Logic Model Checking

Lecture Notes 12:18

Caltech 101b.2

January-March 2005

Course Text:

The Spin Model Checker: Primer and Reference Manual

Addison-Wesley 2003, ISBN 0-321-22862-6, 608 pgs.

Spin's LTL syntax

- ltl formula ::=

true, false	
any lower-case propositional symbol, e.g.: p, q, r, …	
(f)	round braces for grouping
unary f	unary operators
f_1 binary f_2	binary operators

unary ::=

`[]`	--- always, henceforth
`<>`	--- eventually
X	--- next
!	--- logical *negation*

caution

binary ::=

U	--- strong until
&&	--- logical *and*
||	--- logical *or*
->	--- logical *implication*
<->	--- logical *equivalence*

```
(p  -> q) is shorthand for: (!p || q)
(p <-> q) is shorthand for: (p -> q) && (q -> p)
```

semantics

given a state sequence (from a run σ):

s_0, s_1, s_2, s_3 …

and a set of propositional symbols: **p,q,…** such that

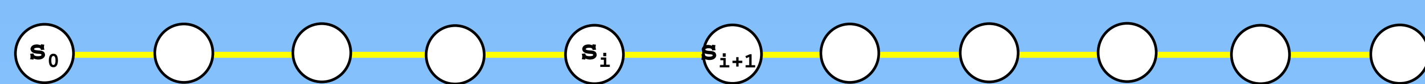
$\forall$**i,(i** $\geq$ **0) and** $\forall$**p,** $s_i \models p$ is defined

we can define the semantics of the temporal logic formulae:

[]f, <>f, Xf, and e U f

i.e., the property holds for the remainder of run σ, starting at position s_0

$\sigma \models f$	iff	$s_0 \models f$
$s_i \models$ []f	iff	$\forall$j,(j >= i): $s_j \models f$
$s_i \models$ <>f	iff	$\exists$j,(j >= i): $s_j \models f$
$s_i \models$ Xf	iff	$s_{i+1} \models f$



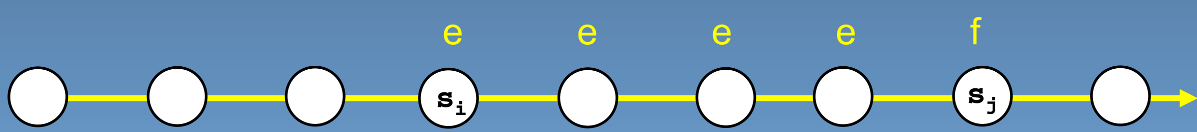
weak and strong until

(cf. book p. 135-136)

weak until

$s_i \models e \; U \; f$ iff

$s_i \models f \lor (s_i \models e \land s_{i+1} \models (e \; U \; f))$



strong until (Spin)

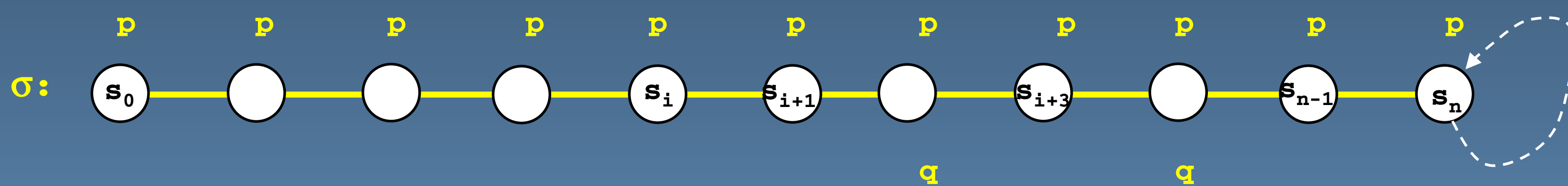
$s_i \models e \; U \; f$ iff

$\exists j, (j >= i): s_j \models f$ and

$\forall k, (i <= k < j): s_k \models e$

```
equivalences:
        (e U f) == (e U f) ∧ (<> f)
        (e U f) == (e U f) ∨ ([] e)
```

