- SMV —the Symbolic Model Verifier
- Example: the alternating bit protocol
- LTL —Linear Time temporal Logic
- CTL*
- Fixed Points
- Correctness

SMV - Symbolic Model Verifier was one of the first model checkers. It is based on CTL, was developed in early ’90, and had a strong impact on the verification field.

- SMV (Symbolic Model Verifier) was developed at CMU, see www.cs.cmu.edu/~modelcheck/smv.html
- it provides a language for describing the models/diagrams and it checks the validity of CTL formulas in such models
- the output is ‘true’ or a trace showing why the formula is false

SMV - Syntax (informal)

- SMV programs consist of one or more modules (one of them should be `main`)
- each module can declare variables and assign values to them
- assignment uses two qualifications: `initial` (to indicate the initial state) and `next` (to indicate the next state in the corresponding state transition diagram)
- the assignments may be nondeterministic - this is indicated by using the set notation $\{\ldots\}$ (choose one element form this set)

(...cont.)

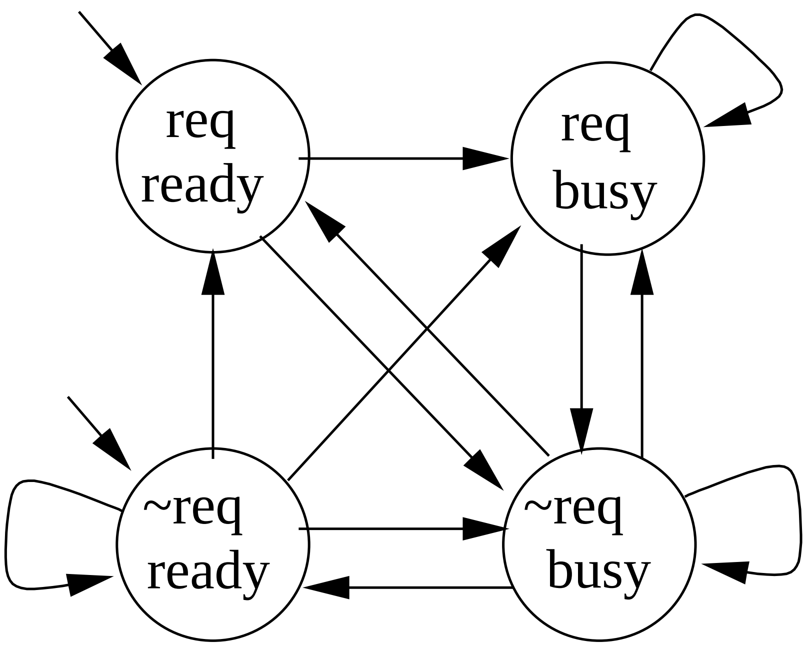
- one may use the `case` construct; in such a case the conditions in front of ':' are parsed from top to bottom and the first which is found true is executed; a default variant (with a always true condition, indicated by 1) is usually placed at the bottom of the `case` construct
- a module may have proper specifications to be checked, written in CTL syntax (but &, `|`, `->`, `!` are used instead of $\wedge, \vee, \rightarrow, \neg$)

Our first program is rather typical:

- it models a part of the system which pass from `ready` to `busy` either due to some hidden reasons (not seen in the model) or due to a visible request `request`;
- the system pass from `busy` to `ready` in a nondeterministic way, too (no visible reason)
- the intention of this simple abstract model is to check if it satisfies the formula
  `AG(request -> AF status = busy)`

...SMV, 1st example

```
MODULE main
VAR
   request :  boolean;
   status :  {ready,busy};
ASSIGN
   init(status) := ready;
   next(status) :=
     case
       request :  busy;
       1 :  {ready,busy};
     esac;
SPEC
   AG(request -> AF status = busy)
```



The 2nd program illustrates the use of modules:

- the program models a counter from 000 to 111
- a module `counter_cell` is instantiated 3 times with names `bit0`, `bit1`, and `bit2`
- `counter_cell` has a formal parameter
- the period '.' is used to access the variables of a particular instance (`m.v` indicates a reference to the variable `v` of module `m`)
- we check the following easy formula
  `AG AF bit2.carry_out`

```
MODULE main
VAR
   bit0 :  counter_cell(1);
   bit1 :  counter_cell(bit0.carry_out);
   bit2 :  counter_cell(bit1.carry_out);
SPEC
   AG AF bit2.carry_out

MODULE counter_cell(carry_in)
VAR
   value :  boolean;
ASSIGN
   init(value) := 0;
   next(value) := value + carry_in mod 2;
DEFINE
   carry_out := value & carry_in;
```

Note: `define` statement is used to avoid increasing the state space; its effect may be obtained with a variable, too:

```
VAR
   carry_out :  boolean;
ASSIGN
   carry_out := value & carry_in;
```

