specifications

↑ ← ??

Component Level Preconditions

↑

Component Level Invariants and Postconditions

Composition of I/P/P Specifications

System

Invariants, P/P Conditions

Hoare Rules for composition should apply. Automatable??

Component E

Invariants, P/P Conditions

Component E

Invariants, P/P Conditions

Component A

Invariants, P/P Conditions

Component B

Invariants, P/P Conditions

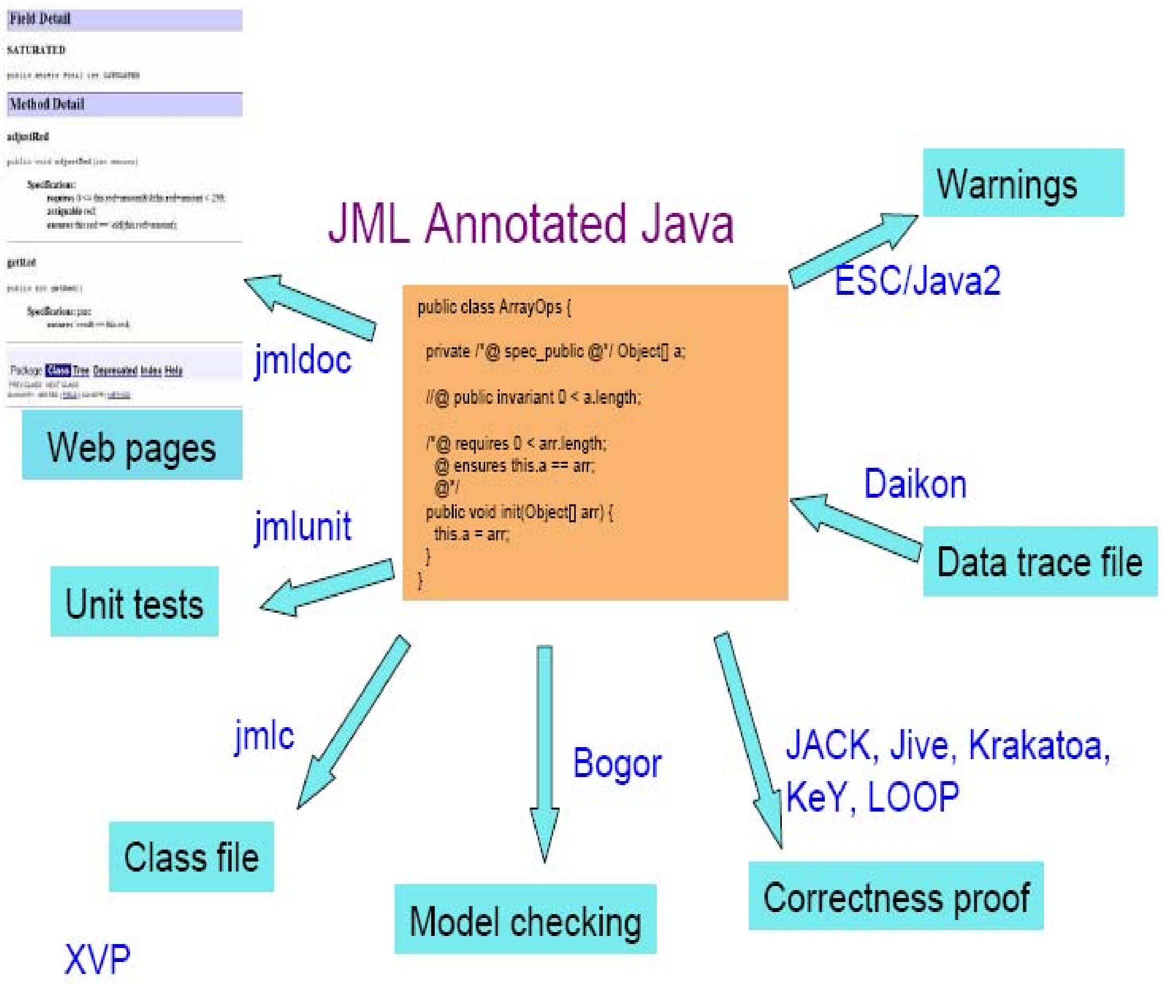
Component C

Invariants, P/P Conditions

Component D

Invariants, P/P Conditions

Tools for JML-Based Verification



Java Modeling Language

Ilustrate Assertions with Java Modeling Language

> Hoare-style (Contracts).
> Method pre- and postconditions.
> Invariants.

