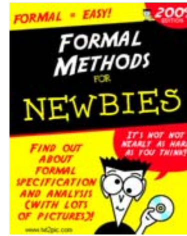
CISC422/853: Formal Methods in Software Engineering: Computer-Aided Verification



Topic 7: Specifying, or How to Describe How the System Should (or Should Not) Behave

Juergen Dingel
Feb, 2009

Readings:

- Spin book: Chapter 4 (Defining Correctness Claims), Chapter 6 (Automata and Logic)
- Course notes on CTL

Outline

- Formal specifications
- Types of formal specifications:
  - assertions
  - invariants
  - safety and liveness properties
- How to express safety properties:
  - FSAs, regular expressions, Never Claims
- How to express liveness properties:
  - progress labels, Buechi Automata, Never Claims, Linear Temporal Logic, Computation Tree Logic
- How to manage the complexity of specifications:
  - specification patterns

(Formal) Specifications

- What is a specification?
- What is a formal specification?
- Properties of good formal specifications?
  - As precise and detailed as necessary, and as abstract (i.e., unconstraining) as possible
  - Consistent
  - Correct (internally, externally)
- Why use formal specifications?
  - Unambiguous
  - Sometimes more succinct
  - Amenable to automatic analysis

Observables

- "Atomic propositions" used in a specification
- In BIR
  - global and local variables (in their scope)
  - existential (anonymous) thread program counter

    `Property.existsThread(t, loc5)`
- In PROMELA
  - global and local variables
  - end-states, progress states, and accept states
    - needed for expression of liveness (progress) properties
    - E.g., non-progress through reserved boolean variable **`np_`** (false in s iff at least one process is at a control flow state marked with a progress label)
    - more on these later
  - process ids through reserved variable **`_pid`**

Types of Formal Specifications for Concurrent and Reactive Systems

- Assertions
- Invariants
- Safety properties
- Liveness properties

Assertions

- Express a property of observables at particular location
- Most basic formal specification; already used by John von Neumann in 1947
- In BIR and Promela: assert(b);
- What kind of correctness claim does an assertion make, that is, what does it mean if there is
  - no assertion violation?:
    "No matter along which path control has reached the location of the assertion, the boolean expression in the assertion evaluates to true at that location"
  - an assertion violation?:
    "There is at least one execution such that the boolean expression in the assertion does not evaluate to true at that location"

Example:

```
thread T() {
  …
  loc loc7:
     when b do {
        …
        assert(x>y);
        …
     }
  …
}
```


Example 1: Simple Race Condition

```
byte state = 1;
active proctype A()
{         (state == 1) -> state++;
          assert(state == 2)
}
active proctype B()
{         (state == 1) -> state--;
          assert(state == 0)
}
```

```
$ spin -a simple.pml
$ gcc -o pan pan.c
$ ./pan -E          # -E means ignore invalid endstate errors...
pan: assertion violated (state==2) (at depth 6)
pan: wrote simple.pml.trail
...
```

```
$ spin -t -p simple.pml
  1:    proc  1 (B) line   7 "simple.pml" (state 1) [((state==1))]
  2:    proc  0 (A) line   3 "simple.pml" (state 1) [((state==1))]
  3:    proc  1 (B) line   7 "simple.pml" (state 2) [state--]
  4:    proc  1 (B) line   8 "simple.pml" (state 3) [assert((state==0))]
  5:    proc  0 (A) line   3 "simple.pml" (state 2) [state++]
spin: line   4 "simple.pml", Error: assertion violated
spin: text of failed assertion: assert((state==2))
```

[Source: spinroot.com]

Example 2: Checking Mutual Exclusion Using Assertions

- Does protocol below ensure mutual exclusion and deadlock freedom?
- How can we check this using Bogor or Spin?

```
system MuxTry {
boolean flag1;
boolean flag2;

thread T1 () {
loc loc0:
do {flag1 := true;} goto loc2;

loc loc2:
when (!flag2) do {} goto loc3;

loc loc3:
do {} goto loc4;

loc loc4:
do {flag1 := false;} goto loc0;
}
```

```
thread T2 () {
loc loc0:
do {flag2 := true;} goto loc2;

loc loc2:
when (!flag1) do {} goto loc3;

loc loc3:
do {} goto loc4;

loc loc4:
do {flag2 := false;} goto loc0;
}
```

critical regions

Example 2: Checking Mutual Exclusion Using Assertions (Cont'd)

To check mutual exclusion, instrument protocol as follows:

```
system MuxTry {
boolean flag1;
boolean flag2;
int c;

thread T1 () {
loc loc0:
do {flag1 := true;} goto loc2;

loc loc2:
when (!flag2) do {} goto loc3;

loc loc3:
do {c := c+1; assert(c==1);}
goto loc4;

loc loc4:
do {c := c-1; flag1 := false;}
goto loc0;
}
```

```
thread T2 () {
loc loc0:
do {flag2 := true;} goto loc2;

loc loc2:
when (!flag1) do {} goto loc3;

loc loc3:
do {c := c+1; assert(c==1);}
goto loc4;

loc loc4:
do {c := c-1; flag2 := false;}
goto loc0;
}
```

critical regions

What about deadlock freedom?

Assertions in Java

- Java 1.4 also supports assertions
- What does it mean if a Java assertion is
  - violated?
  - not violated?
- What's the difference between assertions in Bogor/Spin and Java?

Invariants

- Express property of observables that holds at every location
- What kind of correctness claim does an invariant make, that is, what does it mean if there is
  - no invariant violation?:
    "At all locations along all executions of the system, the property holds"
  - an invariant violation?:
    "There is at least one location along an execution such that the property does not hold at that location"
- How do invariants compare to
  - assertions?
  - "loop invariants" in Hoare Logic?

Multiplication Example

Consider a simple program with a loop invariant

```
// assume parameters m and n
count := m;
output := 0;

// loop invariant: m * n == output + (count * n)
while (count > 0) do {
   output := output + n;
   count := count – 1;
}
```

[Source: CIS842 @ KSU]

Multiplication Example

BIR Version:

```
system Mult {
int m;
int n;
int count;
int output;

main thread Main () {
loc loc0:
  do {m := (int (0,255)) 5;
      n := (int (0,255)) 4;
      count := m;
      output := (int (0,255)) 0;
      start T1();
  } return;
}
```

Using two threads is unnatural, but the motivation will be clear in a moment…

```
thread T1 () {
loc loc0:
  when (count > 0)
   do {output := output + n;
         count := count - 1;}
  goto loc0;
  when (count == 0) do {}
  return;
}
```

Remember:
No interleaving between these two assignments!

Now, …how to program the check of the invariant?

