NICE NOTES of the first 5 hours

up to 20.03.2009

Model Checking

- verification technique for automatic verifying finite systems.

$$\text{Model} \stackrel{?}{\models} \text{property}$$

automaton/
transition system/
LTS

temporal logic formula

"does it satisfy"
check by brute-force search

[the semantics of the formulas is in term of LTS states]

models for properties

Two approaches:

preferable

1. make a model, verify, if fine, implement
2. given an implemented system, extract a model, verify

Why verification? We want correctness
(not easy to see / concurrency problems)

(model based)
verification vs. validation

"are we building the right thing?"
(as good as the model itself)

"are we using the right model?"

Verification methods

- peer reviewing [informal]
- simulation } can work without a model,
- testing } but never complete
- deduction reasoning (theorem proving / process algebra)
  - very theoretical
  - requires expertise
- model checking
  (automated, fairly efficient, up to 10^{120} states, but only on finite models)

The model checking process consists of:
- modelling (part of any model-based formal verif. technique)
- specification (of properties to be checked)
- verification (YES / NO + counterexample trace / OUT-OF-MEMORY)
- analysis of results (requires some expertise)

Models are: LTS / Kripke structures / automata

Properties are: temporal logic formulas

a kind of modal logic.

extension of propositional logic with temporal / modal operators

e.g. $G\,e$ - "e holds globally" (in any reachable state from a given state)

$G(\neg e_1 \vee \neg e_2)$ - "e_1 and e_2 never happen at the same time"

Temporal logics: LTL, CTL, CTL*

Complexity of Model Checking

- Both for LTL and CTL, MC algorithms are
  - ✓ - exponential in the size of the formula (fine - formulas are short)
  - :-(- linear in the size of the model (bad - models are big)

Drawback: state space explosion

e.g. n - binary variables $\Rightarrow$ 2^n states

How to fight it?

- symbolic model checking (OBDD's)
  ordered binary decision diagrams
- partial order reduction (reduces the model)
- symmetry (—"—)
- compositional reasoning ("assume - guarantee" reasoning)
- abstraction (e.g. for systems with data)
- induction ("invariant" model representing a whole family of models)

→ may enable verification of infinite systems (which are reduced to finite ones, which are model checked)

Modelling Systems

- Model is an abstraction of the system which often takes into account the "specification" (the properties that we want to check)

- We deal with reactive systems
  - high level of interaction with the environment
  - no termination

  (state-transition models)
  - LTS, Kripke structures

Labelled Transition Systems

① State-labelled : Kripke structures

Def. A Kripke structure is a 4-tuple $M = (S_0, S, R, L)$ where
- S is a finite set of <u>states</u>
- S_0 is a finite set of <u>initial states</u>, $S_0 \subseteq S$
- $R \subseteq S \times S$ is the <u>transition relation</u> (total)
- $L: S \to 2^{AP}$ is the <u>labelling function</u> for a set of atomic propositions AP.

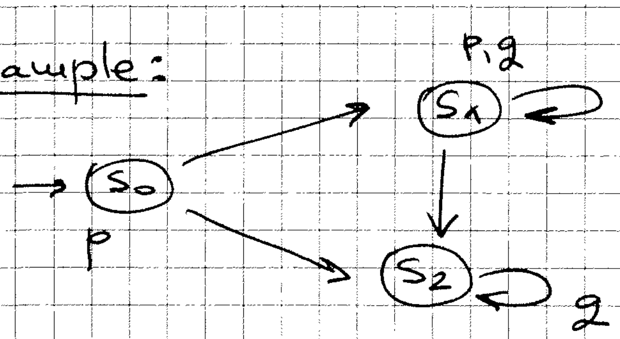
R is total if $(\forall s \in S)(\exists s' \in S)\ (s, s') \in R$

we write $s \to s'$ if $(s, s') \in R$ or also sRs'.

A <u>path</u> in M is an infinite sequence
$\pi = s_0 s_1 s_2 \dots$ s.t. $(\forall i \geq 0)\ s_i \to s_{i+1}$ in M.

if $s_0 \in S_0$, then π is called an <u>execution</u>.

Example:



$AP = \{p, q\}$

$S = \{s_0, s_1, s_2\}$, $S_0 = \{s_0\}$

$R = \{(s_0, s_1), (s_0, s_2), (s_1, s_1), (s_1, s_2), (s_2, s_2)\}$

$L(s_0) = \{p\}$, $L(s_1) = \{p, q\}$, $L(s_2) = \{q\}$.

② Transition-labelled systems

Def. A transition-labelled transition system is a 4-tuple $M = (S_0, S, A, R)$ where

- S is a finite set of states
- $S_0 \subseteq S$ is the set of initial states
- A – set of action labels
- $R \subseteq S \times A \times S$ is the transition relation

for $(s, a, s') \in R$ we write $s \xrightarrow{a} s'$.

