The SPIN Model Checker

Metodi di Verifica del Software

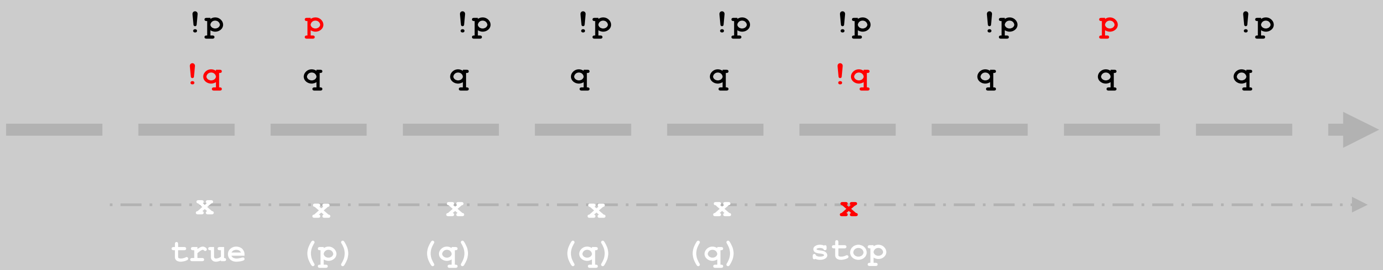
Andrea Corradini – GianLuigi Ferrari

Lezione 5

2011

Slides per gentile concessione di Gerard J. Holzmann

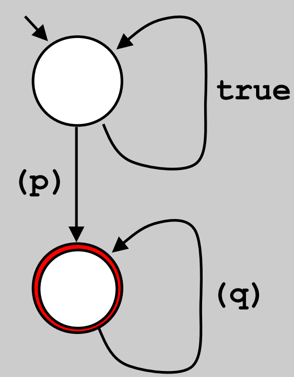
a never claim defines an *observer* process that executes *synchronously* with the system



property:
the truth of p is always followed within a finite number of steps by the truth of !q

never claim (negation of property):
the truth of p is *not* followed within a finite number of steps by the truth of !q

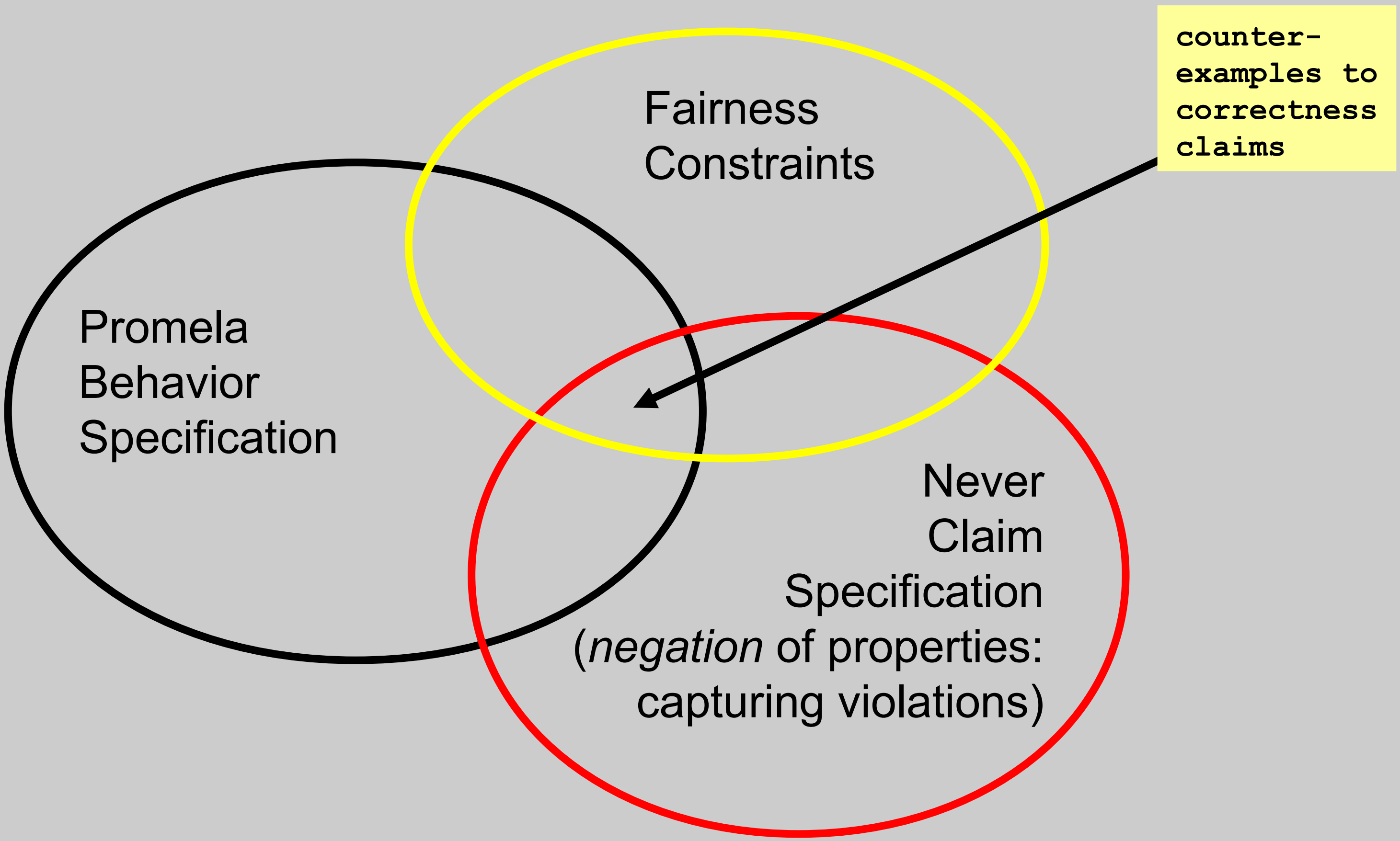
```
never {
    do
    :: true
    :: (p) -> break
    od;
accept:
    do
    :: (q)
    od
}
```