examples



[]p is satified at all locations in σ

<>p is satisfied at all locations in σ

[]<>p is satisfied at all locations in σ

<>q is satisfied at all locations except s_{n-1} and s_n

Xq is satisfied at s_{i+1} and at s_{i+3}

pUq (strong until) is satisfied at all locations except s_{n-1} and s_n

<>(pUq) (strong until) is satisfied at all locations except s_{n-1} and s_n

<>(pUq) (weak until) is satisfied at all locations

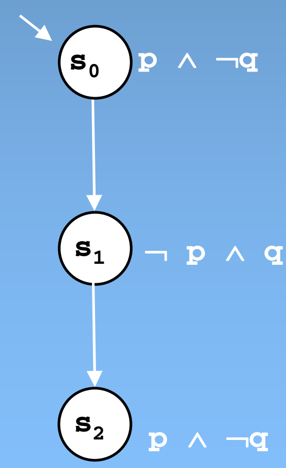
[]<>(pUq) (weak until) is satisfied at all locations

in model checking we are typically only interested in whether a temporal logic formula is satisfied for all runs of the system, starting in the initial system state (that is: at s_0)

equivalences

(cf. book p. 137)

- [] p ↔ (p U false) — weak until
- <>p ↔ (true U p) — strong until
- ![]p ↔ <>!p
  - if p is not invariantly true, then eventually p becomes false
- !<>p ↔ []!p
  - if p does not eventually become true, it is invariantly false
- []p && []q ↔ [] (p && q)
  - note though: ([] p || [] q) → [] (p || q)
  - but: ([]p || [] q) ~~←~~ [] (p || q)
- <>p || <> q ↔ <> (p || q)
  - note though: (<> p && <> q) ← <> (p && q)
  - but: (<> p && <> q) ~~→~~ <> (p && q)



some standard LTL formulae

[] p	always p	invariance
<> p	eventually p	guarantee
p -> (<> q)	p implies eventually q	response
p -> (q U r)	p implies q until r	precedence
[]<> p	always, eventually p	recurrence (progress)
<>[] p	eventually, always p	stability (non-progress)
(<> p) -> (<> q)	eventually p implies eventually q	correlation

non-progress
acceptance

dual types of properties

in every run where p eventually becomes true q also eventually becomes true (though not necessarily in that order)

the earlier informally stated sample properties

(vugraph 12 lecture 11)