Java Modeling Language

- JML Annotations/Assertions
- **Top-level in classes and interfaces:**
  - > invariant
  - > spec_public
  - > nullable
- **For methods and constructors:**
  - > requires
  - > ensures
  - > assignable
  - > pure

Example JML

```
public class ArrayOps {
private /*@ spec_public @*/ Object[] a;
//@ public invariant 0 < a.length;
/*@ requires 0 < arr.length;
  @ ensures this.a == arr;
  @*/
public void init(Object[] arr) {
this.a = arr;
}
```

spec_public, nullable, and invariant

- spec_public
  > **Public visibility.**
  > **Only public for specification purposes.**
- nullable
  > **field (and array elements) may be null.**
  > **Default is** non_null**.**
- invariant **must be:**
  > **True at end of constructor.**
  > **Preserved by each method.**

requires and ensures

- requires **clause:**
  - > **Precondition.**
  - > **Obligation on callers, after parameter passing.**
  - > **Assumed by implementor.**
- ensures **clause:**
  - > **Postcondition.**
  - > **Obligation on implementor, at return.**
  - > **Assumed by caller.**

assignable and pure

- assignable
  - > **Frame axiom.**
  - > **Locations (fields) in pre-state.**
  - > **New object fields not covered.**
  - > **Mostly checked statically.**
  - > **Synonyms:** modifies**,** modifiable
- pure
  - > **No side effects.**
  - > **Implies** assignable ****nothing
  - > **Allows method's use in specifications.**

Redundant Clauses

- ensures_redundantly
  - > **Alerts reader.**
  - > **States something to prove.**
  - > **Must be implied by:**
    - » ensures **clauses,**
    - » assignable **clause,**
    - » invariant**, and**
    - » **JML semantics.**
- **Also** requires_redundantly**, etc.**

Formal Specifications

- **Formal assertions are written as Java expressions, but:**
  - > **Can't have side effects**
    - » **No use of =, ++, --, etc., and**
    - » **Can only call *pure* methods.**
  - > **Can use some extensions to Java:**

Syntax	Meaning
\result	result of method call
a ==> b	a implies b
a <== b	b implies a
a <==> b	a iff b
a <=!=> b	!(a <==> b)
\old(E)	value of E in the pre-state

BoundedStack's Data and Invariant

BoundedStack's Data and Invariant

```
public class BoundedStack {
private /*@ spec_public nullable @*/
Object[] elems;
private /*@ spec_public @*/ int size = 0;
//@ public invariant 0 <= size;
/*@ public invariant elems != null
@ && (\forall int i;
@ size <= i && i < elems.length;
@ elems[i] == null);
@*/
```

BoundedStack's Constructor

```
/*@ requires 0 < n;
@ assignable elems;
@ ensures elems.length == n;
@*/
public BoundedStack(int n) {
elems = new Object[n];
}
```

BoundedStack's push Method

```
BoundedStack's push Method
/*@ requires size < elems.length1;
@ assignable elems[size], size;
@ ensures size == \old(size+1);
@ ensures elems[size1] == x;
@ ensures_redundantly
@ (\forall int i; 0 <= i && i < size1;
@ elems[i] == \old(elems[i]));
@*/
public void push(Object x) {
elems[size] = x;
size++;
}
```

BoundedStack’s pop Method

BoundedStack’s pop Method

```
/*@ requires 0 < size;
@ assignable size, elems[size1];
@ ensures size == \old(size1);
@ ensures_redundantly
@ elems[size] == null
@ && (\forall int i; 0 <= i && i < size1;
@ elems[i] == \old(elems[i]));
@*/
public void pop() {
size;
elems[size] = null;
}
```

BoundedStack's top Method

BoundedStack's top Method

```
/*@ requires 0 < size;
@ assignable \nothing;
@ ensures \result == elems[size1];
@*/
public /*@ pure @*/ Object top() {
return elems[size1];
}
}
```