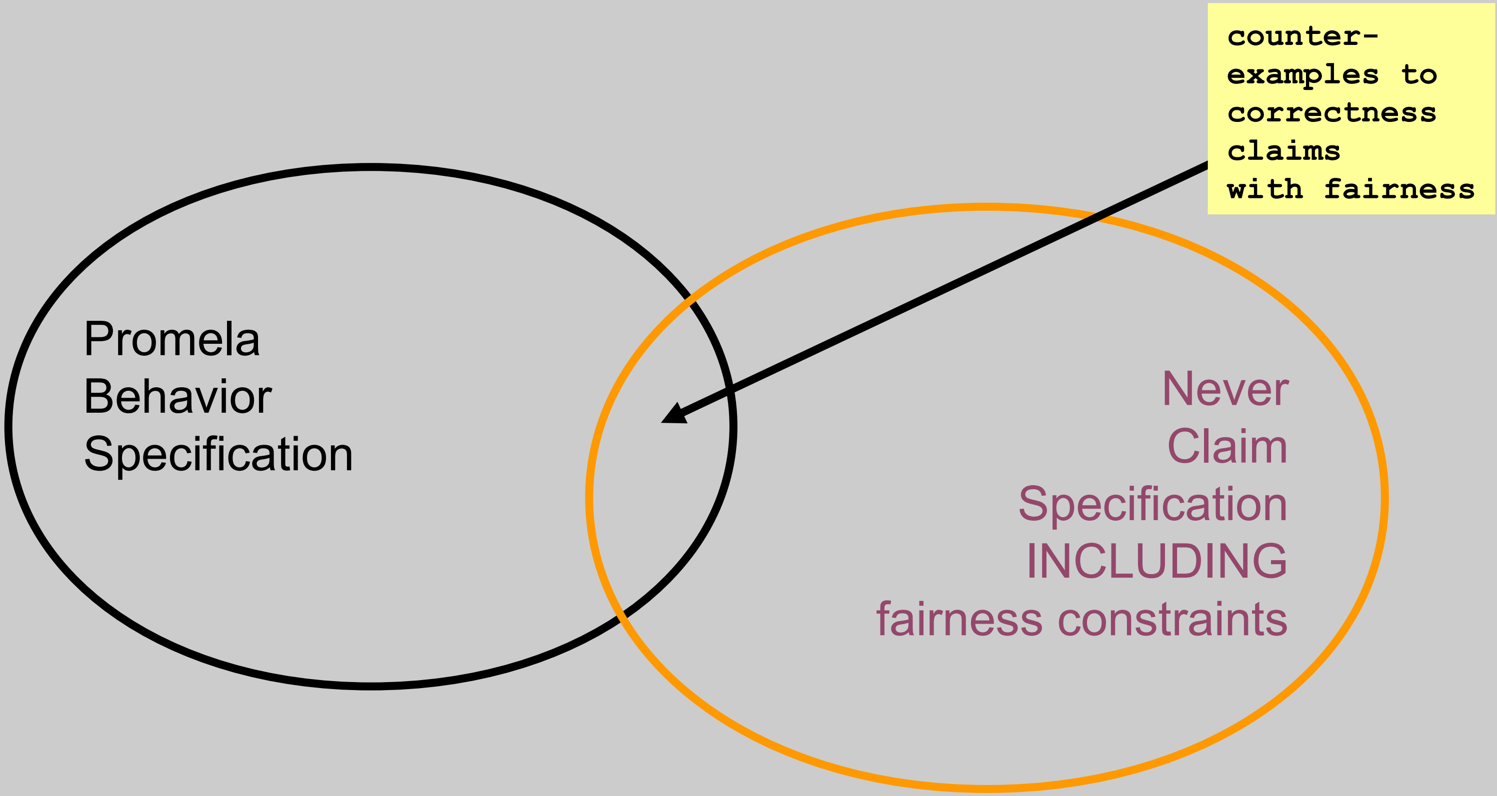
never claims

- can be either deterministic or non-deterministic
- should *only* contain side-effect free expression statements (corresponding to boolean propositions on system states)
- are used to define *invalid* execution sequences
  - a signature or pattern of *invalid* system behavior
- truncate (i.e. abort) when they block
  - a block means that the behavior expressed *cannot* be matched
  - the never claim process gives up trying to match the current execution sequence, backs up and tries to match another execution
  - pausing in the never claim must be represented explicitly with self-loops on *true*
- a never claim reports a violation when:
  - closing curly brace of never claim is reached
  - an acceptance cycle is closed
- non-progress can be expressed as a never claim, or as part of a never claim
  - a built-in option allows spin to generate a default never claim for checking non-progress properties, but this is optional

the language intersection picture

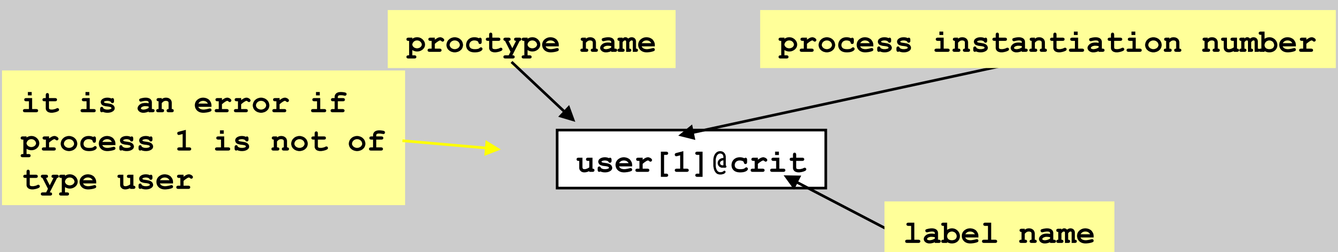


the language intersection picture



referencing process states from within never claims

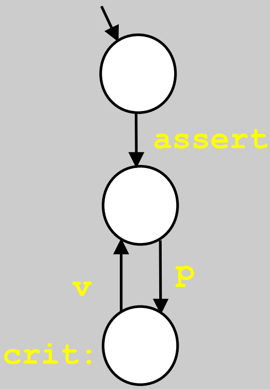
- from within a never claim we can refer to the control-flow states of any active process
- the syntax of a “remote reference” is:
  - proctypename[pidnr]@labelname
- this expression is true *if and only if* the process with process instantiation number *pidnr* is currently at the control-flow point marked with *labelname* in *proctypename*