- p is invariantly true

  [] p

- p *eventually* becomes invariantly true

  <>[] p

- p *always eventually* becomes false at least once more

  []<>!p

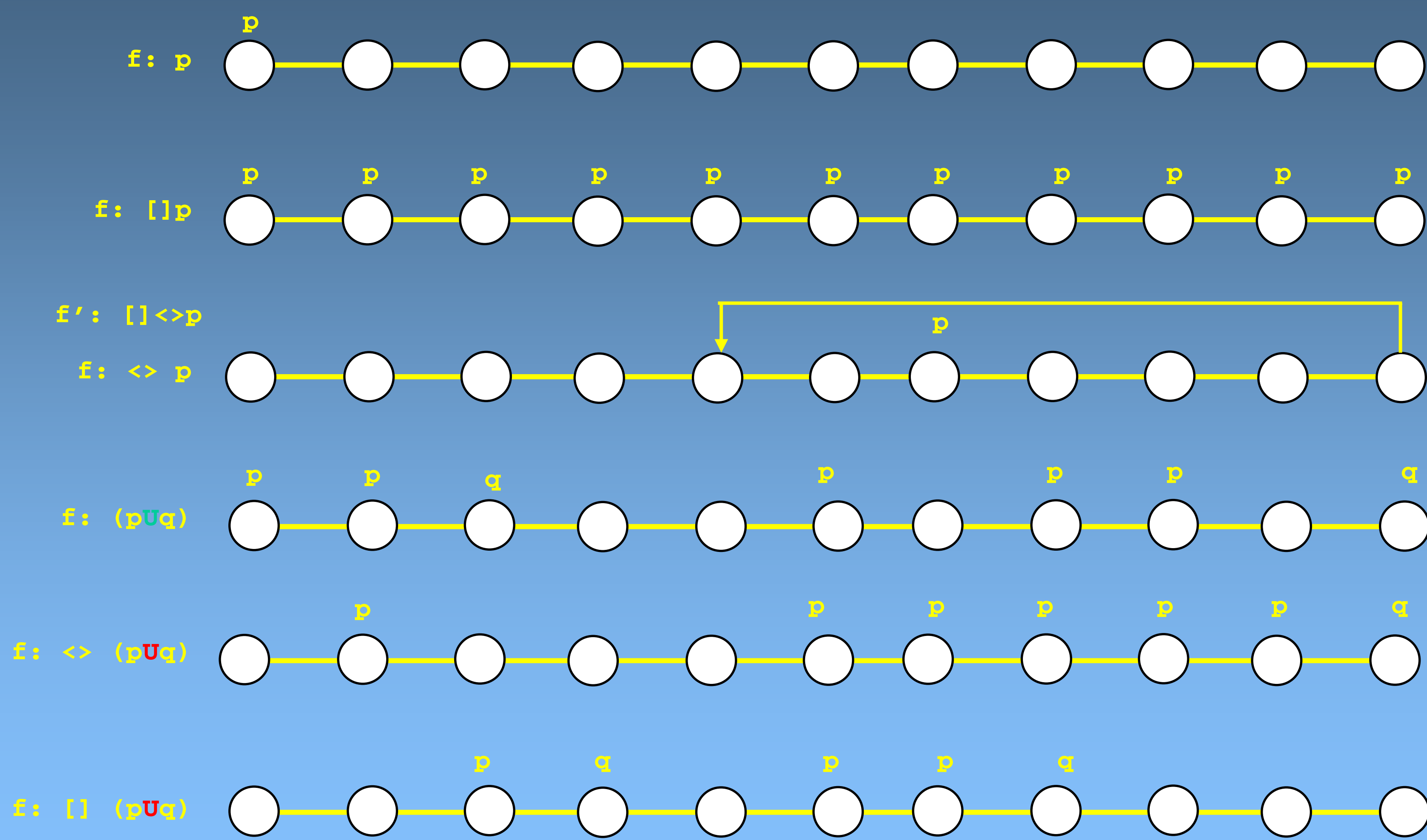
- p *always* implies ¬q

  [] (p -> !q)

- p *always* implies *eventually* q

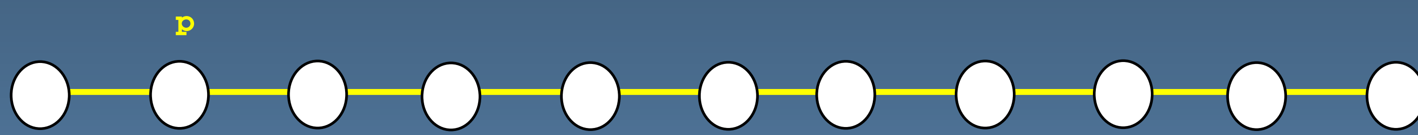
  [] (p -> <> q)

visualizing LTL formulae



the simplest operator: X

f: X(p)



- the next operator X is part of LTL, but should be viewed with some suspicion
  - it makes a statement about what should be true in all possible *immediately* following states of a run
  - in distributed systems, this notion of 'next' is ambiguous
  - since it is unknown how statements are interleaved in time, it is unwise to build a proof that depends on specific scheduling decisions
    - the 'next' action could come from any one of a set of active processes – and could depend on relative speeds of execution
  - the only *safe* assumptions one can make in building correctness arguments about executions in distributed systems are those based on longer-term *fairness*

stutter invariant properties

(cf. book p. 139)