Checking Invariants

- To check invariant *I* on a program with the threads *Main, T1, …, Tn* add an assertion of *I* as the last transition of *Main*:

```
main thread Main ()
 …
 …
loc locAssert:
  do {assert (I);}
  return;
```

- Why does this work?
  - Model-checker will explore all possible interleavings between *Main* and each *Ti*
  - Thus, the assertion statement will get interleaved (on some trace) between every pair of execution steps of each *Ti* and thus checking the invariant on every state along every possible execution of *T1, …, Tn*

Multiplication Example: Checking Invariants

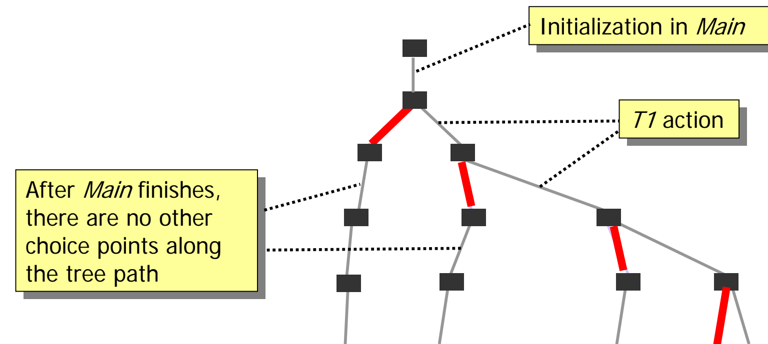
```
system Mult {
 …
 main thread Main () {
  loc loc0:
   do {m := (int (0,255)) 5;
       n := (int (0,255)) 4;
       count := m;
       output := (int (0,255)) 0;
       start T1();
   }
   goto loc1;

  loc loc1:
   do {assert (m*n ==
            output+(count*n));}
   return;
}
```

Assertion added

```
thread T1 () {
 loc loc0:
  when (count > 0)  do {
        output := output + n;
        count := count - 1;
  }
  goto loc0;
  when (count == 0)  do {}
  return;
}
```


Checking Invariants



In other words, there exists a path where we do 0 steps of *T1* then check *I*, there exists a path where we do 1 step of *T1* then check *I*, there exists a path where we do 2 steps of *T1*, then check *I*, etc.

Checking Invariants in Spin

```
mtype = { p, v };
chan sem = [0] of { mtype };
byte count;
active proctype semaphore() {
  do
  :: sem!p -> sem?v
  od
}
active [5] proctype user() {
  do
  :: sem?p -> count++;
            /* critical section */
            count--;
            sem!v
  od
}
```

```
active proctype invariant() {
   assert(count <= 1)
}
```


Increase in number of states: x 3

```
active proctype invariant() {
   do :: assert(count <= 1) od
}
```


Increase in number of states: x 1

Assertions and Invariants

- Assume location l in t can be characterized by $p_{t\ at\ l}$.
  - Then, checking for assertion q at l in t is equivalent to checking the invariant $p_{t\ at\ l}$ -> q
  - $p_{t\ at\ l}$ is also called filter
  - *"Assertions are invariants with non-trivial filters"*

```
thread t() {
  ...
  loc loci: do {assert(q); ... }
  ...
}
main thread Main() {
  loc loc0: do {...
              start t();}
           return;
}
```

```
thread t() {
  ...
  loc loci: do {assert(q); ... }
  ...
}
main thread Main() {
  loc loc0: do {...
              start t();}
           goto loc1;
  loc loc1:
    do {assert(p_t at loci) -> q);}
    return;
}
```


Safety and Liveness: Informally

Consider the mutual exclusion problem again:

```
proctype T1() {
  do
    ::  // non-critical region
        // entry protocol
        // critical region
        // exit protocol
  od;
```

```
proctype T2() {
  do
    ::  // non-critical region
        // entry protocol
        // critical region
        // exit protocol
  od;
```

Req1: *"Both processes are never in their critical region at the same time"*

Safety and Liveness: Informally (Cont'd)

A trivial solution?!

```
proctype T1() {
  do
    ::  // non-critical region
        (false); // entry protocol
        // critical region
        skip; // exit protocol
  od;
```

```
proctype T2() {
  do
    ::  // non-critical region
        (false); // entry protocol
        // critical region
        skip; // exit protocol
  od;
```

Req1: *"Both processes are never in their critical region at the same time"* √ Safety

Req2: *"After starting its entry protocol, a process will always eventually be allowed into its critical region"* X Liveness

Safety and Liveness: Informally (Cont'd)

Safety

Intuitively:

"something bad never happens"

Examples:

- *"x is always positive"*
- *"The system never deadlocks"*
- *"The elevator will always be between the first and third floor"*
- *"The system will terminate after 10 steps"*

Liveness

Intuitively:

"something good eventually happens"

Examples:

- *"x will eventually be positive"*
- *"The system will terminate"*
- *"After pressing the request button, the elevator will eventually appear"*

- Terms due to Leslie Lamport
- What about assertions and invariants?

Safety and Liveness: Formally

P is a *safety property* iff for every trace

$t=s_0 s_1 s_2 \ldots$

we have

$t \notin P \Rightarrow \exists i.\ \forall t'.\ s_0 s_1 s_2 \ldots s_i t' \notin P$

"Once the 'bad thing' has occurred, there's no recovering from it"

P is a *liveness property* iff for every trace

$t=s_0 s_1 s_2 \ldots$

we have

$\forall i.\ \exists t'.\ s_0 s_1 s_2 \ldots s_i t' \in P$

"It is always possible for the 'good thing' to happen"

"Every property can be expressed as the conjunction of a safety property and a liveness property" [Alpern & Schneider, 1985]

"Safety and liveness are fundamental notions"

Safety and Liveness: Formally (Cont'd)

Let P be safety property and $t=s_0 s_1 s_2 \ldots$ be a trace violating P, that is, $t \notin P$. Then,

$\exists i.\ \forall t'.\ s_0 s_1 s_2 \ldots s_i t' \notin P$

"Safety properties are finitely refutable, i.e., counter examples will be finite"

Let P be a liveness property and $t=s_0 s_1 s_2 \ldots$ be a trace violating P, that is, $t \notin P$. Then,

$\forall i.\ \exists t'.\ s_0 s_1 s_2 \ldots s_i t' \in P$

"Liveness properties are not finitely refutable, i.e., counter examples will be infinite"

Safety and Liveness: More Examples

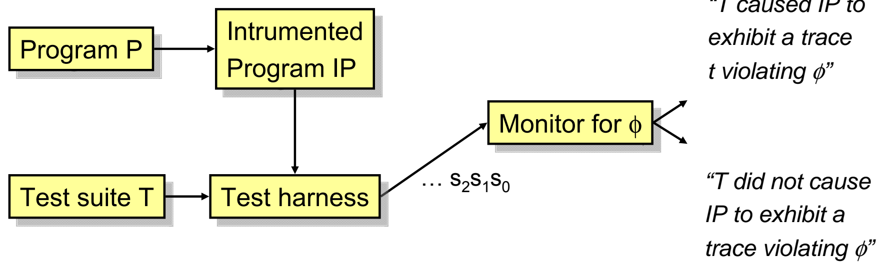
- *Req 1: "A use of a variable must be preceded by a declaration"*
- *Req 2: "When a file is opened, it must subsequently be closed"*
- *Req 3: "You cannot shift from 'drive' to 'reverse' without passing through 'neutral' "*
- *Req 4: "No pair of adjacent dining philosophers can be eating at the same time"*
- *Req 5: "Never two processes in their critical region at the same time"*
- *Req 6: "Every philosopher will always eat eventually"*

Which requirements are safety properties and which are liveness properties?

Safety, Liveness, and Run-time Monitoring

Theorem: Every property over finite traces is a safety property

⇒ Any property that a run-time monitor can check is a safety property

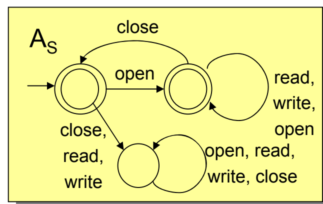


How to Express Safety and Liveness Properties?