- if there is only *one* process of type user, we can also omit the [pid] part and use a simpler form:

```
user@crit
```

referencing process states



- an example

proctype names

process instantiation numbers

```
never {
    do
    :: user[1]@crit && user[2]@crit -> break
    :: else
    od
    /* reaching the end of a never claim is alw
}
```

label names

a way to make sure we are using the right pid numbers in the claim

```
mtype = { p, v };
chan sem = [0] of { mtype };

active proctype semaphore()

    do :: sem!p ; sem!v od


active [2] proctype user()
{   assert(_pid == 1 || _pid == 2);
    do
    :: sem?p ->
crit:       /* critical section */
        sem?v
    od
}
```

using a state label, instead of a counter to check mutual exclusion

we do not need an accept label in the never claim in this case
Q1: why not?
Q2: what if we added one anyway?

remote referencing expressions can *only* be used in never claims... (they are meant to *monitor* behavior not to *define* behavior)

checking when a process has terminated

```
active proctype runner()
{
   do
   :: ... ...
   :: else -> break
   od
}
```

make it visible →

```
active proctype runner()
{
   do
   :: ... ...
   :: else -> break
   od;
L: (false)
}
```

the expression:

(runner@L)

will be true if and only if the process reaches label L
once the process reaches this label it can never proceed beyond it

another method:
we can also try to use the predefined global variable

_nr_pr

to count how many processes are running...