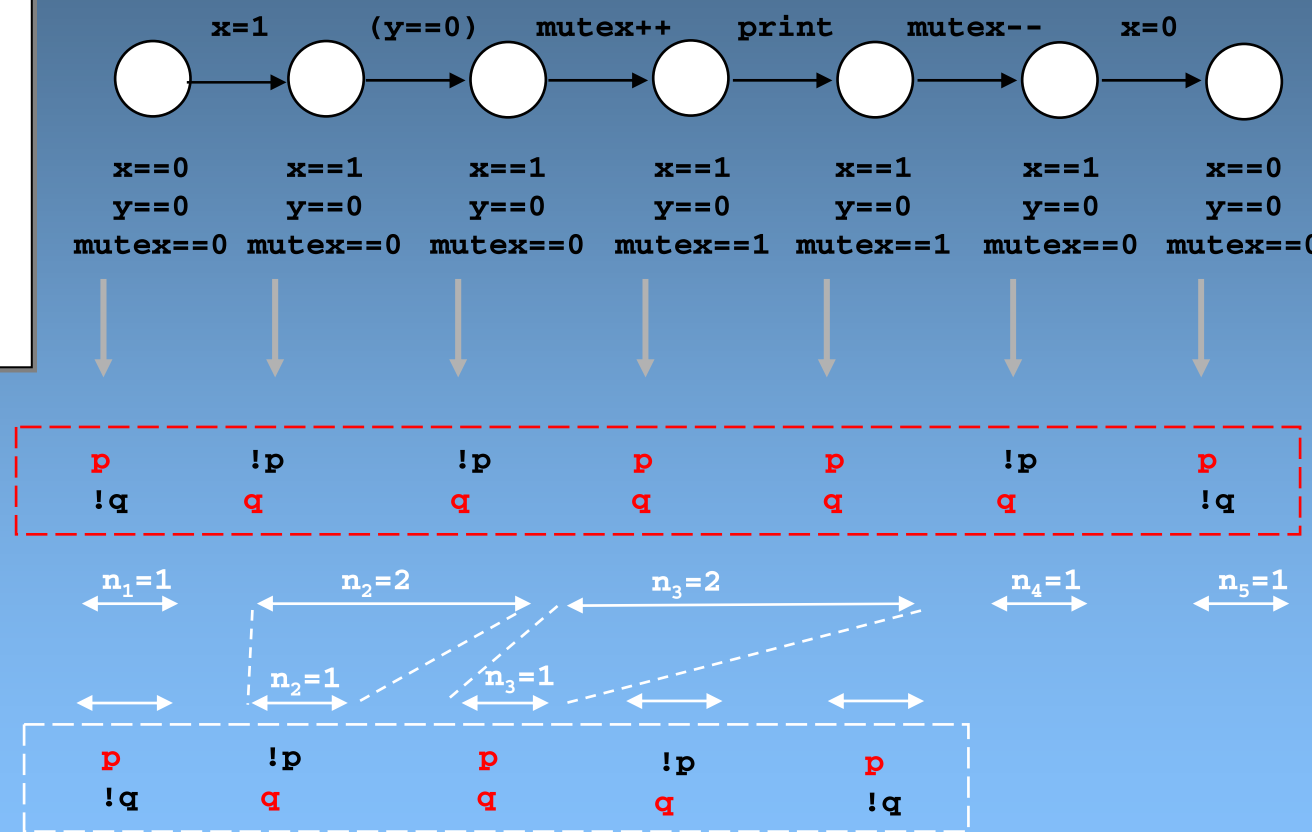
- Let ϕ = V(σ,P) be a *valuation* of a run σ for a given set of propositional formulae P
  - a series of truth assignment to all propositional formulae in P, for each subsequent state that appears in σ
  - the truth of any temporal logic formula in P can be determined for a run when the valuation is given
  - we can write ϕ as a series of intervals: $\phi_1^{n1}, \phi_2^{n2}, \phi_3^{n3}, \ldots$ where the valuations are identical within each interval of length n1, n2, n3, ...

- Let E(ϕ) be the set of all valuations (for different runs) that differ from ϕ only in the values of n1, n2, n3, … (i.e., in the length of the intervals)
  - E(ϕ) is called the *stutter extension* of ϕ

valuations

```
p: (x == mutex)
q: (x != y)
```

```
bit  x, y;
byte mutex;
active proctype A() {
      x = 1;
     (y == 0) ->
      mutex++;
      printf("%d\n", _pid);
      mutex--;
      x = 0
}
```



a run σ and its valuation ϕ:

another run in the same set E(ϕ)

stutter invariant properties

(cf. book p. 139)

- a *stutter invariant* property is either true for all members of $E(\phi)$ or for none of them:
  - $\sigma \models f \wedge \phi = V(\sigma,P) \rightarrow \forall \nu \in E(\phi), \nu \models f$
- the truth of a stutter invariant property does not depend on *'how long'* (for how many steps) a valuation lasts, just on the *order* in which propositional formulae change value
- we can take advantage of stutter-invariance in the model checking algorithms to *optimize* them (using partial order reduction theory)...
- theorem: X-free temporal logic formulae are stutter invariant
  - temporal logic formula that do contain X can also be stutter-invariant, but this isn't guaranteed and can be hard to show
  - the morale: avoid the *next* operator in correctness arguments

```
example: [](p -> X (<>q))
is a stutter-invariant LTL formula
that contains a X operator
```

interpreting formulae...

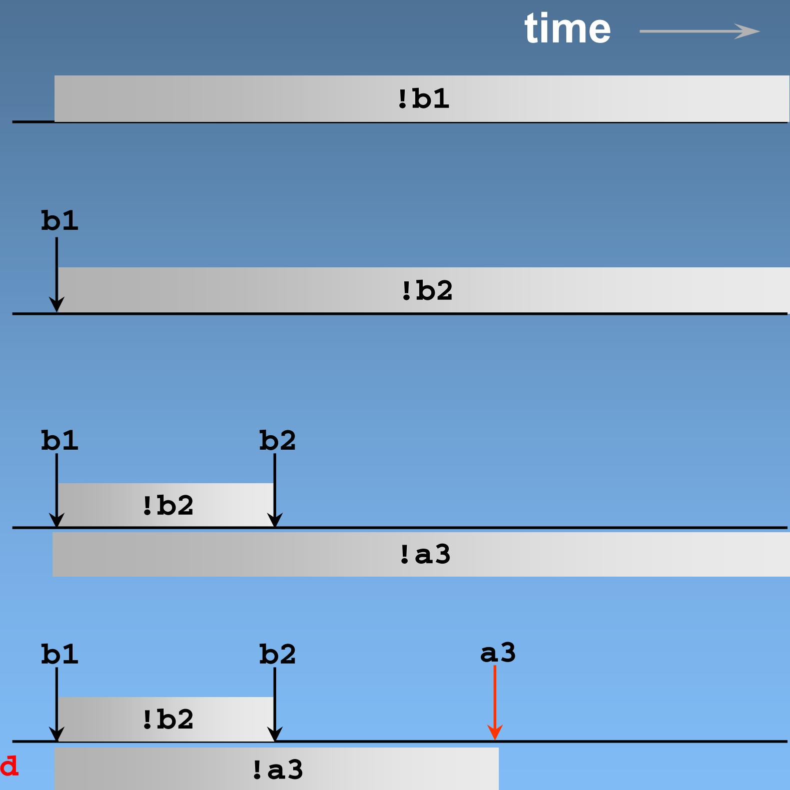
LTL: (<>(b1 && (!b2 U b2))) -> []!a3

1. suppose b1 never becomes true
   `(p->q) means (!p ∨ q)`
   `the formula is` ***satisfied!***

2. b1 becomes true, `but not b2`
   `the formula is` ***satisfied!***

3. b1 becomes true, then b2
   `but not a3`
   `the formula is` ***satisfied***

4. b1 becomes true, then b2, then a3
   `the formula is` ***not satisfied***
   `i.e., the property is violated`

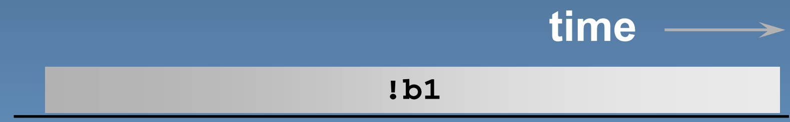


another example

`LTL: (<>b1) -> (<>b2)`

1. b1 never becomes true

   `formula satisfied`

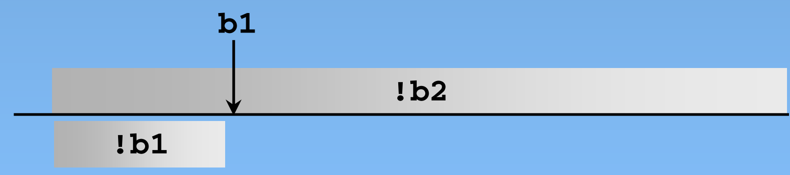


2. b1 and b2 both become true

   `formula satisfied`

3. b1 becomes true but not b2

   formula *not* satisfied
   the property is *violated*



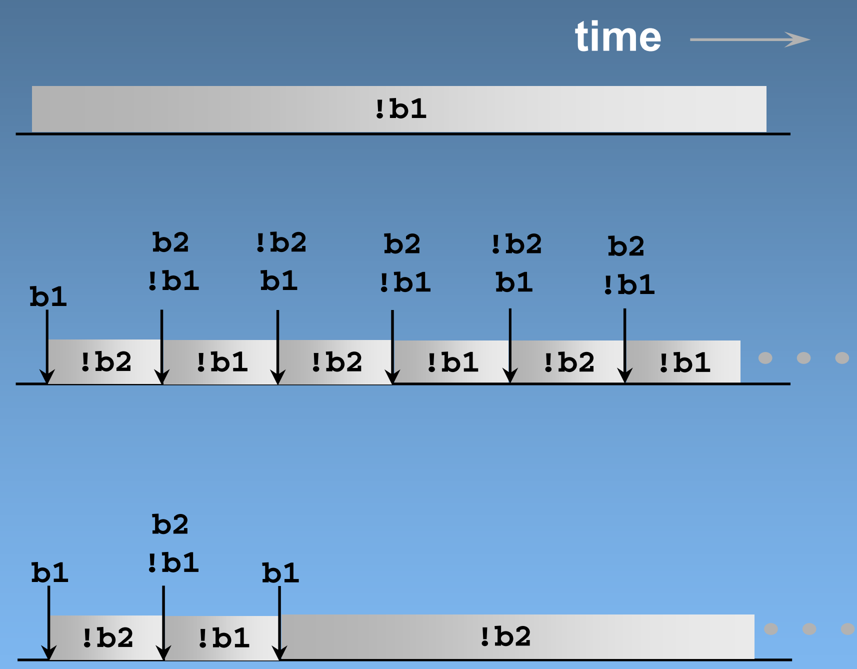
prefix the last formula with "[]"

LTL: []((<>b1) -> (<>b2))

1. b1 never becomes true
   `formula satisfied`

2. b1 and b2 alternate, indefinitely
   `formula satisfied`

3. b2 becomes true only once
   `the formula is *not* satisfied`
   `property is violated`



where intuition can fail...

e.g., expressing the property: "p implies q"

- p -> q
  - not that there are no temporal operators (`[]`, <>, U) in this formula -- it is a propositional formula (a *state* property) that will apply *only* to the *initial state* of each run...
  - the formula is immediately satisfied if (!p || q) is *true* in the initial system state – and the rest of the run is irrelevant

- `[]`p -> q
  - beware of precedence rules...
  - as written this is parsed as (`[]`p) -> (q)
  - if p is not invariantly true, the formula is vacuously satisfied *(by the definition of ->,* "->" is *not* a temporal operator!*)*
  - if p is invariant, then the formula is satisfied *if q holds in the initial system state...*

expressing properties in LTL

“p implies q”

- [](p -> q)
  - note: there is still no temporal relation between p and q
  - this formula is satisfied if in every reachable state the propositional formula (!p || q) holds
- [](p -> <> q)
  - this would still be satisfied if p and q become true simultaneously, in one step (repeatedly)
  - doesn’t capture the notion that somehow the truth of p *causes*, sometime later, the truth of q

expressing properties in LTL

"p implies q"

- [](p -> X(<>q))
  - puts one or more steps in between the truth of p and q, but this uses the maligned X operator... (but stutter invariance is maintained in this case)
  - formula is still satisfied if p *never becomes true*, probably not what is meant

- [](p -> X(<>q)) && (<>p)
  - this may actually capture what we intended
  - compare to our first guess of just: (p -> q)

beware of LTL
always double-check your formulae
be especially on guard when a model checker *fails* to find a matching run...

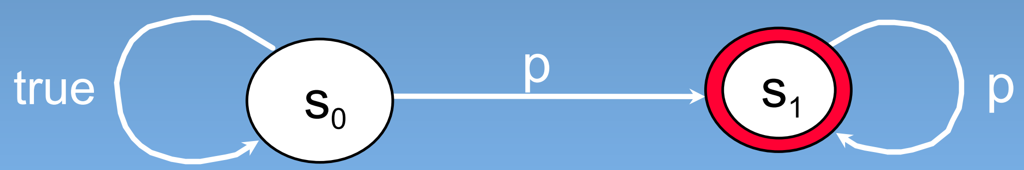
always use Spin to generate the never claim for each LTL formula, and study it to see if it matches your intuition of what you thought it should be...

from logic to automata

(cf. book p. 141)

- for any LTL formula f there exists a Büchi automaton that *accepts* precisely those runs for which the formula f is satisfied

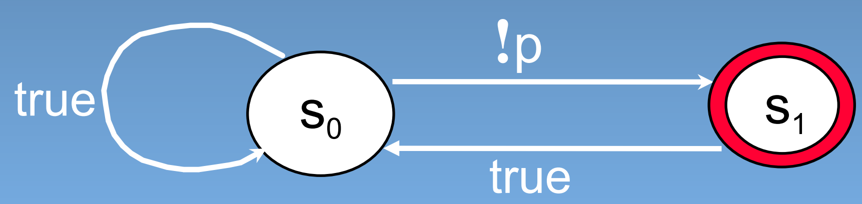
- example: the formula **`<>[]p`** corresponds to the non-deterministic Büchi automaton:



from logic to automata

- to turn an LTL correctness *requirement* into a Promela *never claim*, just negate the LTL formula, and generate the claim from the negated form:

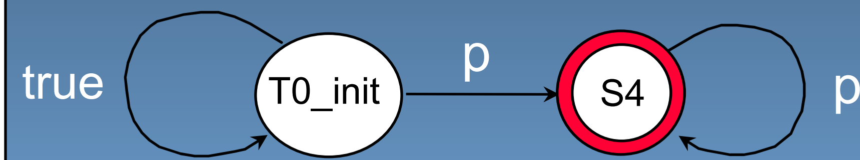
!<>[]p ≡ []![]p ≡ []<>!p



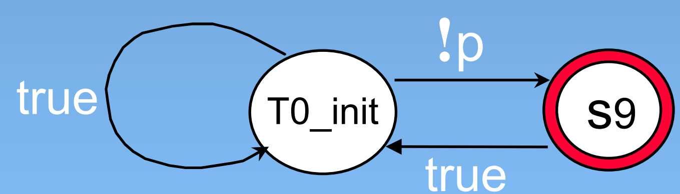
the automaton only accepts a run if p keeps returning to false infinitely often
i.e., securing that in the run considered p does not remain true invariantly, ever

using Spin to do the negations and the conversions

```
$ 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;
}
```




```
$ 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;
}
```



syntax rules

```
$ spin -f '([] p -> <> (a+b <= c))'
```

```
#define q          (a+b <= c)
```

define lower-case propositional symbols for all arithmetic and boolean subformulae

```
$ spin -f '[] (p -> <> q)'
never {    /* [](p -> <> q) */
T0_init:
        if
        :: (((! ((p))) || ((q)))) -> goto accept_S20
        :: (1) -> goto T0_S27
        fi;
accept_S20:
        if
        :: (((! ((p))) || ((q)))) -> goto T0_init
        :: (1) -> goto T0_S27
        fi;
accept_S27:
        if
        :: ((q)) -> goto T0_init
        :: (1) -> goto T0_S27
        fi;
T0_S27:
        if
        :: ((q)) -> goto accept_S20
        :: (1) -> goto T0_S27
        :: ((q)) -> goto accept_S27
        fi;
}
$
```