- Safety
  - assertions and invariants
  - FSAs
  - Regular Expressions
  - Never Claims
- Liveness
  - progress labels
  - Buechi automata
  - Never Claims
- Linear Temporal Logic (LTL)

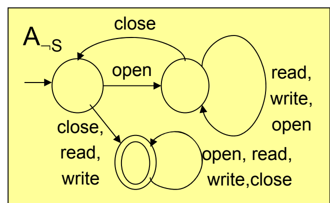
FSAs for Safety Properties

- S = *"Every file is opened before reading, writing, or closing"*



what we want to happen

- ¬S = *"A read, write, or close happened before an open"* (violation)



what we don't want to happen

Model checkers look for violations, so they typically expect violations of the safety property to be specified

Specifying Safety Properties in Bogor

```
extension Property for edu.ksu.cis.projects.bogor.ext.lite.property.Property
   { expdef boolean transformation(string, string); }

   function FSASpec() {
    loc init: live {}
        when Property.transformation("MAIN", "open") do { } goto opened;
        when Property.transformation("MAIN", "close") do { } goto bad$State;
    loc opened: live {}
        when Property.transformation("MAIN", "open") do { } goto bad$State;
        when Property.transformation("MAIN", "close") do { } goto init;
    loc bad$State: live {} // bad state
        do { } goto bad$State;
   }
```

What's the property specified here?

Observables, Again

- Alphabet Σ contains the "observables", i.e., the atomic propositions over which the specification is phrased
- Checker must be able to determine whether or not an observable is true in a given state
  - may be able to determine directly (e.g., variable values)
    - E.g., x>3 or **np_** in Spin
  - if not, helper variables and assignments must be used. E.g.:
    - **fileOpen := true**;
    - count number of processes in critical region
- In Bogor:
  - Observable: Values of variables and program counters
  - Not observable: e.g., ids of enabled threads

Regular Expressions (Recap)

- Let Σ be the alphabet
- Regular expressions over Σ are built as follows:
  - Every $a \in \Sigma$ is a regular expression over Σ
  - If e, e_1, e_2 are regular expressions over Σ, then
    - e_1 ; e_2 — concatenation/sequencing
    - e_1 | e_2 — choice/disjunction
    - e^* — reflexive and transitive closure/iteration with 0
    - (e) — grouping
    - e? — option (derived)
    - e^+ — transitive closure/iteration without 0 (derived)
    - . — any symbol/don't care (derived)
    - [e_1, e_2, ...] — union/multiple disjunction (derived)
    - [- e_1, e_2, ...] — complement/exclusion (derived)

    also are regular expressions over Σ
- Every regular expression e over Σ defines a set of words over Σ, that is, $L(e) \subseteq \Sigma^*$

Regular Expressions

Theorem:

- For every FSA A, there exists a regular expression e_A, such that $L(A) = L(e_A)$.
- For every regular expression e, there exists an FSA A_e, such that $L(e) = L(A_e)$.

⇒ FSAs and regular expressions can be used interchangeably

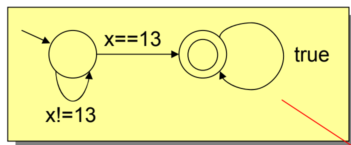
⇒ So, if FSAs can be used to express safety properties, then regular expressions can, too

Regular Expressions (Cont'd)

- Example:
  - Let open, close ∈ Σ
  - positive:
    - *"Every open is immediately followed by a close"*
    - ([- open]* (open close)?)*
  - negative:
    - *"At least one open is not immediately followed by a close"*
    - (.)* open ([- close] (.)*)?

Never Claims: FSAs in Spin

- Used in Spin to express violations of
  - safety properties, and
  - liveness properties (more on this later)
- For the moment, we can think of a Never Claim as
  - a Promela program that defines an FSA
  - expressing how a safety property can be violated, that is, the negation of a safety property



"x is never equal to 13"

acceptance label

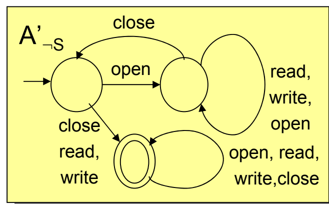
acceptance cycle

```
never {
        do
        :: (x!=13)
        :: (x==13) -> break
        od;
accept_s: do
          :: (true)
          od
}
```


Never Claims

More on this later!

- NC executed **synchronously** (remember?) with system
- Matching a never claim NC:
  - A run t matches NC, if t causes NC to
    - be fully executed, that is, the outermost "}" of the claim to be reached, or
    - reach an acceptance cycle
  - NC specifies behaviours that should never occur. So, if run t matches NC, we have found a violation!
- Not matching a never claim NC:
  - If the run t does not match NC, then t is ok, because it does not exhibit the violating behaviour described by NC
  - Note that if run t causes the NC to "get stuck" (i.e., NC has no enabled transition and t is not done), then t does not match NC



Never Claims (Cont'd)

"access operations are used in proper order"

There's an implicit acceptance cycle at the end of every never claim. So, the following two claims are equivalent:

```
never {
        do
        :: open -> do
                   :: close -> break
                   :: else -> skip
                   od
        :: else -> break
        od
accept: do
        :: true -> skip
        od
}
```

```
never {
        do
        :: open -> do
                   :: close -> break
                   :: else -> skip
                   od
        :: else -> break
        od
}
```


Never Claims (Cont'd)

- Can contain all of Promela's control-flow constructs
  - e.g., if, do, goto, etc
- Should be side-effect free (not change the state of the system being analyzed), that is, should only contain expression statements
- Can be non-deterministic

```
never {
   do
   :: (x!=13)
   :: (x==13) -> break
   od;
}
```

```
never {
   do
   :: (true)
   :: (x==13) -> break
   od;
}
```

```
never {
   do
   :: (x==13) -> break
   :: (true)
   od;
}
```

All of these can be used to check that *"x is never 13"!*
Why?

Expressing Liveness Properties

- Want to say that something good always eventually happens, i.e., that system always eventually makes some progress
- Violation: Execution along which eventually no more progress (towards the good thing) is made
- 3 possibilities to express liveness property in Spin:
  1. using progress labels (for simple liveness properties)
  2. using Buechi Automata (for simple and complex liveness properties)
  3. using Linear Temporal Logic (LTL) (for simple and complex liveness properties)
- Assumption: system has only infinite executions (possibly use "stutter extension")

1. Using Progress Labels

- Simple idea: label certain states as "good", i.e., as progress states
  - e.g., Philosopher1.eating
- Claim with respect to system S and progress state s:
  - From all reachable states in S, eventually s will always be reached

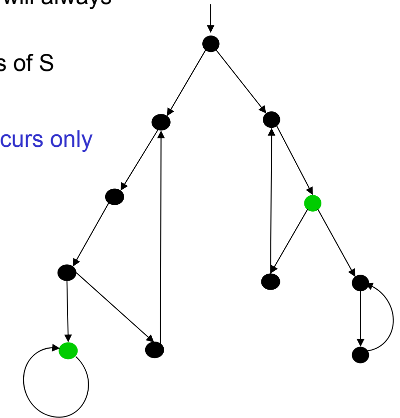
    ⇒ s occurs infinitely often along all executions of S
- Violation:
  - There is an execution in S along which s occurs only finitely many times