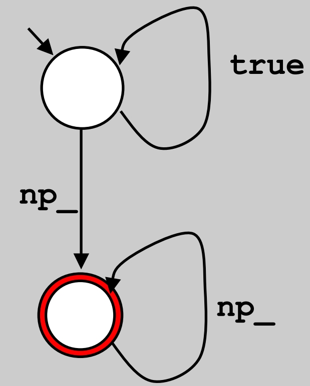
never claims

- can contain *all* control flow constructs
  - including if, do, unless, atomic, d_step, goto
- should contain *only* expression statements
  - so, q?[ack] or nfull(q) is okay, but not q?ack or q!ack
- the convention is to use accept-state labels *only* in never claims and progress and end-state labels *only* in the behavior model
- special precautions are needed if non-progress conditions are checked *in combination with* never claims
  - non-progress is normally encoded in Spin as a predefined never claim
  - you can use progress labels inside a never claim, but only if you also encode the non-progress cycle check within the claim....

the predefined non-progress cycle detector

- one of the predefined system variables in Promela (similar to ‘timeout’, ‘else’, and ‘_nr_pr’) is np_
- np_ (non-progress state) is defined to be *true* if and only if *none* of the active processes is currently at a state that was marked with a progress label
- the predefined non-progress cycle detector is the following two-state never claim, accepting only non-progress cycles (following any finite prefix)

```
never {
   do
   :: true
   :: np_ -> break
   od;
accept:
   do
   :: np_
   od
}
```




```
(non-)progress is a liveness property
captured with an accept state label inside
the never claim
non-progress cycles are therefore internally
captured as acceptance cycles
```

never claims can also be used to *restrict* a search for property violations to a smaller set of executions

- model checking is often an exercise in controlling computational complexity
- abstraction is the best (and morally right) way to address these problems, but not always easy
- suppose we have defined a model that is too detailed and therefore intractable / unverifiable
- we can select interesting behaviors from the system by using a never claim as a *filter*
- the model checker will not search executions where the expression statements in the claim cannot be matched...
- simple example:

```
never {
   do
   :: atomic { (p || q) -> assert(r)}
   od
}
```

restrict to behavior where either p or q remain true, and check assertion r at every step, but only in those executions

example of a constraint

```
never {
   do
   :: ( x + y < N )
   od
}
```

restrict the search to only those executions where x+y < N holds; place assertions or accept labels elsewhere

reminder:
if a never claim is present, and we compile with -DNP, the never claim is replaced with the predefined non-progress claim.

if we want to check a progress condition AND a constraint simultaneously, we have to define an explicit constrained NP automaton

```
never {
   do
   :: true
   :: np_ -> break
   od;
accept:
   do
   :: np_
   od
}
```

x

```
never {
   do
   ::(x+y < N)
   :: np_ && (x+y < N)-> break
   od;
accept:
   do
   :: np_ && (x+y < N)
   od
}
```

scope and visibility

- a never claim in a Spin model is defined *globally*
- within a claim we can therefore refer to:
  - global variables
  - message channels (using poll statements)
  - process control-flow states (remote reference operations)
  - predefined global variables such as timeout, _nr_pr, np_
  - but *not* process local variables
- bummer: in a never claim we cannot refer to *events*, we can only reason about properties of *states*...
  - so the effect of an event has to be made visible in the state of the system to become visible in a never claim
  - there is another mechanism available, not yet discussed, that can be used to reason about a limited subset of events: trace assertions (which can be used to refer only to send/recv events...)

impossible and inevitab

the model checker attempts to generate counter-examples to all claims, by showing that they can be violated...

- an *assertion* formalizes the claim
  - it is *impossible* for the given expression to evaluate to false when the assertion is reached
- an *end-state* label formalizes the claim
  - it is *impossible* for the system to terminate without all active processes having either terminated, or having stopped at a state that was marked with an end-state label
- a *progress-state* label formalizes the claim
  - it is *impossible* for the system to execute forever without passing through at least one of the states that was marked with a progress-state label infinitely often
- an *accept-state* label formalizes the claim
  - it is *impossible* for the system to execute forever while passing through at least one of the states that was marked with an accept-state label infinitely often
- a *never claim* formalizes the claim
  - it is *impossible* for the system to exhibit the behavior (finite or infinite) that completely matches the behavior that is specified in the claim
- a trace assertion formalizes the claim
  - it is *impossible* for the system to exhibit behavior that does not completely match the pattern defined in the trace assertion

trace assertions

- trace assertions can be used to reason about valid or invalid sequences of *send and receive* statements

```
mtype = { a, b };

chan p = [2] of { mtype };
chan q = [1] of { mtype };

trace {
   do
   :: p!a; q?b
   od
}
```

this assertion only claims something about how send operations on channel *p* relate to receive operations on channel *q*

it claims that every send of a message *a* to *p* is followed by a receive of a message *b* from *q*

a deviation from this pattern triggers an error

if at least one send (receive) operation on a channel q appears in the trace assertion, *all* send (receive) operations on that channel q must be covered by the assertion

only send and receive statements can appear in trace assertions

cannot use *variables* in trace assertions, only constants, mtypes or _

can use q?_ to specify an *unconditional* receive

notrace assertions

- reverses the claim: a notrace assertion states that a particular access pattern is impossible

```
mtype = { a, b };

chan p = [2] of { mtype };
chan q = [1] of { mtype };

notrace {
   if
   :: p!a; q?b
   :: q?b; p!a
   fi
}
```

this notrace assertion claims that there is *no execution* where the send of a message a to channel p is followed by the receive of a message b from q, or vice versa: it claims that there must be intervening sends or receives to break these two patterns of access

a notrace assertion is fully matched (producing and error report) when the closing curly brace is reached

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

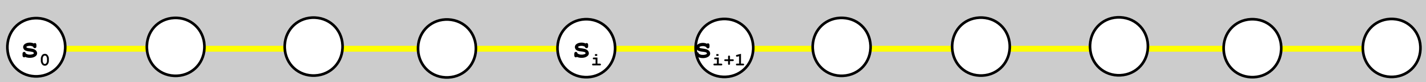
$\forall i, (i \geq 0)$ **and** $\forall p,\ s_i \models p$ is defined

we can define the semantics of the temporal logic formulae:

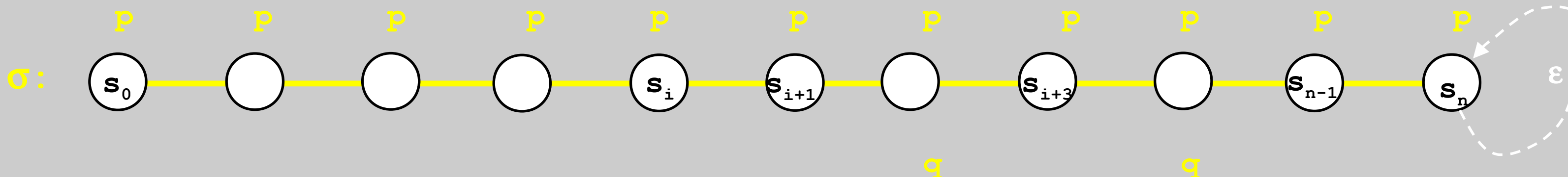
[]f, <>f, Xf, and e U f

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

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



examples



[]p

<>p

[]<>p

<>q

Xq

pUq (strong until)

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

<>(pUq) (weak until)

[]<>(p

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)

some standard LTL formulae

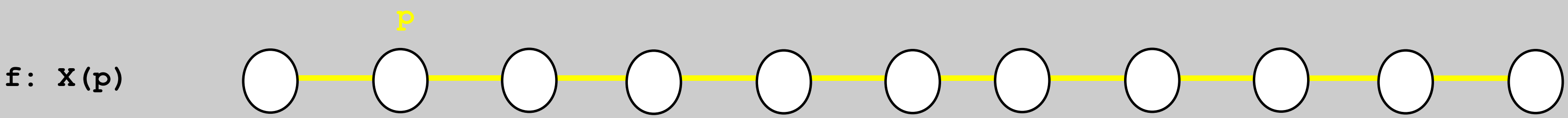
[] p	always p	invariance
<> p	eventually p	guarantee
p -> (<> q)	p implies eventually q	response
p -> (q U r)	p implies q until r	precedence
[]<> p	always, eventually p	recurrence (progress)
<>[] p	eventually, always p	stability (non-progress)
(<> p) -> (<> q)	eventually p implies eventually q	correlation

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 simplest operator: X



- 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)

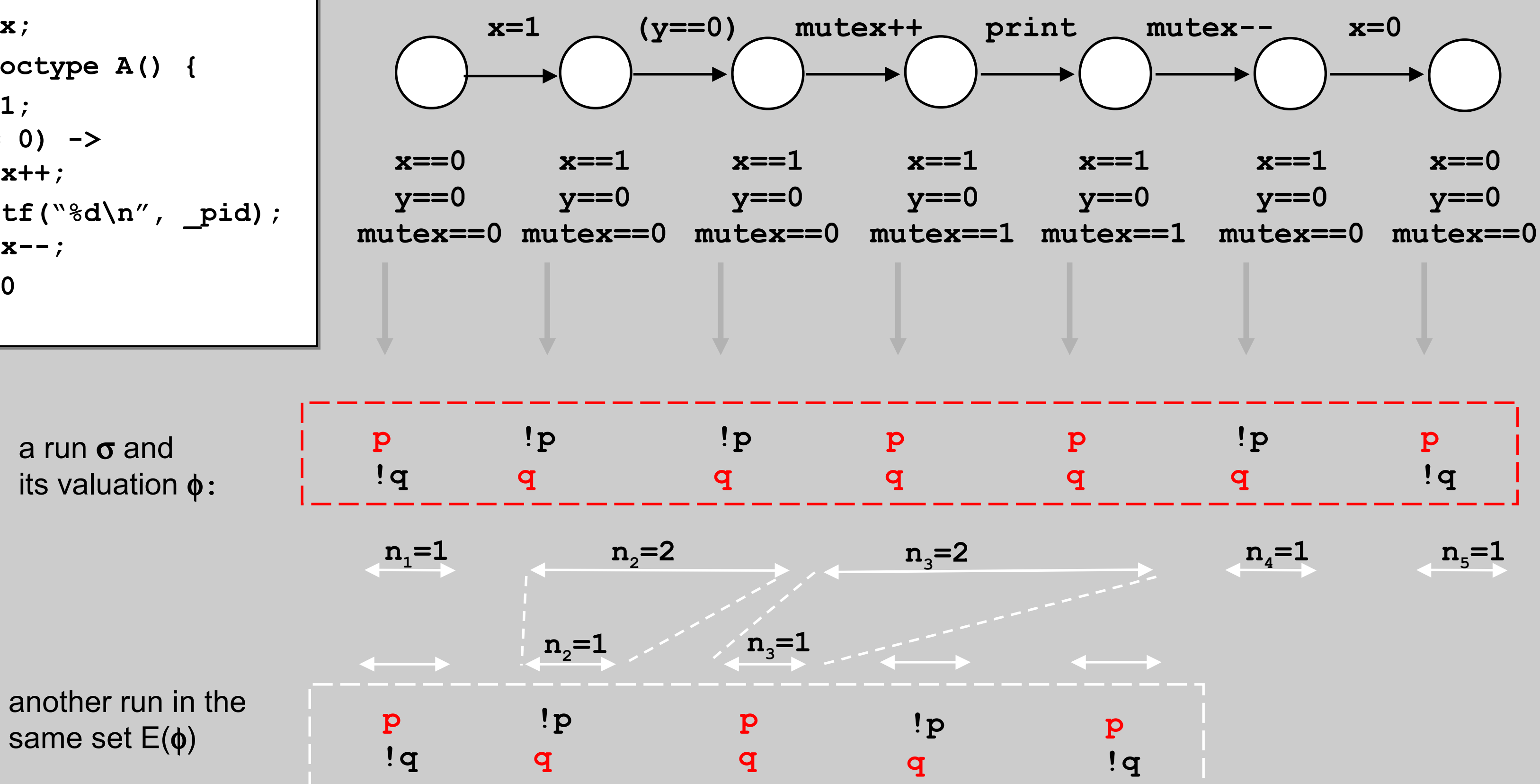
- Let ϕ = V(σ,P) be a *valuation* of a run σ for a given set of propositional formulae P (a path in the Kripke structure)
  - a series of truth assignment to all propositional formulae in P, for each subsequent state that appears in σ
  - the truth of any temporal logic formula in P can be determined for a run when the valuation is given
  - we can write ϕ as a series of intervals: ϕ_1^{n1}, ϕ_2^{n2}, ϕ_3^{n3}, ... 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
}
```



stutter invariant properties

(cf. book p. 139)

- a *stutter invariant* property is either true for *all* members of E(ϕ) or for *none* of them:

  $\forall\ \sigma \models f\ \wedge \phi = V(\sigma,P) \rightarrow \forall \nu \in E(\phi),\ \nu \models f$

- 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
```

from logic to automata

(cf. book p. 141)

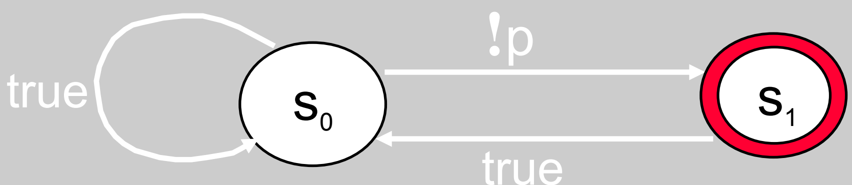
- for any LTL formula f there exists a Büchi automaton that *accepts* precisely those runs for which the formula f is satisfied

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

from logic to automata

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

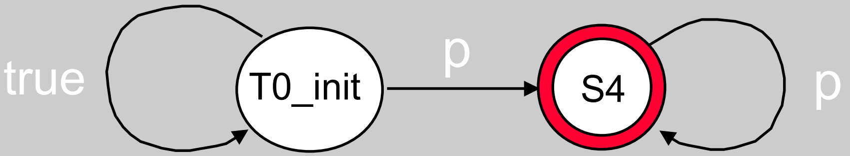
`!<>[]p ≡ []![]p ≡ []<>!p`



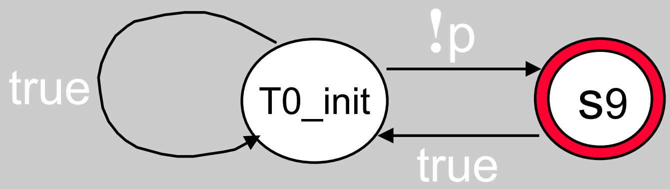
the automaton only accepts a run if p keeps returning to false infinitely often
i.e., securing that in the run considered p does not remain true invariantly, ever

using Spin to do the negations and the conversions

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




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

beware of operator precedence rules..

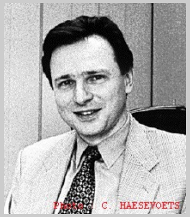
```
$ spin -f '[] (p -> <> q)'
never {    /* [](p -> <> q) */
T0_init:
        if
        :: (((! ((p))) || ((q)))) -> goto accept_S20
        :: (1) -> goto T0_S27
        fi;
accept_S20:
        if
        :: (((! ((p))) || ((q)))) -> goto T0_init
        :: (1) -> goto T0_S27
        fi;
accept_S27: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

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;
}
```

automata theoretic verification

Moshe Vardi



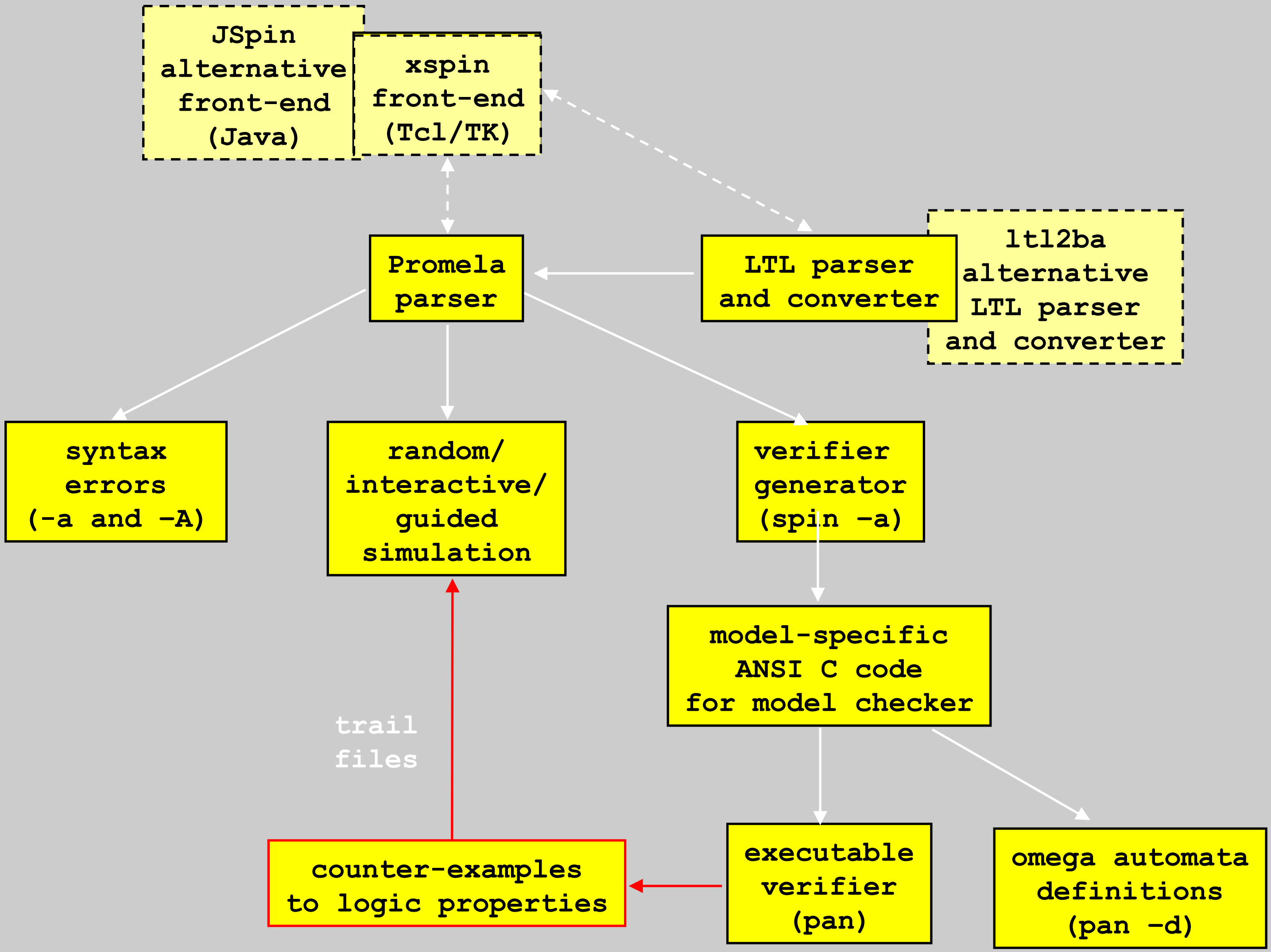
Pierre Wolper

- language of the model: L(model)
- language of the property: L(prop)
- prove that: $L(\text{model}) \subseteq L(\text{prop})$
- by showing that: $L(\text{model}) \cap (L^{\omega} \setminus L(\text{prop})) = \emptyset$
- which means: $L(\text{model}) \cap L(\neg\text{prop}) = \emptyset$

Spin checks if the intersection of an asynchronous product of process behaviors (the global model automaton) with a property automaton (generated from a negated LTL formula) is empty (i.e., accepts *no* runs).

All accepting runs of the resulting ω-automaton correspond to *violations* of the original (non-negated) property.

spin structure



formulating LTL properties

(book p. 148)

```
int x = 100;

active proctype A()
{
   do
   :: x%2 -> x = 3*x + 1
   od
}

active proctype B()
{
   do
   :: !(x%2) -> x = x/2
   od
}
```

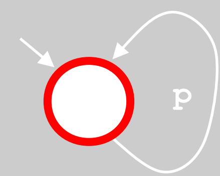
Q1: is the value of x bounded?

```
$ spin -f '[] (x > 0 && x <= 100)'
```

```
tl_spin: expected ')', saw '>'
tl_spin: [] (x > 0 && x <= 100)