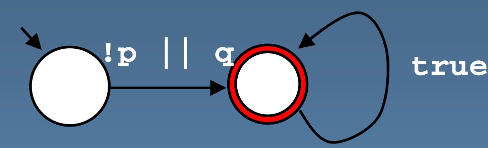
beware of operator precedence rules..

there is *no* minimization algorithm for non-deterministic Büchi automata. sometimes alternative converters can produce smaller automata:

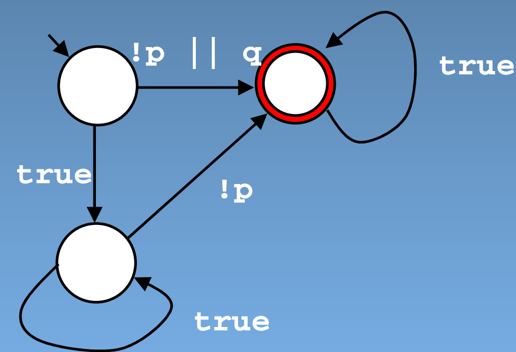
```
$ ltl2ba -f '[] (p -> <> q)'
never { /* [] (p -> <> q) */
accept_init:
        if
        :: (!p) || (q) -> goto accept_init
        :: (1) -> goto T0_S2
        fi;
T0_S2:
        if
        :: (1) -> goto T0_S2
        :: (q) -> goto accept_init
        fi;
}
```

gaining intuition for ltl formula

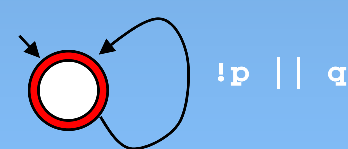
- p -> q



- [] p -> q

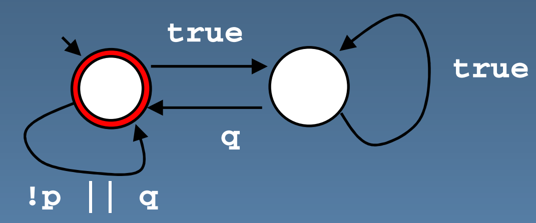


- [] (p -> q)

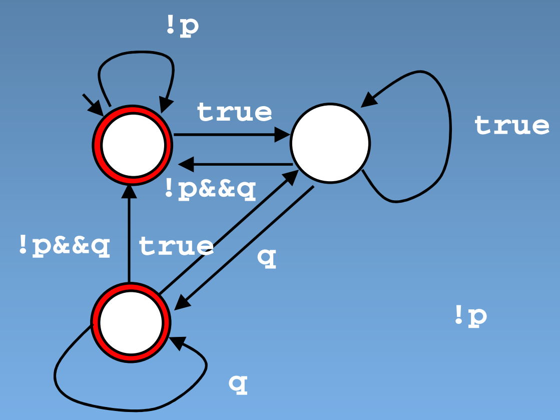


gaining intuition for ltl formula

- [] (p -> <> q)

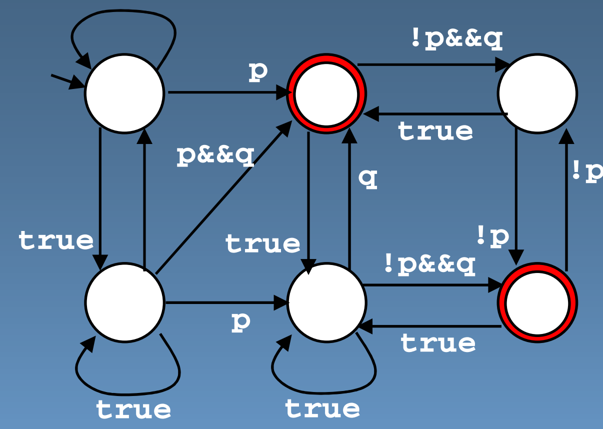


- [] (p -> X <> q)



the last few steps...

- [] (p -> X <> q) && (<> p) spin -f



but, what we really want for *verification* is the violation of this property: the negated formula...

be warned:
larger property automata are generally harder to understand and they incur more complexity during the verification process

- !([] (p -> X <> q) && (<> p)) spin -f