    ⇒ There is a run in S that's either

    1) finite, or

    2) ending in a cycle not containing s (non-progress cycle)

  If we add self-loops to all states with no outgoing arcs (stutter extension), 1) can be reduced to 2)



Stutter Extension

- **Goal:** When detecting non-progress cycles, don't want to have to distinguish between finite and infinite execution
- **Solution:**
  - Let M be an iFSA
  - We define "stutter extension" of M

    stutter(M) = (M.S, M.S_0, L', δ', M.F)

    where

    L' = M.L ∪ {"idle"}

    δ' = M.δ ∪ {(s, "idle", s) | s∈ M.S ∧ s has no outgoing transitions}
  - And then use stutter(M) to look for non-progress cycles

Non-Progress Cycles in Spin

```
mtype = { p, v };

chan sem = [0] of { mtype };

byte count;

active proctype semaphore()
{
    do
    :: sem!p ->
progress: sem?v
    od
}

active [5] proctype user()
{
    do
    :: sem?p ->
            count++;
            /* critical section */
            count--;
            sem!v
    od
}
```

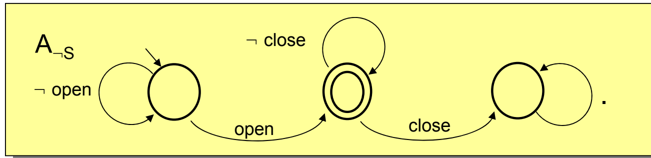

compile verifier with –DNP and run it with –l:

```
$ gcc -DNP -o pan pan.c
$ ./pan -l
```

or select "Non-Progress Cycles" in "Basic Verification Options" in xspin

2. Using Buechi Automata

- Progress labels alone are not enough to express more complicated liveness properties such as
  - S = *"The first open is always eventually followed by a close"*
- **Violation**:
  - ¬S = *"The first open is never followed by a close"*



- **Ok, but**:

  open read$^\omega$ $\notin$ L($A_{\neg S}$)

because only **finite** words can be accepted by FSA

→ need to change acceptance condition of FSA

ω–Acceptance

An *accepting ω-run* of an FSA A is an ω-run

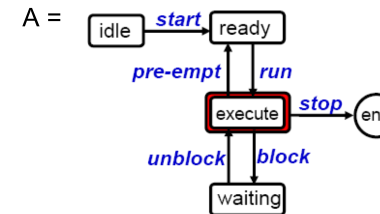
$\sigma = (s_0, l_0, s_1)(s_1, l_1, s_2)(s_2, l_2, s_3)\ldots(s_{i-1}, l_{i-1}, s_i)\ldots$

of A such that

$\exists\ s_f \in$ A.F. *"s_f occurs infinitely often in σ".*

Example:

A =



accepting ω–run of A:

idle (ready execute)$^\omega$

ω–word of A:

start run (pre-empt run)$^\omega$

ω–regular language $L^\omega(A)$ of A:

start run ((pre-empt run) + (block unblock))$^\omega$

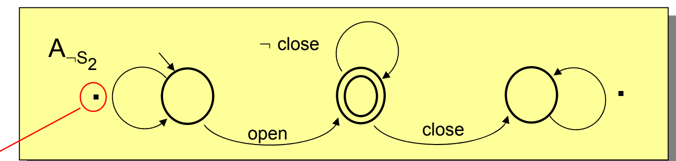
Buechi Automata

An Buechi automaton is a FSA with ω-acceptance.

- Buechi automata are sometimes also called ω-automata

Buechi Automata for Liveness: Example 2

- Let's try another liveness property
  - S_2 = *"Every open is always eventually followed by a close"*
- **Violation**:
  - $\neg S_2$ = *"At least one open is never followed by a close"*



Used to overread initial occurrences of 'open' that are followed by a close

For example:

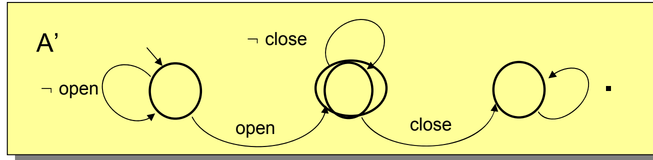
open [- close]$^\omega$ $\subseteq$ $L^\omega(A_{\neg S_2})$

open close open [- close]$^\omega$ $\subseteq$ $L^\omega(A_{\neg S_2})$

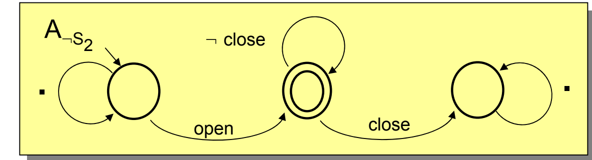
Note: other solutions are possible! Which?

Buechi Automata for Liveness: Example 2 (Cont'd)

- Let's try another liveness property
  - S_2 = *"Every open is always eventually followed by a close"*
- **Violation**:
  - $\neg S_2$ = *"At least one open is never followed by a close"*
- Why doesn't A' capture $\neg S_2$?



Buechi Automata as Never Claims



- Never claims are Buechi automata!
- Remember: run t is accepted/ matched by never claim NC, if t causes NC
  - to end in an acceptance cycle, or
  - to be fully executed (implicit acceptance cycle)
- Example:
  - S_2 = *"Every open is always eventually followed by a close"*
  - word $w \in L^\omega(A_{\neg S2})$ iff w matches $NC_{\neg S_2}$

$NC_{\neg S_2}$

```
never {
 init_s0:
   if
   :: (open) -> goto accept_s1
   :: (true) -> goto init_s0
   fi;
 accept_s1:
   if
   :: (!close) -> goto accept_s1
   fi;
}
```


Temporal Logic

Temporal logic =
propositional logic
\+
temporal operators

to capture properties of individual states, e.g.
- $x=0 \wedge (y=13 \rightarrow z=13)$
- m*n = output + (count*n)

- e.g., *"always"*, *"eventually"*, *"after"*, *"until"*
- to capture properties of sequences of states, e.g.,
  - $\neg$ (x=13) is always true
  - whenever "request", then eventually "granted"

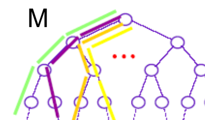
3. Using Linear Temporal Logic (LTL)

Linear Temporal logic (LTL) =
propositional logic
\+
temporal operators

to capture properties of individual states, e.g.
- $x=0 \wedge (y=13 \rightarrow z=13)$
- m*n = output + (count*n)

- G Φ *"always Φ"*
- F Φ *"eventually Φ"*
- X Φ *"Φ in next state"*
- Φ_1 U Φ_2 *"Φ_1 until Φ_2"*

Examples:
- G x != 13
- G (request $\rightarrow$ F granted)
- !access U locked



M ⊨ G x!=13 iff
"x!13 in every state in every run of M"

Linear Temporal Logic (LTL) (Cont'd)

- **Syntax**

  $\phi ::= \quad p \in AP \mid$ // atomic proposition (e.g., "x==0", "reqGranted")