Synchronous and asynchronous composition

By default, SMV modules are composed *synchronously*:

> at each clock tick, each module executes a transition

(mainly used for hardware verification)

It is also possible to model *asynchronous* composition

> at each clock tick, SMV chooses a module in a random way and executes a transition there

(mainly used for verifying communication protocols)

SMV, 3rd example — Mutual Exclusion

A CTL model for 'mutual exclusion problem' was presented before. Here we give a SMV implementation. A few new features are:

- there is a module `main` with (1) a variable `turn` which determines the process to enter in its critical section and (2) two instantiations of the module `prc`
- because of the `turn` variable the state transition diagram (shown later) is slightly more complicate
- one important new feature is the presence of the `fairness` statement; it contains a CTL formula ϕ and restricts the search to those paths where ϕ is true infinitely often (`running` is an SMV keyword indicating that the corresponding module is selected for execution infinitely often)

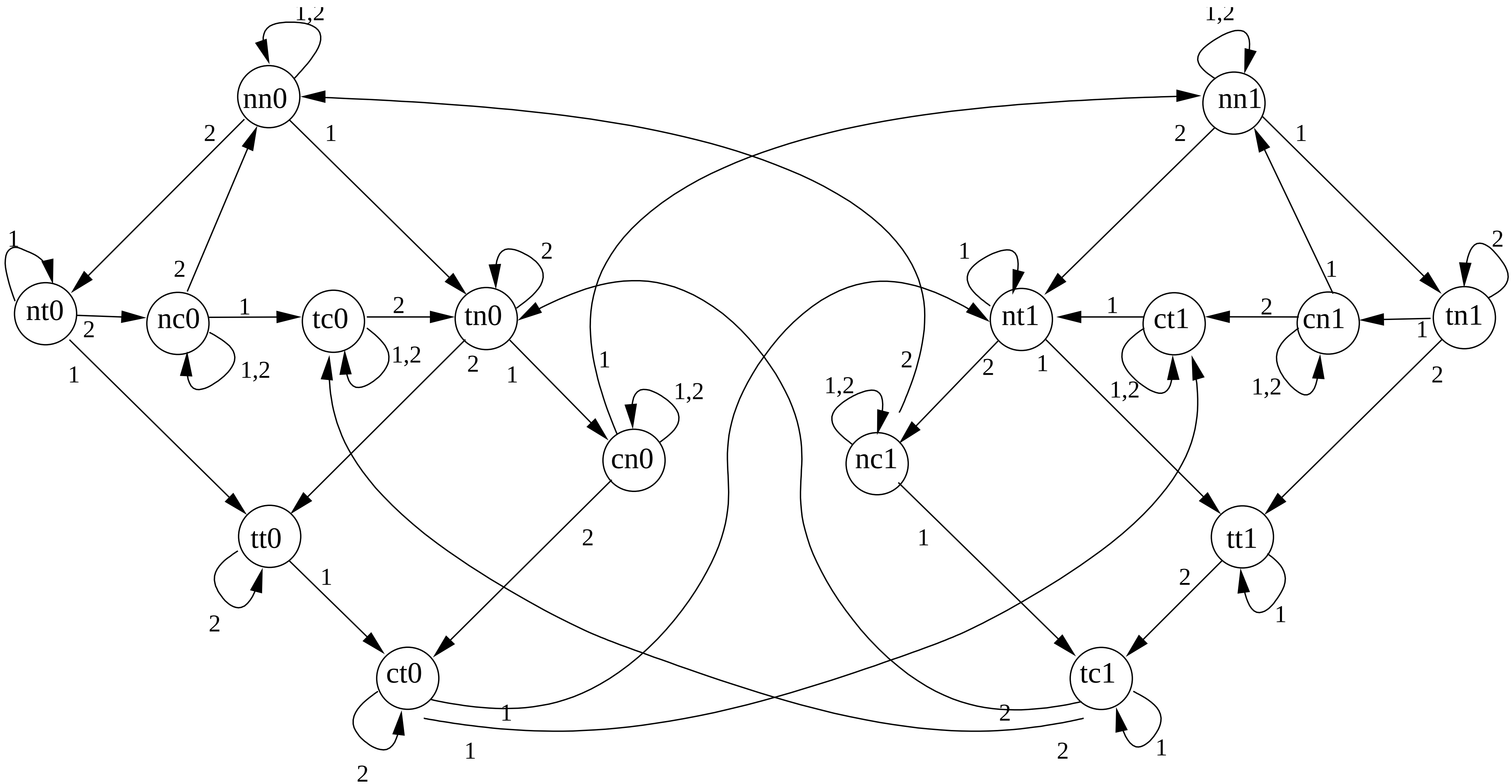
...SMV, 3rd example

```
MODULE main
   VAR
     pr1 :  process prc(pr2.st, turn, 0);
     pr2 :  process prc(pr1.st, turn, 1);
     turn :  boolean;
   ASSIGN
     init(turn) := 0;
   --safety
   SPEC AG!((pr1.st = c) & (pr2.st = c))
   --liveness
   SPEC AG((pr1.st = t) -> AF (pr1.st = c))
   SPEC AG((pr2.st = t) -> AF (pr2.st = c))
   --no strict sequencing
   SPEC EF(pr1.st = c & E[pr1.st = c U
          (!pr1.st = c & E[!  pr2.st = c U pr1.st = c
]))])
```