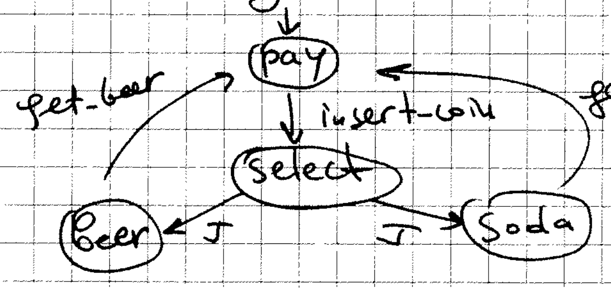
A path in M is an infinite sequence

$\pi = s_0 a_0 s_1 a_1 s_2 a_2 \ldots$

s.t. $(\forall i \geq 0)$ $s_i \xrightarrow{a_i} s_{i+1}$

Execution is a path that starts in an initial state.

Example (vending machine)



most general

(3) LTS with labelled states and transitions

$M = (S, S_0, A, R, L)$ — with the previously defined meanings

One can always label states by

[redacted] $L(s) = \{s\}$ i.e. $AP = S$

- We will model concurrent systems via 1st order formulas (Predicate logic)

  - Predicate and functional symbols are predefined (are interpreted)

  - Given a set of variables $V = \{v_1, .., v_n\}$ with values in a finite domain D.

    A <u>valuation</u> for V is any function $s : V \to D$.

  - A <u>state</u> of a concurrent system with variables V is any valuation [equivalently, state is $s \in D^n$]

  - Given a state, we can write a formula that is true for exactly that valuation, namely

$$v_1 = s(v_1) \wedge v_2 = s(v_2) \wedge \dots \wedge v_n = s(v_n)$$

In general, a formula is true for a set of valuations.

<u>Convention</u>: A formula φ is identified with the set of valuations that make it true.

- For concurrent systems, we describe the set of initial states S_0 by a formula $\mathcal{S}_0$.

How about the transitions?

For the transitions, let $V' = \{v' \mid v \in V\}$
$= \{v_1', \dots, v_n'\}$

A transition is a valuation over $V \cup V'$

old variables (before the transition is taken)

new variables (after the transition)

- $\mathcal{R}(V, V')$ is a 1st order formula that represents all transitions in a concurrent system.

- For atomic propositions we take $AP = \{v = d \mid v \in V, d \in D\}$ and write $s \models v = d$ iff $s(v) = d$.

Now, out of a concurrent system with variables V, initial states described by $\mathcal{S}_0$, transitions by $\mathcal{R}(V, V')$ we extract a Kripke structure as follows.

$M = (S_0, S, R, L)$ where

- $S = \{s : V \to D \mid s \text{ - valuation}\}$
- $S_0 = \{s_0 \in S \mid s_0 \models \mathcal{S}_0\}$
- $R(s, s')$ holds if $\mathcal{R}(V, V')$ is true for the valuation $\sigma : V \cup V' \to D$ given by

$$\sigma(v) = s(v), \quad \text{for } v \in V$$
$$\sigma(v') = s'(v), \quad \text{for } v' \in V' \ (v \in V)$$

- the atomic proposition $(v = d) \in L(s)$ iff $s \models v = d$

<u>Example:</u> $V = \{x, y\}$, $D = \{0,1\}$

a state (valuation) is $(\underbrace{d_1}_{\text{value for } x}, \underbrace{d_2}_{\text{value for } y}) \in D^2$

– one "transition formula", one instruction

$$x := (x+y) \bmod 2$$

– <u>Initially</u> $x = y = 1$.

S_0 : $x = 1 \wedge y = 1$

R : $x' = (x+y) \bmod 2 \wedge y' = y$

function symbol with a "predefined" meaning

The Kripke structure is :

$$M = (S, S_0, R, L)$$

$S = \{(0,0), (0,1), (1,0), (1,1)\}$

$S_0 = \{(1,1)\}$

$R = \{((0,0),(0,0)), ((0,1),(1,1)),$
$((1,0),(1,0)), ((1,1),(0,1))\}$

$L((0,0)) = \{x=0, y=0\}$

$\vdots$

$L((1,1)) = \{x=1, y=1\}$

Single execution

$$(1,1), (0,1), (1,1), (0,1), \ldots$$

Concurrent systems

- A concurrent system is a set of components that execute together (and interact!)

- Possible types of execution

we'll consider both:
  - synchronous (each components makes a step at the same time)
  - asynchronous (interleaving)

- Possible types of communication (interaction)
  - via shared variables → only this in the MC book
  - via message passing (channels)
  - via handshaking (protocol)

Granularity of transitions

- if too coarse, errors may exist in the system but not in the model (they won't be found)

- if too fine, the model creates new states so an error could be found in the new states that does not happen in the system (implementation)

  [due to interleaving]

Example: Consider the following two (versions of a) concurrent systems.