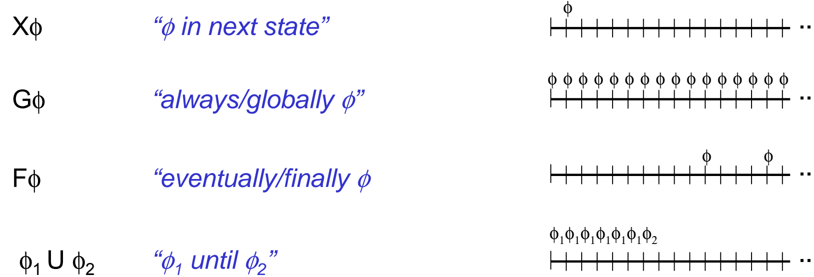
  $\neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \phi \leftrightarrow \phi \mid$

  $X\phi \mid G\phi \mid F\phi \mid \phi \; U \; \phi$

  where AP is set of "atomic propositions"

> Note Spin uses
> []ϕ instead of Gϕ
> <>ϕ instead of Fϕ

- **Semantic intuition**



$X\phi$	*"ϕ in next state"*	
$G\phi$	*"always/globally ϕ"*	
$F\phi$	*"eventually/finally ϕ"*	
$\phi_1 \; U \; \phi_2$	*"ϕ_1 until ϕ_2"*	

Linear Temporal Logic: Examples 1

- G x!=13
  - *"Along all runs, in all states, the value x is not equal to 13*
- $\neg$F(phil1eats $\wedge$ phil2eats)
  - *"Along all runs, it is not the case both philosophers eat at the same time"*
- G(pc1=loc1 $\rightarrow$ G pc1=loc1)
  - *"Along all runs of S, in every state, whenever pc1=loc1, then pc1=loc1 in all future states"*
- G(gear=reverse $\rightarrow$ X(gear=reverse $\vee$ gear=neutral))
  - *"Along all runs of S, in every state, whenever the gear is in reverse, then in the next state it will either still be in reverse or in neutral"*
- (F DB_updated) $\rightarrow$ ($\neg$ DB_read U DB_updated)
  - *"If the DB will be updated at some point in the future, then don't read it until then*

Formal Semantics of LTL

- We now want to define precisely when an LTL formula ϕ holds for some iFSA M
- **Problem:** LTL formulas are interpreted over infinite runs not finite ones
- **Solution:**
  - Let M be an iFSA
  - We define "stutter extension" of M

    stutter(M) = (M.S, M.S_0, L', δ', M.F)

    where

    L' = M.L $\cup$ {"idle"}

    δ' = M.δ $\cup$ {(s, "idle", s) | s has no outgoing transitions in M}
  - And interpret LTL formulas over L^ω(stutter(M))

> As before

Formal Semantics of LTL

Let M be iFSA, ϕ be LTL formula

$$M \models \phi \quad \text{iff} \quad \forall r \in L^\omega(\text{stutter}(M)).\ r \models \phi$$

> Note implicit universal quantification over all paths

where

- $r \models p$ iff eval(s_0,p) = true
- $r \models \phi_1 \wedge \phi_2$ iff $r \models \phi_1$ and $r \models \phi_2$

  ...
- $r \models X\phi$ iff $r_1 \models \phi$
- $r \models G\phi$ iff $\forall i \geq 0.\ r_i \models \phi$
- $r \models F\phi$ iff $\exists i \geq 0.\ r_i \models \phi$
- $r \models \phi_1 \; U \; \phi_2$ iff $\exists i \geq 0.\ r_i \models \phi_2$ and $\forall 0 \leq k < i.\ r_k \models \phi_1$

where

r is assumed to be of the form $s_0 s_1 s_2$... and

$r_i = s_i s_{i+1} s_{i+2}$ ... for all $i \geq 0$ and

eval: M.S$\times$AP $\rightarrow$ {true, false}

> For every state s and every atomic proposition p, we have a way of determining if p holds in s

LTL Equivalences

- all propositional equivalences, e.g., $\neg\neg\phi \leftrightarrow \phi$
- $\neg G\phi \quad \leftrightarrow \quad F\neg\phi$
- $\neg F\phi \quad \leftrightarrow \quad G\neg\phi$

} G and F are duals

- $G\phi \quad \leftrightarrow \quad \phi \wedge X\, G\, \phi$
- $F\phi \quad \leftrightarrow \quad \phi \vee X\, F\phi$
- $\phi_1\, U\, \phi_2 \quad \leftrightarrow \quad \phi_2 \vee (\phi_1 \wedge X(\phi_1\, U\, \phi_2))$

} use X to "unroll" G, F, and U

- $true\ U\ \phi \quad \leftrightarrow \quad F\phi$ — F is special case of U

Linear Temporal Logic: Examples 2

- *(X X open) → (X X X X close)*
  - *"Along all runs, if there's an 'open' in the second state, then there is a 'close' in the fourth state"*
- *G (open → F close)*
  - *"Along all runs, it must always be the case that an 'open' is eventually followed by a 'close'"*
- FG p
  - *"Along all runs, it must eventually be the case that p always holds"*
  - Example of a run satisfying/violating the specification?
- GF p
  - *"Along all runs, it must always be the case that eventually p holds", aka, "p holds infinitely often"*
  - Example of a run satisfying/violating the specification?
- G((rainStart ∧ ¬rainEnd ∧ F rainEnd) → (roofClosed U rainEnd))
  - *"Once the rain has started, the roof remains closed until the rain ends"*
  - *"The roof is always closed while it rains"*

LTL Notes

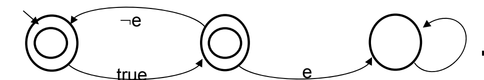
- Just like Buechi Automata, LTL can be used to express both
  - safety properties, e.g., G x!=13, and
  - liveness properties, e.g., F x=0
- Invented by Prior (1960's)
- First used to reason about concurrent systems by Amir Pnueli (Turing Award Winner)

LTL Notes (Cont'd)

- LTL's universal path quantification
  - An LTL formula is implicitly universally quantified over all paths of the system:

    $$M \models \phi \text{ iff } \forall r \in L^{\omega}(stutter(M)).\ r \models \phi$$

  - How do you express that *there exists a path satisfying a certain property ϕ*? Hint: Remember Assignment 1!
- Never Claims versus LTL
  - Never Claims (= Buechi Automata) are strictly more expressive:
    - Anything expressible in LTL can be expressed with a Never Claim
    - Not everything expressible with a Never Claim is expressible in LTL
      - Example: e may be received after an even # of transitions and e is never received after an odd # of transitions



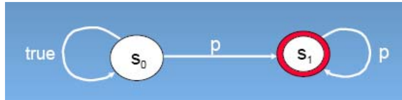
LTL and Buechi Automata

Theorem: For any LTL formula ϕ, there exists a Buechi automaton A_ϕ that accepts those runs for which ϕ is satisfied, i.e.,

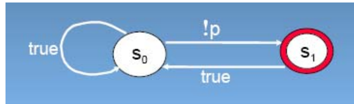
$$\forall\phi.\ \exists A_\phi.\ L(A_\phi) = \{r \mid r \models \phi\}$$

Examples: The LTL formula

1. G p corresponds to which Buechi automaton?
2. FG p corresponds to the Buechi automaton:



3. GF ¬p (¬p holds infinitely often) corresponds to Buechi automaton:



LTL and Spin

- Can use Spin to generate Buechi automaton (Never Claim) corresponding to a given LTL formula:

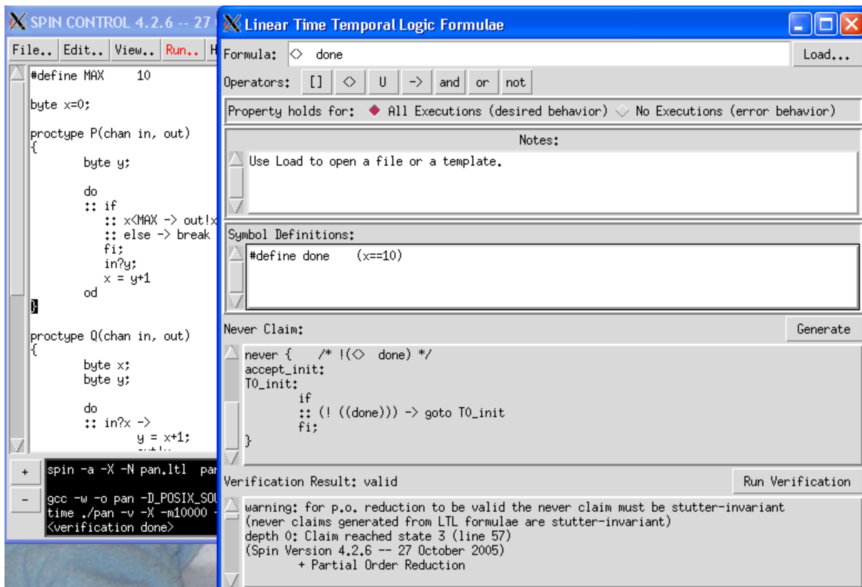
```
$ spin -f '<>[]p'
never {    /* <>[]p */
T0_init:
        if
        :: ((p)) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: ((p)) -> goto accept_S4
        fi;
}
```

true T0_init p S4 p

```
$ spin -f '!<>[]p'
never {    /* !<>[]p */
T0_init:
        if
        :: (! ((p))) -> goto accept_S9
        :: (1) -> goto T0_init
        fi;
accept_S9:
        if
        :: (1) -> goto T0_init
        fi;
}
```

true T0_init !p S9 true

LTL and Spin (Cont'd)



LTL and Spin (Cont'd)

- xspin has LTL property manager
  - edit, store, load LTL properties
  - view them as never claims
- Checking system S wrt LTL formula ϕ in Spin (*"Does S satisfy ϕ?"*):
  - Spin computes $NC_{\neg\phi}$, i.e., Never Claim of negation of ϕ
  - Spin explores state space of $S \otimes NC_{\neg\phi}$, i.e., the synchronous composition of S with $NC_{\neg\phi}$
  - If $S \otimes NC_{\neg\phi}$ has run ending in acceptance cycle, then
    - "Violation!"
    - S can exhibit behaviour violating ϕ and output counter example
  - If $S \otimes NC_{\neg\phi}$ has no run ending in acceptance cycle, then
    - "No violation!"
    - S cannot exhibit behaviour violating ϕ

Summary

- (Formal) Specifications
- Types of formal specifications
  - assertions
  - invariants
  - safety and liveness properties
- How to express safety properties
  - FSAs and regular expressions
  - Never Claims
  - LTL

how to check them using Bogor and Spin

- How to express liveness properties
  - progress labels
  - Buechi Automata and Never-Claims
  - LTL

how to check them using Spin

Summary (Cont'd)

- **Bogor**
  - checks for violations of safety properties expressed using assertions, invariants, or FSAs (expressed in BIR)
  - does not check for liveness properties
- **Spin**
  - checks for violations of liveness properties expressed using
    - progress labels: *"Is there a run not ending in a progress cycle?"*
    - Buechi Automata (expressed as Never Claims): *"Is there a run that fully matches the Never Claim?"*
    - LTL formulas (expressed as Never Claims): *"Is there a run that fully matches the Never Claim representing the negated LTL formula?"*

Computation Tree Logic (CTL) (1)

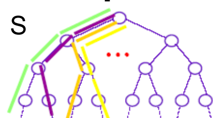
Computation Tree Logic (CTL) = propositional logic + temporal operators

to capture properties of individual states, e.g.
- $x=0 \wedge (y=13 \rightarrow z=13)$
- m*n = output + (count*n)

- AG p *"along all paths, in all states p"*
- AF p *"along all paths, eventually p"*

Examples:
- AG x != 13
- AF (request → EF granted)
- A[access U locked]

S



S

S ⊨ AG x!=13 iff

"x!=13 in every state in every path of S"

Also see course notes on CTL

Computation Tree Logic (CTL) (2)

- Temporal operators in CTL have the following form

PathQuantifier StateQuantifier

where
- PathQuantifier ∈ {All, Exists}
- StateQuantifier ∈ {Globally, Finally, neXt, Until}

CTL formulas are defined by the following BNF

$$\varphi ::= ff \mid tt \mid p \mid (\neg \varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid \mathbf{AX}\,\varphi \mid \mathbf{EX}\,\varphi \mid \mathbf{AG}\,\varphi \mid \mathbf{EG}\,\varphi \mid \mathbf{AF}\,\varphi \mid \mathbf{EF}\,\varphi \mid \mathbf{A}[\varphi_1\ \mathbf{U}\ \varphi_2] \mid \mathbf{E}[\varphi_1\ \mathbf{U}\ \varphi_2]$$

As before, p∈AP, that is, p is an atomic proposition

Computation Tree Logic (CTL) (3)

CTL formulas are defined by the following BNF

$$\varphi \ ::= \ ff \mid tt \mid p \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid$$
$$\mathbf{AX}\,\varphi \mid \mathbf{EX}\,\varphi \mid \mathbf{AG}\,\varphi \mid \mathbf{EG}\,\varphi \mid \mathbf{AF}\,\varphi \mid \mathbf{EF}\,\varphi \mid$$
$$\mathbf{A}[\varphi_1 \ \mathbf{U} \ \varphi_2] \mid \mathbf{E}[\varphi_1 \ \mathbf{U} \ \varphi_2]$$

path quantifier → $\mathbf{AX}\,\varphi$ "Along all paths, in the next state, φ holds"

$\mathbf{EX}\,\varphi$ "Along at least one path, in the next state, φ holds"

$\mathbf{AG}\,\varphi$ "Along all paths, in all future states, φ holds"
"Along all paths, φ holds globally"

$\mathbf{EG}\,\varphi$ "Along at least one path, in all future states, φ holds"
"Along at least one path, φ holds globally"

$\mathbf{AF}\,\varphi$ "Along all paths, in some future state, φ holds", or
"Along all paths, φ holds eventually"

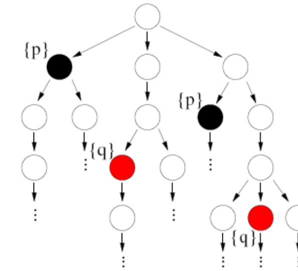
state quantifier → $\mathbf{EF}\,\varphi$ "Along at least one path, in some future state, φ holds", or
"Along at least one path, φ holds eventually"

$\mathbf{A}[\varphi_1 \ \mathbf{U} \ \varphi_2]$ "Along all paths, φ_1 holds at least until φ_2 does"

$\mathbf{E}[\varphi_1 \ \mathbf{U} \ \varphi_2]$ "Along at least one path, φ_1 holds at least until φ_2 does"