...SMV, 3rd example

```
MODULE prc(other-st, turn, myturn)
   VAR
     st :  {n, t, c};
   ASSIGN
     init(st) := n;
     next(st) :=
       case
         (st = n) :  {t, n};
         (st = t) & (other-st = n) :  c;
         (st = t) & (other-st = t) & (turn = myturn) :  c;
         (st = c) :  {c, n};
         1 :  st;
       esac;
     next(turn) :=
       case
         turn = myturn & st = c :  !turn;
         1 :  turn;
       esac;
   FAIRNESS running
   FAIRNESS !(st = c)
```

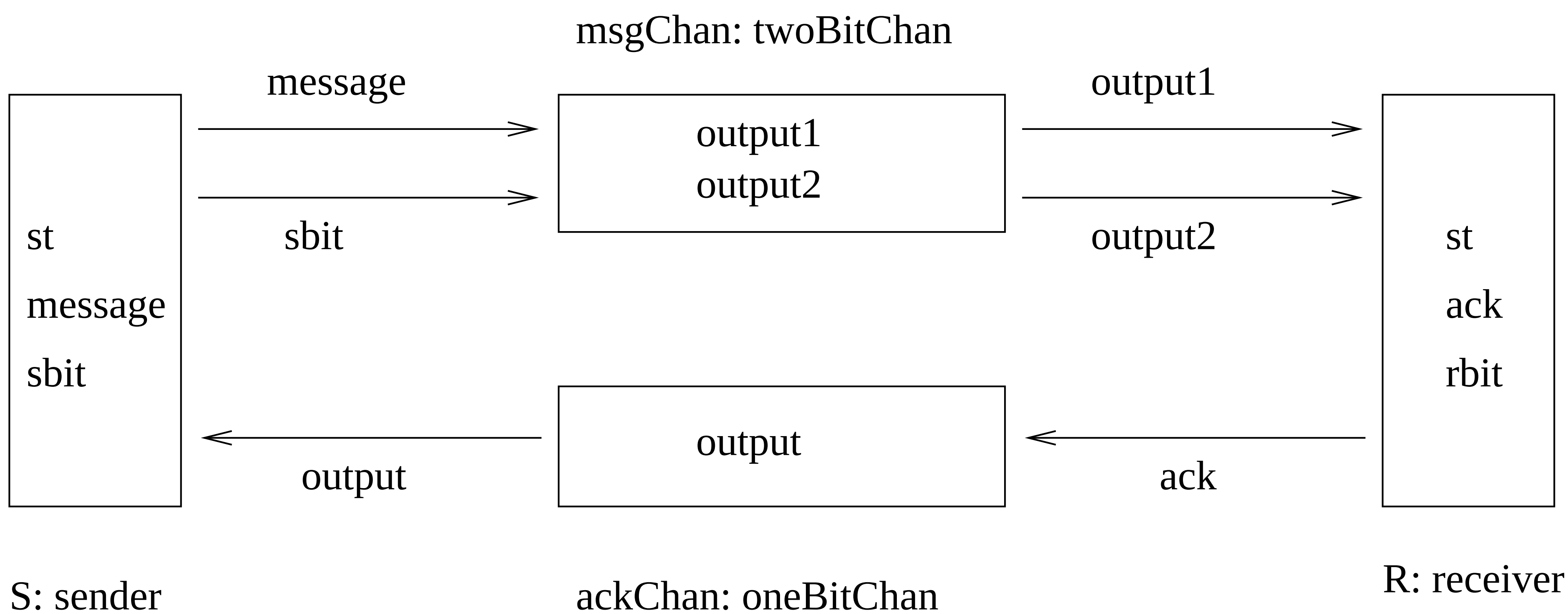
Mutual exclusion in SMV:



ABP: Alternating Bit Protocol

- The Alternating Bit Protocol ABP is a protocol for correctly transmitting data on faulty channels which may lose or duplicate data;
- ABP uses two faulty channels between a sender and a receiver: one to send data from the sender to the receiver and the other to send an acknowledgment from the receiver to the sender;
- in case of a unsuccessful transmission the attempt is repeated;
- to achieve it goal, APB keeps track on this repeated sendings using a control bit which is switched when the sending pass from one datum to another: the sender appends its control bit to the datum to be send and keeps sending till it receives this control bit back via the acknowledgement channel

The figure below describes the structure of the ABP.



...APB

```
MODULE sender(ack)
VAR
  st : {sending, sent};
  message : boolean;
  sbit : boolean;
ASSIGN
  init(st) := sending;
  next(st) :=
    case
      ack = sbit & !(st = sent) : sent;
      1 : sending;
    esac;
  next(message) :=
    case
      st = sent : {0, 1};
      1 : message;
    esac;
  next(sbit) :=
    case
      st = sent : !sbit;
      1 : sbit;
    esac;
FAIRNESS running
SPEC AG AF st = sent
```

...APB

```
MODULE receiver(message, sbit)
VAR
  st :  {receiving, received};
  ack :  boolean;
  rbit :  boolean;
ASSIGN
  init(st) := receiving;
  next(st) :=
    case
      sbit = rbit & !(st = received) :  received;
      1 :  receiving;
    esac;
  next(ack) :=
    case
      st = received :  sbit;
      1 :  ack;
    esac;
  next(rbit) :=
    case
      st = received :  !rbit;
      1 :  rbit;
    esac;
FAIRNESS running
SPEC AG AF st = received
```

...APB

```
MODULE oneBitChan(input)
VAR
  output :  boolean;
ASSIGN
  next(output) := {input, output};
FAIRNESS running
FAIRNESS (input = 0 -> AF output = 0) & (input = 1
  -> AF output = 1)

MODULE twoBitChan(input1, input2)
VAR
  output1 :  boolean;
  output2 :  boolean;
ASSIGN
  next(output2) := {input2, output2};
  next(output1) :=
    case
      input2 = next(output2) :  input1;
      1 :  {input1, output1};
    esac;
FAIRNESS running
FAIRNESS (input1 = 0 -> AF output1 = 0) & (input1 = 1
  -> AF output1 = 1) & (input2 = 0 -> AF output2 = 0)
  & (input2 = 1 -> AF output2 = 1)
```

```
MODULE main
VAR
  S : process sender(ackChan.output);
  R : process receiver(msgChan.output1, msgChan.output2);
  msgChan :  process twoBitChan(S.message, S.sbit);
  ackChan :  process oneBitChan(R.ack);
ASSIGN
  init(S.sbit) := 0;
  init(R.rbit) := 0;
  init(R.ack) := 1;
  init(msgChan.output2) := 1;
  init(ackChan.output) := 1;
SPEC AG(S.st = sent & S.message = 1 -> msgChan.output1 = 1)
```

The are many specification languages for reactive systems, e.g.:

- regular expressions
- state-chats
- graphical interval logics
- modal mu-calculus
- linear time temporal logic
- CTL
- CTL*
- …

LTL (linear time temporal logic) is closely related to CTL. Its syntax is the following:

$$\phi ::= \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi\ \mathrm{U}\ \phi) \mid (\mathrm{G}\ \phi) \mid (\mathrm{F}\ \phi) \mid (\mathrm{X}\ \phi)$$

Examples:

$$\mathrm{GF}\ p$$

$$\mathrm{FG}\ p$$

$$\mathrm{G}(p \vee \mathrm{X}\ p)$$

$$\mathrm{G}\ p \rightarrow \mathrm{F}\ q$$

Comments:

- a LTL formula is evaluated on a path or set of paths; for this reason the CTL qualifications E (there exists a branch) and A (all branches) are dropped (in this respect LTL looks to be less expressive than CTL)
- however, LTL allows nesting modal operators in a way not allowed in CTL, e.g., $\mathrm{GF}\phi$ (in this respect LTL looks to be more expressive than CTL)

Apparently LTL is more permissive as it allows for boolean combinations of paths, but this may be done in CTP, too.

LTL semantics

Let $\mathcal{M} = (S, \rightarrow, L)$ be a (CTL-like) model and $\pi = s_1 \rightarrow \ldots$ a path; π^i denotes the path $s_i \rightarrow s_{i+1} \rightarrow \ldots$

The satisfaction relation $\pi \models \phi$ is inductively defined as follows:

1. $\pi \models \top$
2. $\pi \models p$ iff $p \in L(s_1)$
3. $\pi \models \neg\phi$ iff $\pi \not\models \phi$
4. $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$
5. $\pi \models \mathrm{X}\ \phi_1$ iff $\pi^2 \models \phi_1$
6. $\pi \models \mathrm{G}\ \phi_1$ iff for all $i \geq 1$, $\pi^i \models \phi_1$
7. $\pi \models \mathrm{F}\ \phi_1$ iff for some $i \geq 1$, $\pi^i \models \phi_1$
8. $\pi \models \phi_1\ \mathrm{U}\ \phi_2$ iff there is some $i \geq 1$ such that $\pi^i \models \phi_2$ and for all $j = 1, \ldots, i-1$ we have $\pi^i \models \phi_1$

- Two LTL formulas ϕ and ψ are *semantically equivalent*, written $\phi \equiv \psi$, if for any model they are true for the same paths.
- An LTL formula ϕ is *satisfied in a state s* of a model $\mathcal{M}$ if ϕ holds for all paths starting at s.

Examples

$$\mathrm{G}\ \phi \equiv \neg \mathrm{F}\ \neg \phi$$
$$\mathrm{F}(\phi \vee \psi) \equiv \mathrm{F}\ \phi \vee \mathrm{F}\ \psi$$
$$\mathrm{G}(\phi \wedge \psi) \equiv \mathrm{G}\ \phi \wedge \mathrm{G}\ \psi$$

Note: From the CTL point of view, a LTL formula ϕ is identified with $\mathrm{A}[\phi]$ (all paths are considered when formula satisfiability is to be checked)

For all LTL formulas ϕ and ψ

$$\neg(\phi \,\mathrm{U}\, \psi) \equiv \neg\psi \,\mathrm{U}(\neg\phi \wedge \neg\psi) \vee \mathrm{G}\,\neg\psi$$

Proof:

$\neg(\phi \,\mathrm{U}\, \psi)$ is true

iff

(1) either ψ is always false or

(2) ϕ is false before ψ becomes true

iff

(1) either $\mathrm{G}\,\neg\psi$ is true or

(2) $\neg\psi \,\mathrm{U}(\neg\phi \wedge \neg\psi)$

iff

$\neg\psi \,\mathrm{U}(\neg\phi \wedge \neg\psi) \vee \mathrm{G}\,\neg\psi$ is true

An equivalent form is: $\phi \,\mathrm{U}\, \psi \equiv \neg(\neg\psi \,\mathrm{U}(\neg\phi \wedge \neg\psi)) \wedge \mathrm{F}\,\psi$

The syntax of CTL* define *state formulas* and *path formulas* using the following mutually recursive definitions:

- state formulas (to be evaluated in states)

$$\phi ::= \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid \mathrm{A}[\alpha] \mid \mathrm{E}[\alpha]$$

- paths formulas (to be evaluated along paths)

$$\alpha ::= \phi \mid (\neg\alpha) \mid (\alpha \wedge \alpha) \mid (\mathrm{X}\ \alpha) \mid (\mathrm{G}\ \alpha) \mid (\mathrm{F}\ \alpha) \mid (\alpha\ \mathrm{U}\ \alpha)$$

Examples:

- $\mathsf{A}[(p\ \mathsf{U}\ r) \vee (q\ \mathsf{U}\ r)]$: along all paths, either p is true until r, or q is true until r (not equivalent to $\mathsf{A}[(p \vee q)\ \mathsf{U}\ r]$)
- $\mathsf{A}[\mathsf{X}\ p \vee \mathsf{XX}\ p]$: p is true in the next state, or in the next next state (not equivalent to $\mathsf{AX}\ p\ \vee \mathsf{AX}\ \mathsf{AX}\ p$)
- $\mathsf{E}[\mathsf{GF}\ p]$: there is a path along which p is infinitely many true (not equivalent to $\mathsf{EG}\ \mathsf{EF}\ p$)

Let $\mathcal{M} = (S, \rightarrow, L)$ be a model.

- If ϕ is a state formula, the notation $\mathcal{M}, s \models \phi$ means that ϕ holds in state s.
- If α a path formula, then $\mathcal{M}, \pi \models \alpha$ means that α holds along path π.

These relations are inductively defined as follows:

1. $\mathcal{M}, s \models p$ iff $p \in L(s)$
2. $\mathcal{M}, s \models \neg\phi$ iff $\mathcal{M}, s \not\models \phi$
3. $\mathcal{M}, s \models \phi_1 \wedge \phi_2$ iff $\mathcal{M}, s \models \phi_1$ and $\mathcal{M}, s \models \phi_2$
4. $\mathcal{M}, s \models \mathrm{A}[\alpha]$ iff for any path π starting from s, $\mathcal{M}, \pi \models \alpha$
5. $\mathcal{M}, s \models \mathrm{E}[\alpha]$ iff there is a path π starting from s such that $\mathcal{M}, \pi \models \alpha$

6. $\mathcal{M}, \pi \models \phi$ iff s is the first state of π and $\mathcal{M}, s \models \phi$
7. $\mathcal{M}, \pi \models \alpha_1 \wedge \alpha_2$ iff $\mathcal{M}, \pi \models \alpha_1$ and $\mathcal{M}, \pi \models \alpha_2$
8. $\mathcal{M}, \pi \models \mathrm{X}\alpha$ iff $\mathcal{M}, \pi^1 \models \alpha$
9. $\mathcal{M}, \pi \models \mathrm{G}\alpha$ iff for all $k \geq 0$, $\mathcal{M}, \pi^k \models \alpha$
10. $\mathcal{M}, \pi \models \mathrm{F}\alpha$ iff there exists a $k \geq 0$ such that $\mathcal{M}, \pi^k \models \alpha$
11. $\mathcal{M}, \pi \models \alpha_1 \; \mathrm{U} \; \alpha_2$ iff there exists a $k \geq 0$ such that $\mathcal{M}, \pi^k \models \alpha_2$ and for all $0 \leq j < k$, $\mathcal{M}, \pi^j \models \alpha_1$

- CTL is the particular case of CTL* where the paths formulas are restricted to

$$\alpha ::= (\mathbf{X}\ \phi) \mid (\mathbf{G}\ \phi) \mid (\mathbf{F}\ \phi) \mid (\phi\ \mathbf{U}\ \phi)$$

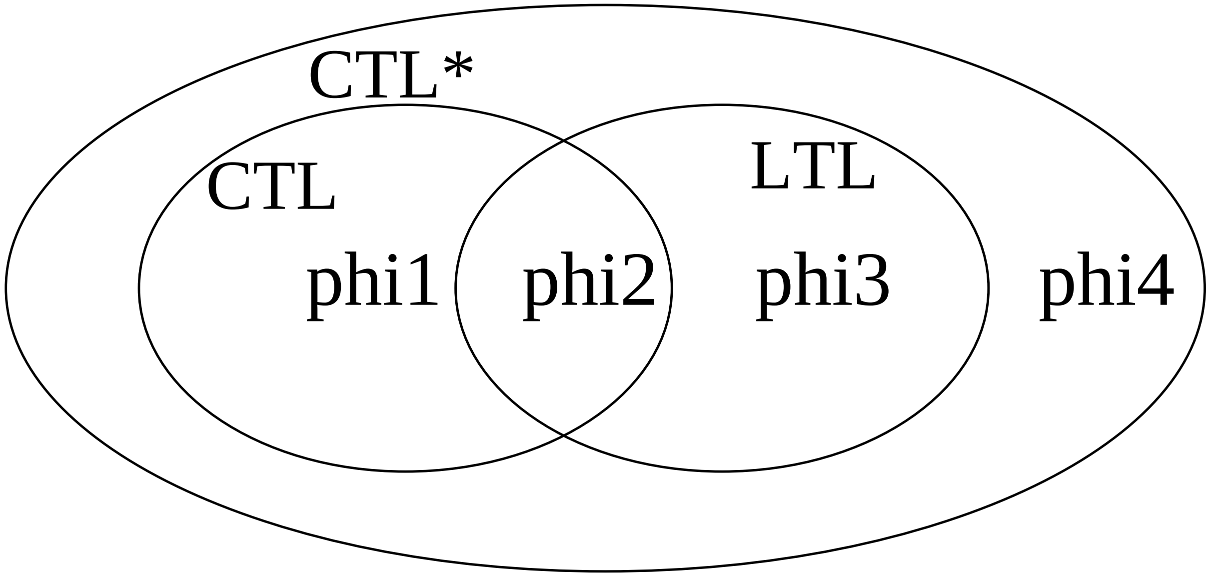
where ϕ is a state formula. (In other words, each temporal operator is directly preceded by a path quantification A or E leading to the known CTL operators consisting of ‘two letters’: AG, etc.)

- an LTL formula α is identified with CTL* formula

$$\mathbf{A}[\alpha]$$

(semantically all paths are considered when LTL formula satisfiability is checked)

- LTL and CTL are incomparable with respect to their expressive power
- a useful common extension CTL* was developed and extensively studied



Example:

$\phi_1 = AG(EF\ p)$

$\phi_3 = A[FG\ p]$

$\phi_4 = \phi_1 \vee \phi_3$

Or:

$\phi_3 = A[GF\ p \rightarrow F\ q]$

$\phi_4 = E[GF\ p]$

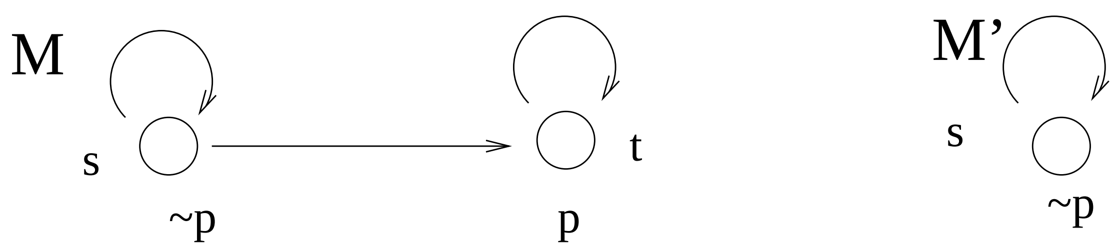
The CTL formula

$$\texttt{phi1} = \mathsf{AG}\ \mathsf{EF}\ p$$

describes

> "wherever we have got to, we can always get back to a state in which p is true"

This property can not be expressed in LTL. If there is an LTL formula ϕ such that $\mathsf{A}[\phi] \equiv \mathsf{AG}\ \mathsf{EF}\ p$, then with respect to the diagram



$\mathcal{M}, s \models \mathsf{AG}\ \mathsf{EF}\ p$ is valid, hence also $\mathcal{M}, s \models \mathsf{A}[\phi]$. On the other hand, the paths in $\mathcal{M}'$ of the diagram is a subset of the paths in $\mathcal{M}$, hence $\mathcal{M}', s \models \mathsf{A}[\phi]$, but this is not true.

The LTL formula

$$\texttt{phi3} = \mathrm{A}[\mathrm{GF}\ p \rightarrow \mathrm{F}\ q]$$

describes

"if there are infinitely many p along the path,
then there is an occurrence of q"

This property can not be expressed in CTL.

The CTL* formula

$$\texttt{phi4} = \mathrm{E}[\mathrm{GF}\ p]$$

describes

"there is a path with infinitely many p"

This property can be expressed neither in CTL nor in LTL.

- Boolean combinations of paths in CTL:
  - $\mathrm{E}[\mathrm{F}p \wedge \mathrm{F}q] \equiv \mathrm{EF}[p \wedge \mathrm{EF}q] \vee \mathrm{EF}[q \wedge \mathrm{EF}p]$
  - $\mathrm{E}[(p_1 \mathrm{U} q_1) \wedge (p_2 \mathrm{U} q_2)] \equiv \mathrm{E}[(p_1 \wedge p_2)\mathrm{U}(q_1 \wedge \mathrm{E}[p_2 \mathrm{U} q_2])] \vee \mathrm{E}[(p_1 \vee p_2)\mathrm{U}(q_2 \wedge \mathrm{E}[p_1 \mathrm{U} q_1])]$
  - $\mathrm{E}[\neg(p \mathrm{U} q)] \equiv \mathrm{E}[\neg q \mathrm{U}(\neg p \wedge \neg q)] \vee \mathrm{EG}\neg q$
- The *weak until* operator W is defined in LTL or CTL* by
  - $p \mathrm{W} q \equiv (p \mathrm{U} q) \vee \mathrm{G}p$

  This does not work in CTL, but the following identities do the job

  - $\mathrm{E}[p \mathrm{W} q] \equiv \mathrm{E}[p \mathrm{U} q] \vee \mathrm{EG}p$
  - $\mathrm{A}[p \mathrm{W} q] \equiv \neg \mathrm{E}[\neg q \mathrm{U} \neg(p \vee q)]$

Fixed points

- Let S be a set of states and $F : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ a function.
- F is called *monotone* if $X \subseteq Y$ implies $F(X) \subseteq F(Y)$.
- An $X \in \mathcal{P}(S)$ is called *fixed point* if $F(X) = X$.
- Denote $F^k(X) = F(F(\ldots F(X)\ldots))$, where F is applied k times.
- F is called *continuous* if $F(\bigcup X_i) = \bigcup F(X_i)$ for any increasing sequence $X_0 \subseteq X_1 \subseteq X_2 \ldots$.

A well-known theorem of Kleene shows that in such a setting

- *a monotone and continuous* F *has both a least fixed point, denoted* $\mu Z.F(Z)$*, and a greatest fixed point, denoted denoted* $\nu Z.F(Z)$*;*
- *moreover, the following formulas may be used to compute them:*

$$\mu Z.F(Z) = \emptyset \cup F(\emptyset) \cup F(F(\emptyset)) \cup \ldots$$

and

$$\nu Z.F(Z) = S \cap F(S) \cap F(F(S)) \cap \ldots$$

In the special case when S is finite, say with n elements, the continuity condition is not necessary. Indeed, *Theorem: If S has n elements and F is monotone, then*

$$\mu Z.F(Z) = F^n(\emptyset) \quad \text{and} \quad \nu Z.F(Z) = F^n(S)$$

Proof:

(1) Clearly $\emptyset \subseteq F^1(\emptyset)$; applying F we get $F^1(\emptyset \subseteq F^2(\emptyset)$; repeating, we get: $\emptyset \subseteq F^1(\emptyset) \subseteq F^2(\emptyset) \subseteq \ldots \subseteq F^{n+1}(\emptyset)$ The above chain of inclusions can not be strict, hence one of '$\subseteq$' should in fact be an equality (otherwise at each step we add at least one element, hence $F^{n+1}(\emptyset)$ will have at least $n+1$ elements, which is not possible); it follows that for some $0 \leq i_0 \leq n$, $F^{i_0}(\emptyset) = F(F^{i_0}(\emptyset))$, which entails that $F^{i_0}(\emptyset)$ is a fixed point.

(2) To show that $F^i(\emptyset)$ is less than any other fixed point is easy: Let X be a fixed point; then $\emptyset \subseteq X$; applying F we get $F(\emptyset) \subseteq F(X) = X$; repeating, we get that $F^k(\emptyset) \subseteq X$ for any k, hence $F^{i_0}(\emptyset) \subseteq X$.

(3) The case of the greatest fixed point is similar, but one has to start with S and the reverse the inclusions.

Denote by $[\![\phi]\!]$ the set of states satisfying ϕ and by F the mapping

$$Z \mapsto [\![\psi]\!] \cup ([\![\phi]\!] \cap \{s : \text{exists } s' \text{ such that } s \rightarrow s' \text{ and } s' \in Z\}$$

Theorem: If F is as above and $n = |S|$, then: (1) F is monotone: (2) $[\![\mathrm{E}[\phi \mathrm{~U~} \psi]]\!]$ is the least fixed point of F; and (3) $[\![\mathrm{E}[\phi \mathrm{~U~} \psi]]\!] = F^{n+1}(\emptyset)$

Proof:

(1) The mapping $H(Z) = \{s : \text{exists } s' \text{ such that } s \rightarrow s' \text{ and } s' \in Z\}$ is monotone (similar to a tutorial question). F is obtained from H by intersection and union with certain sets, hence is monotone, too.

(2) Looking at the states $F^k(\emptyset)$ we see that

—$F^0(\emptyset)$ contains the states in $[\![\psi]\!]$;

—$F^1(\emptyset)$ contains the states in $[\![\psi]\!]$, or those in $[\![\phi]\!]$ which have transitions to states in $[\![\psi]\!]$;

—...

In general,

$F^k(\emptyset)$ contains those states which have a path of length less than k to a state in $[\![\psi]\!]$ going through states in $[\![\phi]\!]$, only

hence the union of all $F^k(\emptyset)$ gives $[\![\mathrm{E}[\phi\ \mathrm{U}\ \psi]]\!]$.

We know that the chain $F^k(\emptyset)$ is increasing and $F^{n+1}(\emptyset)$ is a fixed point, hence the union of all $F^k(\emptyset)$ is just $F^{n+1}(\emptyset)$.

(3) already shown at (2)

The final observation is that $\textsc{sat}_{EU}$ uses an equivalent, but somehow simpler iterative process: instead of

$$F^{k+1}(\emptyset) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \{s : \text{ exists } s' \text{ such that } s \rightarrow s' \text{ and } s' \in F^{k}(\emptyset)\}$$

it uses the iterative process

$$F_1^{k+1}(\emptyset) = F_1^{k}(\emptyset) \cup ([\![\phi]\!] \cap \{s : \text{ exists } s' \text{ such that } s \rightarrow s' \text{ and } s' \in F_1^{k}(\emptyset)\}$$

```
function SAT_EG(φ):
/* pre: φ is an arbitrary CTL formula */
/* post: SAT_EG(φ) returns the set of states satisfying EG φ */

local var X, Y
begin
   X := ∅;
   Y := SAT(φ);
   repeat until X = Y
      begin
         X := Y;
         Y := Y ∩ {s ∈ S : exists s' with s → s' and s' ∈ Y};
      end
   return Y
end
```

Correctness of SAT_{EG}

Denote by $[\![\phi]\!]$ the set of states satisfying ϕ and by G the mapping

$$Z \mapsto [\![\phi]\!] \cap \{s\colon \text{exists } s' \text{ such that } s \to s' \text{ and } s' \in Z\}$$

Theorem: If F is as above and $n = |S|$, then: (1) G is monotone: (2) $[\![\text{EG } \phi]\!]$ is the greatest fixed point of G; and (3) $[\![\text{EG } \phi]\!] = G^{n+1}(S)$

The proof is similar to the previous theorem.

Finally, instead of the iterative process

$$G^{k+1}(S) = [\![\phi]\!] \cap \{s\colon \text{exists } s' \text{ such that } s \to s' \text{ and } s' \in G^k(S)\}$$

the SAT_{EG} algorithm uses the simpler equivalent iterative process

$$G_1^{k+1}(S) = G^k(S) \cap \{s\colon \text{exists } s' \text{ such that } s \to s' \text{ and } s' \in G^k(S)\}$$