(I)
$\alpha: x := x + y$
$\beta: y := y + x$

Initial state
$x = 1, y = 2$

(II)
α_0: load R_1, x
α_1: add R_1, y
α_2: store x, R_1
β_0: load R_2, y
β_1: add R_2, x

Then $\alpha\beta$ results in a state $x=3, y=5$.

$\beta\alpha$ results in $x=4, y=3$

But in II. we can do $\alpha_0\beta_0\alpha_1\beta_1\alpha_2\beta_2$

resulting in $x=3, y=3$, which is unreachable in I.

If we are interested in a property φ s.t.

$(x=3, y=3) \not\models \varphi$, then

① If the implemented system was I and we model it by II, then we find an error which does not exist

② If the implemented system was II and we model it by I, then an existing error is not found.

Modelling Digital Circuits → synchronous, asynchronous

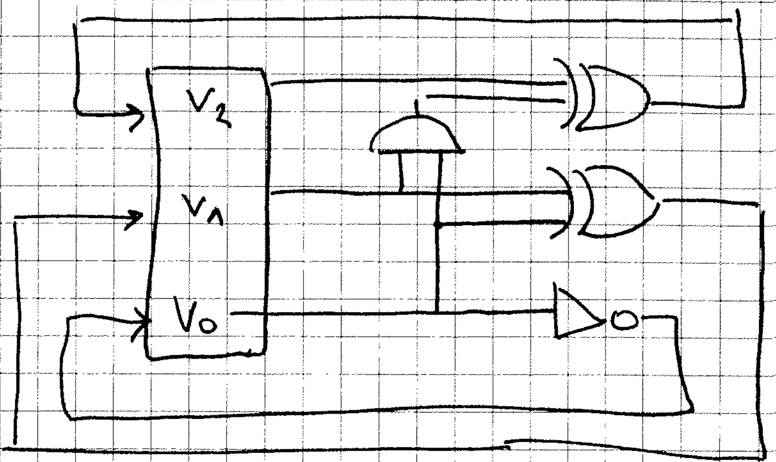
We represent digital circuits by formulas S_0, R (which are now Boolean formulas since all state variables involved are Boolean)

Let V be the set of variables ("state holding" elements of a circuit)

These are in a:

- synchronous circuit: outputs of registers, primary inputs
- asynchronous circuit: all wires

Example : Modulo 8 counter



$V = \{v_0, v_1, v_2\}$

$v_0' = \neg v_0$

$v_1' = v_0 \oplus v_1$

$v_2' = (v_1 \wedge v_2) \oplus v_2$

$\oplus$ - exclusive or

Initial state condition $\quad S_0 : (v_0 = 0) \wedge (v_1 = 0) \wedge (v_2 = 0)$

This is a synchronous circuit
(otherwise would not be a modulo 8 counter)

$R(V, V') : (v_0' \Leftrightarrow \neg v_0) \wedge (v_1' \Leftrightarrow v_0 \oplus v_1) \wedge (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2)$

In general, for any digital circuit

$v_i' = f_i(V) \quad , \quad f_i$ - boolean function

① If the circuit is synchronous, then

$$R(V, V') : \bigwedge_{i=1}^{n} (v_i' \Leftrightarrow f_i(V))$$

② if the circuit is asynchronous, then

$$R(V, V') : \bigvee_{i=1}^{n} (v_i' \Leftrightarrow f_i(V))$$

example : $V = \{v_0, v_1\}$, $v_0' = v_0 \oplus v_1$, $v_1' = v_0 + v_1$.

Initial state is $S_0 = (1,1)$

- in a synchronous semantics : single transition $(1,1) \to (0,0)$, deterministic

- asynchronous semantics : non-deterministic choice. $(1,1) \to (1,0)$

Modelling Programs

AiM: 1st order logic formulas S_0, R corresponding to a concurrent program, (in order to get a Kripke structure)

→ we achieve this in two steps:

(1) label programs

(2) translate labelled programs to formulas via a procedure $\mathcal{C}$.

Labelling procedure

P - program statement, $P^{\mathcal{L}}$ - labelled program statement

If p is an:

① atomic statement (eg. assignment, skip, wait, ...), then $p^{\mathcal{L}} = p$

② sequential composition, $P = P_1 ; P_2$, then
$P^{\mathcal{L}} = P_1^{\mathcal{L}} ; l'' : P_2^{\mathcal{L}}$

③ conditional, $P =$ if b then P_1 else P_2 endif, then
$p^{\mathcal{L}} =$ if b then $l_1 : P_1^{\mathcal{L}}$ else $l_2 : P_2^{\mathcal{L}}$ endif

④ while loop, $P =$ while b do P_1 endwhile, then
$p^{\mathcal{L}} =$ while b do $l_1 : P_1^{\mathcal{L}}$ endwhile

⑤ concurrent program, parallel composition,
$P =$ cobegin $P_1 \parallel P_2 \parallel \dots \parallel P_n$ coend, then
$p^{\mathcal{L}} =$ cobegin $l_1 : P_1^{\mathcal{L}}\, l_1' \parallel l_2 : P_2^{\mathcal{L}}\, l_2' \parallel \dots \parallel l_n : P_n^{\mathcal{L}}\, l_n'$ coend

(here $P_1, \dots, P_n$ are parallel processes)

Hence, we may assume that any program is labelled over a set of variables

entry label m

exit label m'

$V \cup \{pc\}$ — program counter with domain: the set of all labels $\cup \{\perp\}$ — undefined

In a parallel program P,

V_i – variables of P_i

pc_i – program counter of P_i

pc – global program counter

$V = \bigcup_{i=1}^{n} V_i$, it may be that $V_i \cap V_j \neq \emptyset$ then P_i and P_j share the variables in $V_i \cap V_j$

We consider: ASYNCHRONOUS programs with SHARED VARIABL

- we will use a shorthand notation $same(Y) = \bigwedge_{y \in Y} (y' = y)$

for $Y \subseteq V \cup PC$

where $PC = \{pc\} \cup \{pc_i \mid i = 1, .., n\}$

Initial state $\mathcal{S}_0 : \underbrace{pre(V)}_{\text{some precondition (initial condition) on the variables}} \wedge pc = m$

For concurrent programs P

$$\mathcal{S}_0 : pre(V) \wedge pc = m \wedge \left(\bigwedge_{i=1}^{n} pc_i = \perp \right)$$

How do we get the formula $\mathcal{R}$ for the transitions?

For $\mathcal{R}$ we define a translation procedure

$$\mathcal{C}(\ell, P, \ell')$$

(inductively on the structure of P)

If:

① $P = (v := e)$ - assignment

$$\mathcal{C}(\ell, P, \ell') \equiv (pc = \ell \wedge pc' = \ell' \wedge v' = e \wedge same(V \setminus \{v\})$$

② $P = skip$

$$\mathcal{C}(\ell, P, \ell') \equiv (pc = \ell \wedge pc' = \ell' \wedge same(V))$$

③ $P = P_1 ; \ell'' : P_2$ - sequential composition

$$\mathcal{C}(\ell, P, \ell') \equiv \mathcal{C}(\ell, P_1, \ell'') \vee \mathcal{C}(\ell'', P_2, \ell')$$

④ $P =$ if b then $l_1 : P_1$ else $l_2 : P_2$ endif - condition

$$\begin{aligned} \mathcal{C}(\ell, P, \ell') \equiv\ & (pc = \ell \wedge pc' = l_1 \wedge b \wedge same(V)) \\ & \vee (pc = \ell \wedge pc' = l_2 \wedge \neg b \wedge same(V)) \\ & \vee \mathcal{C}(l_1, P_1, \ell') \\ & \vee \mathcal{C}(l_2, P_2, \ell') \end{aligned}$$

⑤ $P =$ while b do $l_1 : P_1$ endwhile - while loop

$$\begin{aligned} \mathcal{C}(\ell, P, \ell') \equiv\ & (pc = \ell \wedge pc' = l_1 \wedge b \wedge same(V)) \\ & \vee (pc = \ell \wedge pc' = \ell' \wedge \neg b \wedge same(V)) \\ & \vee \mathcal{C}(l_1, P_1, \ell) \end{aligned}$$

⑥ $P =$ cobegin $l_1 : P_1\ l_1' \parallel \cdots \parallel l_n : P_n\ l_n'$ coend - parallel composition

$\mathcal{C}(l, P, l') \equiv (pc = l \wedge pc_1' = l_1 \wedge \dots \wedge pc_n' = l_n \wedge pc' = \bot)$

$\vee (pc = \bot \wedge pc_1 = l_1' \wedge \dots \wedge pc_n = l_n' \wedge (\bigwedge_{i=1}^{n} pc_i' = \bot$

$\vee (\bigvee_{i=1}^{n} \mathcal{C}(l_i, P_i, l_i') \wedge \text{same}(V \setminus V_i)$

$\wedge \text{ same}(PC \setminus \{pc_i\})))$

"initialization transition"

"termination transition"

asynchronity

(onlyone component makes a transition at any time)

- Additional statements, useful for describing concurrent systems:

⑦ $P_i = \text{wait}(b)$ — wait until b is true

$\mathcal{C}(l, P_i, l') \equiv (pc_i = l \wedge pc_i' = l \wedge \neg b \wedge \text{same}(V_i))$

$\vee (pc_i = l \wedge pc_i' = l' \wedge b \wedge \text{same}(V_i))$

⑧ $P_i = \text{lock}(v)$, v - Boolean variable, $D = \{0, 1\}$

- Similar to wait($v=0$), but when it gets 0 we change it to 1

$\mathcal{C}(l, P_i, l') \equiv (pc_i = l \wedge pc_i' = l \wedge v = 1 \wedge \text{same}(V_i))$

$\vee (pc_i = l \wedge pc_i' = l' \wedge v = 0 \wedge v' = 1 \wedge$

$\text{same}(V_i \setminus \{v\}))$

⑨ $P_i = \text{unlock}(v)$

$\mathcal{C}(l, P_i, l') \equiv (pc_i = l \wedge pc_i' = l' \wedge v' = 0 \wedge \text{same}(V_i \setminus \{v\}))$

Example: Mutual exclusion

$P = m$: cobegin $P_0 \| P_1$ coend m'

where P_0 = l_0 : while true do

NC_0 : wait (turn = 0);

CR_0 : turn := 1 ;

endwhile;

l_0'

P_1 = l_1 : while true do

NC_1 : wait (turn = 1);

CR_1 : turn := 0 ;

endwhile;

l_1'

Domains of the variables:

pc ... $\{m, m', \perp\}$

pc_i ... $\{l_i, l_i', NC_i, CR_i, \perp\}$

$V = V_0 = V_1 = \{turn\}$ — single shared variable

$PC = \{pc, pc_0, pc_1\}$

S_0 : $(pc = m \wedge pc_0 = \perp \wedge pc_1 = \perp)$

Homework 1: Derive the formula for R (transitions) of this concurrent program; draw the corresponding Kripke structure and check that the program ensures mutual exclusion (a state with $pc_0 = CR_0$ and $pc_1 = CR_1$ is NOT reachable)

NOTE: No initial value of turn is specified, hence the Kripke structure

TEMPORAL logic CTL* → full computational tree logic

- powerful logic
- formulas express properties over states or over paths in a Kripke structure
- the logic has: atomic propositions, Boolean connectives, temporal operators and quantifiers over paths

TEMPORAL operators are:

neXt, Future, Globally, Until, Releases

they express "path properties"

Intuitive meaning:

Xf – f holds in the next state of a given path

Ff – f holds in some state of a given path (at some time in the future)

Gf – f holds in each state of the path

[f U g] – g holds in some state of the path and in all preceeding states f holds.

[f R g] – g holds as long as f did not hold before

CTL* consists of

- Atomic propositions AP
- Boolean connectives: ¬ (not), ∧ (and), ∨ (or)
- Temporal operators: X, F, G, [U], [R]
- Path quantifiers: A, E

intuitively:

Af ... f holds in all paths from a given state

Ef ... f holds in at least one path from a given state.

State formulas (S) and path formulas (P) are simultaneously inductively defined:

$$S ::= true \mid false \mid p \in AP \mid \neg S \mid S \wedge S \mid S \vee S \mid AP \mid EP$$
$$P ::= S \mid \neg P \mid P \wedge P \mid P \vee P \mid XP \mid GP \mid FP \mid [PUP] \mid [PRP]$$

Hence, state formulas are

- constants (true/false) or atomic propositions
- Boolean combinations of state formulas
- quantified path formulas; and

Path formulas are

- state formulas
- Boolean combinations of path formulas
- temporal combinations of path formulas.

The <u>semantics</u> of CTL^* formulas is given relative to a fixed Kripke structure $M = (S, S_0, R, L)$

<u>Semantics of a state formula, given M</u>, $s \in S$

$s \models true$

$s \not\models false$

$s \models p \in AP$ iff $p \in L(s)$

$s \models \neg f$ iff $s \not\models f$

$s \models f \wedge g$ iff $s \models f$ and $s \models g$

$s \models f \vee g$ iff $s \models f$ or $s \models g$

$s \models Ef$ iff there exists a path $\pi \in paths(s)$ s.t. $\pi \models f$

$s \models Af$ iff for all $\pi \in paths(s)$, $\pi \models f$

Here, paths(s) denotes the set of paths starting in s.
Moreover, for a path $\pi = s_0 s_1 s_2 \ldots$ we write

$\pi(i) = s_i$, so $\pi(0)$ is the first state of π and

$\text{paths}(s) = \{\pi \mid \pi(0) = s\}$

In addition, let π^i denote the suffix of π starting from $\pi(i)$, i.e. $\pi^0 = \pi$, $\pi^1 = s_1 s_2, \ldots$, $\pi_2 = s_2 s_3 \ldots$

Semantics of a path formula, given M, π-path

$\pi \models f$ iff $\pi(0) \models f$, for a state formula f

$\pi \models \neg f$ iff $\pi \not\models f$

$\pi \models f \wedge g$ iff $\pi \models f$ and $\pi \models g$

$\pi \models X f$ iff $\pi^1 \models f$

$\pi \models F f$ iff $(\exists i \geq 0)\ \pi^i \models f$

$\pi \models G f$ iff $(\forall i \geq 0)\ \pi^i \models f$

$\pi \models [f U g]$ iff $(\exists i \geq 0)(\pi^i \models g \wedge (\forall j < i)\ \pi^j \models f)$

$\pi \models [f R g]$ iff $(\forall j \geq 0)((\forall i < j)\ \pi^i \not\models f \Rightarrow \pi^j \models g)$

We define equivalence of CTL* formulas, notation $\equiv$, in the usual way, as:

$f \equiv g$ iff for any Kripke Structure M and any state s in M $(M, s \models f \Leftrightarrow M, s \models g)$

. According to the semantics, we can derive several equivalences (dualities) ⟶

$\neg(Gf) \equiv F(\neg f)$

$\neg\neg f \equiv f$

$\neg(f \wedge g) \equiv \neg f \vee \neg g$

$\neg(Af) \equiv E(\neg f)$

$\neg(Xf) \equiv X(\neg f)$

$Ff \equiv [\text{true}\ U f]$

$\neg[f\ R g] \equiv [(\neg f)\ U\ (\neg g)]$

Proof (of the last one)

$\pi \models [(\neg f)\ U\ (\neg g)]$ iff

$(\exists i \geq 0)(\pi^i \models \neg g) \wedge (\forall j < i)(\pi^j \models \neg f)$

iff $(\exists i \geq 0)(\pi^i \not\models g) \wedge (\forall j < i)(\pi^j \not\models f)$

Therefore

$\pi \models \neg[(\neg f)\ U\ (\neg g)]$ iff

$(\forall i \geq 0)(\pi^i \models g) \vee \neg(\forall j < i)(\pi^j \not\models f)$

iff $(\forall i \geq 0)((\forall j < i)(\pi^j \not\models f) \Rightarrow \pi^i \models g)$

iff $\pi \models [f\ R\ g]$. ∎

Another equivalence; useful for better understanding of releases (R) is:

$[f R g] \equiv [g\ U\ (f \wedge g)] \vee Gg$

As a result: for CTL* it is enough to have $\neg, \vee, E, X, [U]$

Two important sub-logics of CTL* are LTL and CTL

(1) LTL – linear temporal logic

- checks temporal operations along a single path

→

- good sides:
  - easy to construct counter examples
  - nice automata-based MC algorithm
- typical tool: SPIN

② CTL – Computational tree logic
- checks temporal operators over computation trees [branching-time logic]
- temporal operators must be preceded by quantifiers
- typical tool: nuSMV

LTL state formulas (S) and path formulas (P) are defined by:

$$S ::= AP$$

$$P ::= true \mid false \mid p \in AP \mid \neg P \mid P \vee P \mid XP \mid FP \mid GP \mid [P \, U \, P] \mid [P \, R \, P]$$

Hence, the only state formulas are "all"-quantified path formulas, and path formulas are

- constants and atomic propositions
- Boolean combinations of path formulas
- temporal combinations of path formulas

(state formulas are <u>not</u> path formulas)
(no nested quantifiers)

<u>Examples:</u> LTL formulas are

AFGp (for all paths starting in the state, after a finite number of states p holds in every state)

$A(\neg(GFp) \vee Fq)$ (on each path, if p holds infinitely often, then eventually q holds)

Not in LTL (syntactically) – due to nested quantifiers

$AFAGp$ – on all paths a state is reachable from where p holds on all sub-paths.

$AGEFp$ – from any reachable state, a state satisfying p is reachable.

But are they expressible in LTL?

We will show that $AFAGp$ is not expressible in LTL. Hence, LTL is a proper sublogic of CTL*.

i.e. we will show that there is no LTL formula φ s.t. $\varphi \equiv AFAGp$.

Theorem: [Clarke & Draghicescu]

A CTL* formula φ is expressible in LTL iff

$$\varphi \equiv A\, d(\varphi)$$

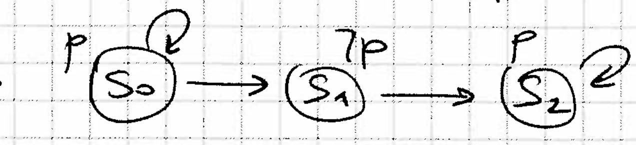
where $d(\varphi)$ is the CTL* formula obtained from φ by deleting all occurrences of path quantifiers. ∎

As a corollary: $AFAGp$ is expressible in LTL iff

$$AFAGp \equiv AFGp.$$

We will show that $AFAGp \not\equiv AFGp$ and hence $AFAGp$ is not expressible in LTL

For this, we construct a Kripke structure

M : 

Then, $M, s_0 \models AFGp$, but $M, s_0 \not\models AFAGp$

since:

Any path π from s_0 has the shape

$$\pi = s_0^{k} s_1 s_2^{\infty} \quad \text{or} \quad \pi = s_0^{\infty}, \quad \text{for } k \in \mathbb{N}.$$

If $\pi = s_0^{\infty}$, then $\pi(i) \models p$, for all i,

so

$$\pi \models Gp \quad \text{i.e.} \quad \pi^{0} \models Gp$$

and therefore $\pi \models FGp$.

If $\pi = s_0^{k} s_1 s_2^{\infty}$, then $(\forall i \geq k+1)\ \pi(i) \models p$

i.e. $\pi^{k+1} \models Gp$

and therefore $\pi \models FGp$.

So, we have shown that $M, s_0 \models AFGp$.

However, consider $\hat{\pi} = s_0^{\infty}$; we will show that

$\hat{\pi} \not\models FAGP$; and so $s_0 \not\models AFAGp$.

for this, let $i \geq 0$ be any number, we show that

$\hat{\pi}(i) \not\models AGp$, and so $\hat{\pi}^{i} \not\models AGp$ i.e.

$$\neg(\exists i \geq 0)\ \hat{\pi}^{i} \models AGp \text{ i.e.}$$

$$\hat{\pi} \not\models FAGp.$$

for this we first note that $\hat{\pi}(i) = s_0$

and then we consider the path $\pi = s_0 s_1 s_2^{\infty}$.

for this path we have that

$$\pi \not\models Gp \quad \text{since } \pi^1 = s_1 s_2^\infty \not\models p$$

$$\text{since } \pi^1(0) = s_1 \not\models p.$$

Hence, $s_0 \not\models AGp$ and so indeed $\hat{\pi}(i) \not\models AGp$ which completes the proof. ■

CTL state formulas ($\mathcal{S}$) and path formulas ($\mathcal{P}$) are defined by

$$\mathcal{S} ::= \text{true} \mid \text{false} \mid p \in AP \mid \neg \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid E\mathcal{P} \mid A\mathcal{P}$$

$$\mathcal{P} ::= X\mathcal{S} \mid F\mathcal{S} \mid G\mathcal{S} \mid [\mathcal{S}\, U\, \mathcal{S}] \mid [\mathcal{S}\, R\, \mathcal{S}]$$

Hence, state formulas are (as before)

- constants or atomic propositions
- Boolean combinations of state formulas
- quantified path formulas, whereas

path formulas are only temporal combinations of state formulas.

(no direct nesting of temporal operators)

Example: CTL formulas are

$AGEFp$ (from every reachable state p is reachable)

$E[p\, U\, (EXq)]$

not in CTL (syntactically) are

$AFGp$

$AXXp$

$E[p\, U\, Xq]$

Homework: ① $AXXp \overset{?}{\equiv} AXAXp$

(There are also results by Clarke & Draghicescu on when a CTL* f-la is expressible in CTL)

Alternative definition of CTL

CTL has only state formulae, with the following ten temporal combinators:

- AX, EX – for all/some next state
- AF, EF – inevitably, potentially
- AG, EG – invariantly (always), potentially always
- A[U], E[U] – for all/some paths until
- A[R], E[R] – for all/some paths releases

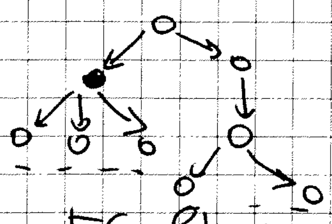
Hence

$$S ::= true \mid false \mid p \in AP \mid \neg S \mid S \vee S \mid AX\,S \mid AF\,S \mid EX\,S \mid EF\,S \mid AG\,S \mid EG\,S \mid A[S \, U \, S] \mid E[S \, U \, S] \mid A[S \, R \, S] \mid E[S \, R \, S]$$

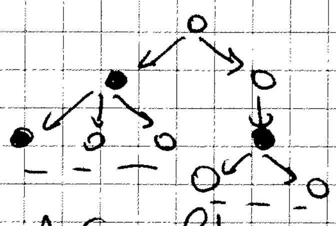
Most widely used: EF, AF, EG, AG

Illustrated below (on the computation tree of a state)

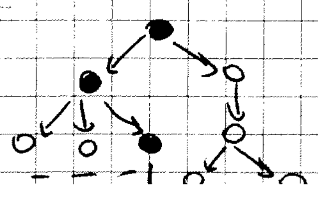
EF black



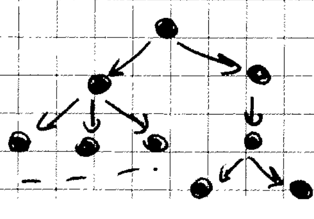
AF black



EG black



AG black



For CTL it is enough to have

$EX, EG, E[\,U\,]$ as temporal operators:

$$AXf \equiv \neg EX(\neg f)$$
$$EFf \equiv E[\text{true}\ U\ f]$$
$$AFf \equiv \neg EG(\neg f)$$
$$AGf \equiv \neg EF(\neg f)$$
$$A[fRg] \equiv \neg E[(\neg f)\ U\ (\neg g)]$$
$$E[fRg] \equiv \neg A[(\neg f)\ U\ (\neg g)]$$

In order to remove $A[\,U\,]$ we use the following

① $[f\ R\ g] \equiv [g\ U\ (f \wedge g)] \vee Gg$

② $A[f\ U\ g] \equiv \neg E[(\neg f)\ R\ (\neg g)]$

③ $E(f \vee g) \equiv Ef \vee Eg$

So we get

$$\begin{aligned} A[f\,U\,g] &\equiv \neg E[(\neg f) R (\neg g)] \\ &\equiv \neg E\big([\neg g\ U\ (\neg f \wedge \neg g)] \vee G(\neg g)\big) \\ &\equiv \neg E[\neg g\ U \neg(f \vee g)] \wedge \neg EG(\neg g) \end{aligned}$$

Another sublogic of CTL^* (CTL) is $ACTL^*$ (ACTL)

in which only A-quantifiers are allowed

[in order not to get some E-quantifiers "on back door" negations are only allowed on atomic propositions]

<u>To conclude</u>, here is how the logics compare

$$\begin{tikzcd}
 & CTL^* & \\
ACTL^* \arrow[ur] & & CTL \arrow[ul] \\
LTL \arrow[u] & & ACTL \arrow[u]
\end{tikzcd}$$

LTL and CTL
as well as
LTL and ACTL
are not comparable :

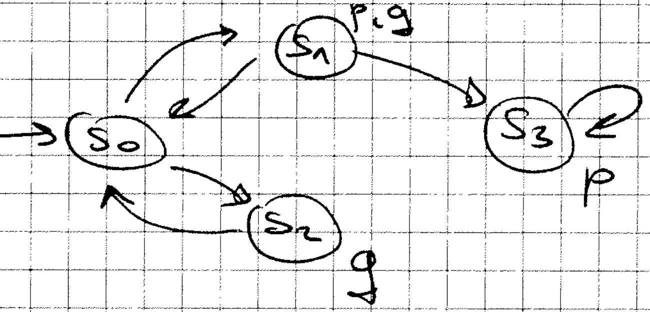
$AFAGp \in ACTL \subseteq CTL$

but $AFAGp \notin LTL$
not expressible
(we have shown)

Also $AFGp \in LTL$, but
$AFGp \notin CTL$
(we have not shown, but can be shown)
and hence also $AFGp \notin ACTL$.

<u>Excercise</u> — Homework ② except for (a)

Consider



and the CTL^* formulas:

(a) $E[g R p]$ (b) $EFGp$ (c) $AGFp$ (d) $AGEFp$

(e) $AGF(p \wedge Xg)$ (f) $AG(\neg g \vee Fp)$ (g) $A(Gp \vee Fg)$

<u>For each formula</u>:

- indicate whether it is in LTL and/or CTL
- determine in which states of the above Kripke structure it holds

In class, we solve (a)

$E[gRp]$ is not in LTL since "E" - quantifier is used

it is in CTL.

Now we have

1. $s_0 \not\models E[gRp]$
2. $s_1 \models E[gRp]$
3. $s_2 \not\models E[gRp]$
4. $s_3 \models E[gRp]$

since 1. $E[gRp] \equiv \neg A[\neg g \cup \neg p]$

and $s_0 \not\models E[gRp]$ iff $s_0 \models A[\neg g \cup \neg p]$.

Now $s_0 \models \neg p$, so for any path π with $\pi(0) = s_0$

we have $\pi \models \neg p$ i.e. $\pi^0 \models \neg p$ and hence

also $\pi \models [\neg g \cup \neg p]$

Therefore $s_0 \models A[\neg g \cup \neg p]$

2. $s_1 \models p \wedge g$

so for any π with $\pi(0) = s_1$ we have $\pi \models p \wedge g$

and also therefore $\pi \models [\varphi \cup (p \wedge g)]$ for any φ

in particular $\pi \models [g \cup (p \wedge g)]$ and so

$$\pi \models [g \cup (p \wedge g)] \vee Gp \equiv [gRp]$$

Hence $s_1 \models A[gRp]$ which implies $s_1 \models E[gRp]$.

3. Let π be any path from S_2. Then

$$\pi = S_2 S_0 \hat{\pi}$$

Now $S_0 \models \neg p$, and $S_2 \models \neg g$

so $\pi \models [\neg g \cup \neg p]$

so $S_2 \models A[\neg g \cup \neg p]$

which is equivalent to $S_2 \not\models E[g R p]$

4. $\pi = S_3^{\infty}$ is the only path from S_3.

Now here since $S_3 \models p$, we have

$$\pi \models Gp$$

and so $\pi \models [p \cup (g \wedge p)] \vee Gp \equiv [g R p]$

which shows that

$$S_3 \models A\, [g R p]$$

and as a consequence also $S_3 \models E[g R p]$.

<u>DONE</u>. (it's good that they all have one full proper example)