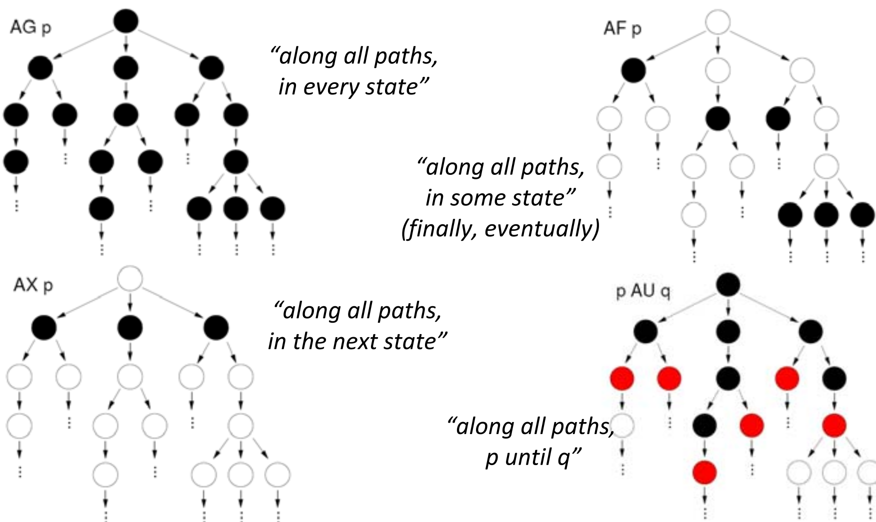
Computation Tree Logic (CTL) (4)

- Consider following computation tree

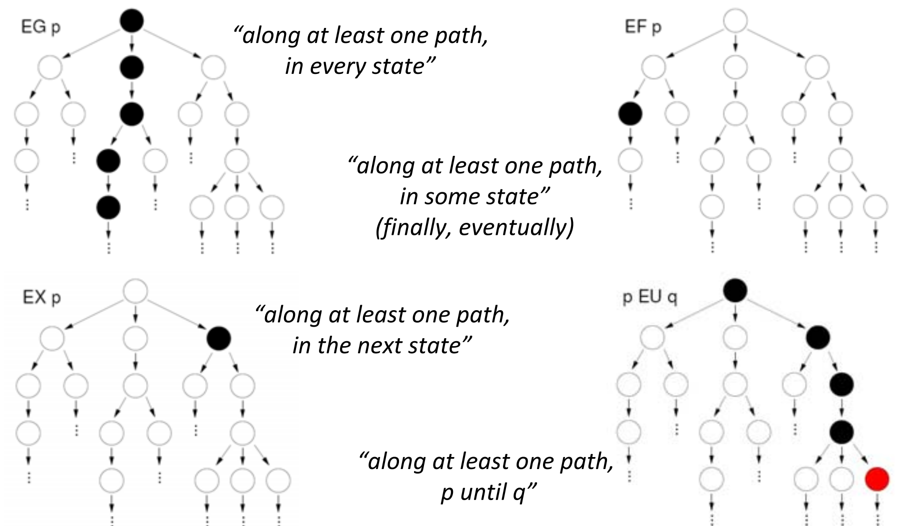


Black states satisfy p
Red states satisfy q

Computation Tree Logic (CTL) (5)



Computation Tree Logic (CTL) (6)



CTL Semantics

Formulas are interpreted over interpreted finite state machines. Given an iFSA $M = (S, S_0, L, \delta, F)$, a state s, and a CTL formula φ, the satisfaction relation $(M, s) \models \varphi$ is defined to be the smallest relation that satisfies:

$(M,s) \models tt$

$(M,s) \models p$ if $eval(p, s) = true$

$(M,s) \models \neg\varphi_1$ if not $(M,s) \models \varphi_1$

$(M,s) \models \varphi_1 \wedge \varphi_2$ if $(M,s) \models \varphi_1$ and $(M,s) \models \varphi_2$

$(M,s) \models \varphi_1 \vee \varphi_2$ if $(M,s) \models \varphi_1$ or $(M,s) \models \varphi_2$

$(M,s) \models \varphi_1 \rightarrow \varphi_2$ if not $(M,s) \models \varphi_1$ or $(M,s) \models \varphi_2$

$(M,s) \models$ **AX** φ if for all s' such that $(s, l, s') \in \delta$ for some $l \in L$, we have $(M, s') \models \varphi$

$(M,s) \models$ **EX** φ if for some s' such that $(s, l, s') \in \delta$ for some $l \in L$, we have $(M, s') \models \varphi$

$(M,s) \models$ **AG** φ if for all runs $s_1 s_2 s_3 \ldots$ in M such that $s = s_1$ we have $(M, s_i) \models \varphi$ for all $i \geq 1$

$(M,s) \models$ **EG** φ if for some runs $s_1 s_2 s_3 \ldots$ in M such that $s = s_1$ we have $(M, s_i) \models \varphi$ for all $i \geq 1$

Taken from course notes on CTL (available on course web pages)

CTL Semantics (Cont'd)

$(M,s) \models$ **AF** φ if for all runs $s_1 s_2 s_3 \ldots$ in M such that $s = s_1$ there exists $i \geq 1$ such that $(M, s_i) \models \varphi$

$(M,s) \models$ **EF** φ if for some run $s_1 s_2 s_3 \ldots$ in M such that $s = s_1$ there exists $i \geq 1$ such that $(M, s_i) \models \varphi$

$(M,s) \models$ **A**$[\varphi_1$ **U** $\varphi_2]$ if for all runs $s_1 s_2 s_3 \ldots$ in M such that $s = s_1$ there exists some $i \geq 1$ such that $(M, s_i) \models \varphi_2$, and for all $1 \leq j < i$, we have $(M, s_j) \models \varphi_1$

$(M,s) \models$ **E**$[\varphi_1$ **U** $\varphi_2]$ if for some run $s_1 s_2 s_3 \ldots$ in M such that $s = s_1$ there exists some $i \geq 1$ such that $(M, s_i) \models \varphi_2$, and for all $1 \leq j < i$, we have $(M, s_j) \models \varphi_1$

CTL Examples (1)

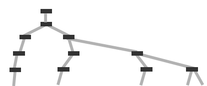
EF UserASuperUser — *"It is possible for User A to become superuser"*

AG EF UserASuperUser — *"It is always possible for User A to become superuser"*

AG (requested → AF granted) — *"A request is always eventually granted"*

AG ¬bad *¬EF bad* — *"It is impossible to reach a bad state"*

AG(floor=2 ∧ direction=up ∧ ButtonOnFloor5Pressed → A[direction=up U floor=5])



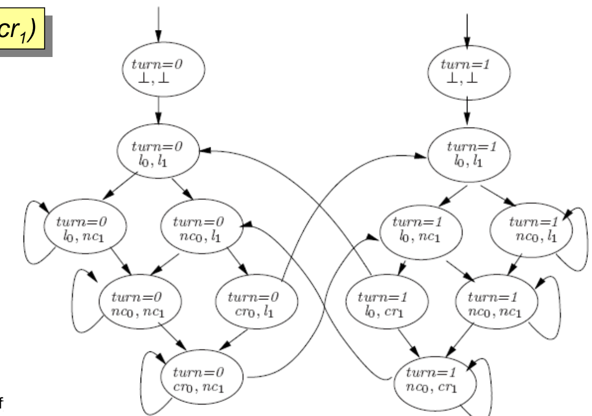
"In every reachable state, when the elevator is on floor 2 and moving up and the request Button on floor 5 is pressed, then
1. it will eventually arrive on floor 5, and
2. up until that point it will move up"