----------------^
$
```

```
#define p (x > 0 && x <= 100)
```

```
$ spin -f '[]p'
never {   /* [] p */
accept_init:
T0_init:
   if
   :: p -> goto T0_init
   fi
}
```



Q2: there is another mistake here, what is it?

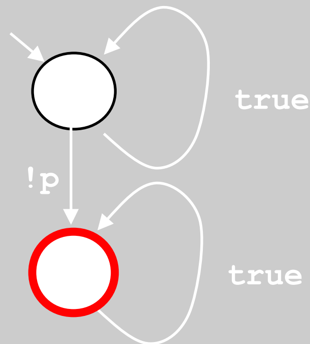
never claims are capture *error behavior*

negative, not *positive* properties

we forgot to negate the positive property into a claim

```
#define p (x > 0 && x <= 100)
```

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




```
if  property ![]p cannot be satisfied
    this means that
    property  []p cannot be violated
    (there is no counter-example)
```

negations

if property ![]p *cannot* be satisfied
this means that
property []p cannot be *violated*

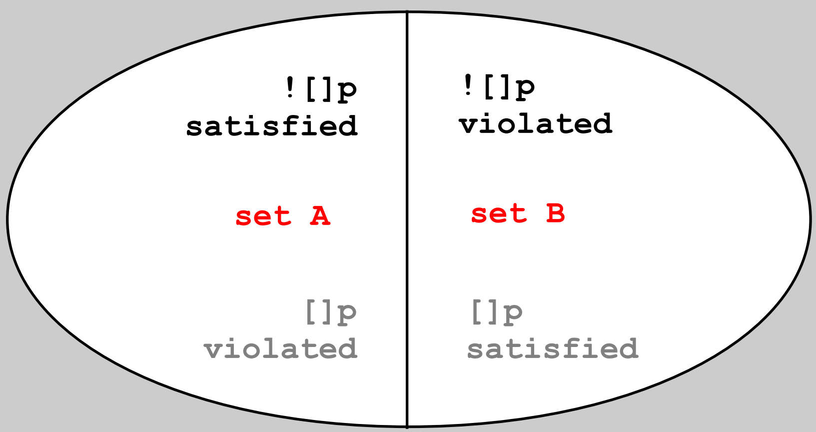
all runs are in set B

there are no runs in set A

there are runs in set B

there are no runs in set A

but if property ![]p can be *violated*
this does *not* mean that therefore
property []p *cannot* be violated



classification of all runs
each run either satisfies a property or it violates it

another property

```
int x = 100;

active proctype A()
{
   do
   :: x%2 -> x = 3*x + 1
   od
}

active proctype B()
{
   do
   :: !(x%2) -> x = x/2
   od
}
```

```
Q: is this formula satisfied?
     []<>p
with
#define p  (x == 1)
```

even in simple cases like this it can be very hard to determine the answer by eye-balling the program/model

to check if []<>p is *always* satisfied, prove that the negation can *never* be satisfied:

```
     !([]<>p)
     <>!(<>p)
     <>[]!p
```