CTL Examples (2)

AG !(pc_0=cr_0 ∧ pc_0=cr_0) — *"C0 and C1 are never in their critical region at the same time"*

AG (pc_0=nc_0 → AF pc_0=cr_0) — *"C0 will always eventually be able to enter its critical region"*

AG (pc_1=nc_1 → AF pc_1=cr_1)



CTL Equivalences

Dualities

- $\neg AG\varphi \leftrightarrow EF\neg\varphi$
- $\neg EG\varphi \leftrightarrow AF\neg\varphi$
- $\neg AF\varphi \leftrightarrow EG\neg\varphi$
- $\neg EF\varphi \leftrightarrow AG\neg\varphi$
- $\neg AX\varphi \leftrightarrow EX\neg\varphi$
- $\neg EX\varphi \leftrightarrow AX\neg\varphi$

"Unwindings"

- $AG\varphi \leftrightarrow \varphi \wedge AX\, AG\varphi$
- $EG\varphi \leftrightarrow \varphi \wedge EX\, EG\varphi$
- $AF\varphi \leftrightarrow \varphi \vee AX\, AF\varphi$
- $EF\varphi \leftrightarrow \varphi \vee EX\, EF\varphi$
- $A[\varphi_1\ U\ \varphi_2] \leftrightarrow \varphi_2 \vee (\varphi_1 \wedge AX\, A[\varphi_1\ U\ \varphi_2])$
- $E[\varphi_1\ U\ \varphi_2] \leftrightarrow \varphi_2 \vee (\varphi_1 \wedge EX\, E[\varphi_1\ U\ \varphi_2])$

CTL Equivalences (Cont'd)

Reductions

- $EG\varphi \leftrightarrow \neg$ (AF$\neg$ varphi)
- $AF\varphi \leftrightarrow A[tt\ U\ \varphi]$
- $EF\varphi \leftrightarrow E[tt\ U\ \varphi]$
- $A[\varphi_1\ U\ \varphi_2] \leftrightarrow \neg(E[\neg\varphi_2\ U\ (\neg\varphi_1 \wedge \neg\varphi_2)] \vee EG\neg\varphi_2)$

CTL vs LTL

- LTL: execution sequences are linear
- CTL: execution sequence are branching
- The two logics are incomparable wrt their expressiveness
  - There are CTL formulas that are not expressible in LTL
    - e.g., AF AG p
  - There are LTL formulas that are not expressible in CTL
    - e.g., F G p
- Also, as we will see the algorithm for CTL model checking will be quite different from the LTL model checking algorithm that we have seen (more on this later)

Specification Patterns: Motivation

- Have already seen one way to classify properties:
  - safety
  - liveness
- This classification is very useful, because it impacts
  - design of analysis algorithms
  - design of optimization algorithms
  - use of existing tools (e.g., Spin)
- However, it's also very coarse

Specification Patterns: Motivation (Cont'd)

- Lots of interesting properties can be expressed in temporal logic
- However, as soon as temporal operators are nested, formulas can become very difficult to design and read
- Example:

$$[]((Q \land \neg R \land <>R) \rightarrow (P \rightarrow (\neg R \; U \; (S \land \neg R))) \; U \; R)$$

or, in ASCII format:

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

"P triggers S between Q and R"

- Would be great to have a more high-level way to think of temporal properties

Specification Pattern System

- Developed by Dwyer, Avrunin, and Corbett
- Pattern system for presenting, codifying, and reusing property specifications for finite-state verification
  - similar to Design Patterns in OO Programming
- Developed by examining over 500 temporal specifications collected from the literature
- Organized into a hierarchy based on the semantics of the requirement
- **http://patterns.projects.cis.ksu.edu**

Specification Patterns: Example

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:

Scope	Formula
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)`

5 different scopes

Specification Patterns: Example (Cont'd)

The Response Pattern (continued)

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

CTL:

Scope	Formula
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])`

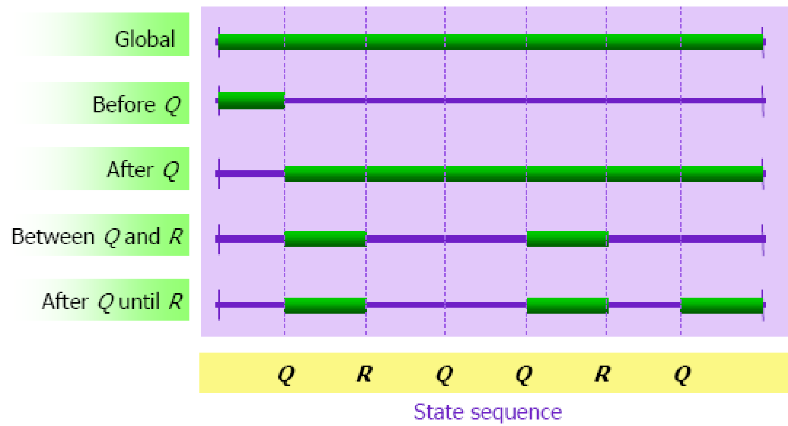
Examples and Known Uses:

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.

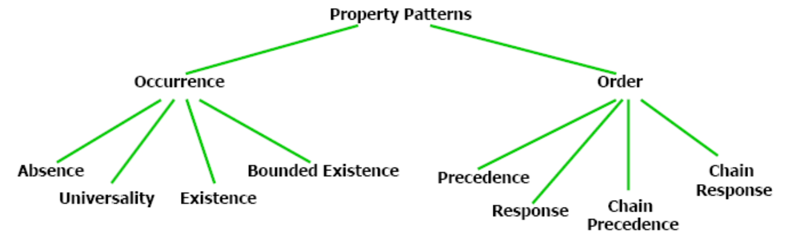
Relationships

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

Specification Patterns: Possible Scopes



Specification Patterns: Hierarchy



Occurrence:

Requires states or events to occur or not to occur

Order:

Constrains the order in which states or events occur

Specification Patterns: Hierarchy (Cont'd)

- Absence
  - A state/event does not occur within a given scope
- Existence
  - A state/event must occur within a given scope
- Bounded existence
  - A state/event must occur {at most, exactly, at least} k times within a given scope
- Universality
  - A state/event does occur throughout a given scope
- Precedence
  - A state/event P must always be preceded by a state/event Q
- Response
  - A state/event P must always be followed by a state/event Q

Specification Patterns: Examples in LTL

Pattern Mappings

Absence

P is false :

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

Scopes

Existence

P becomes true :

<table>
<tr><td>Globally</td><td><>(P)</td></tr>
<tr><td>(*) Before R</td><td>!R W (P & !R)</td></tr>
<tr><td>After Q</td><td>[](!Q) | <>(Q & <>P))</td></tr>
<tr><td>(*) Between Q and R</td><td>[](Q & !R -> (!R W (P & !R)))</td></tr>
<tr><td>(*) After Q until R</td><td>[](Q & !R -> (!R U (P & !R)))</td></tr>
</table>

`http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`

Specification Patterns: Examples in LTL (Cont'd)

Response

S responds to P :

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)

http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml