System specification with temporal logic

[Automated Reasoning, 2015/2016 1b — Lecture 3]

Doina Bucur

d.bucur@rug.nl

Nov 2015

In this lecture...

Specifications and Linear Temporal Logic (LTL)

- Temporal logics
- Defining LTL operators and formulas
- Checking LTL formulas inductively
- Writing LTL formulas. Dualities, patterns and scopes
- Safety, liveness, fairness
- Alternative formalism to temporal logics
- Writing specifications in PROMELA

Historical and bibliographical notes

Assignment

We learn the most widespread language to write system specifications, Linear Temporal Logic (LTL); we show induction rules to verify a Kripke structure against a given LTL formula. Finally, we categorize specifications.

Basic system specifications

You have already (informally) seen simple types of properties. I summarize them here more formally:

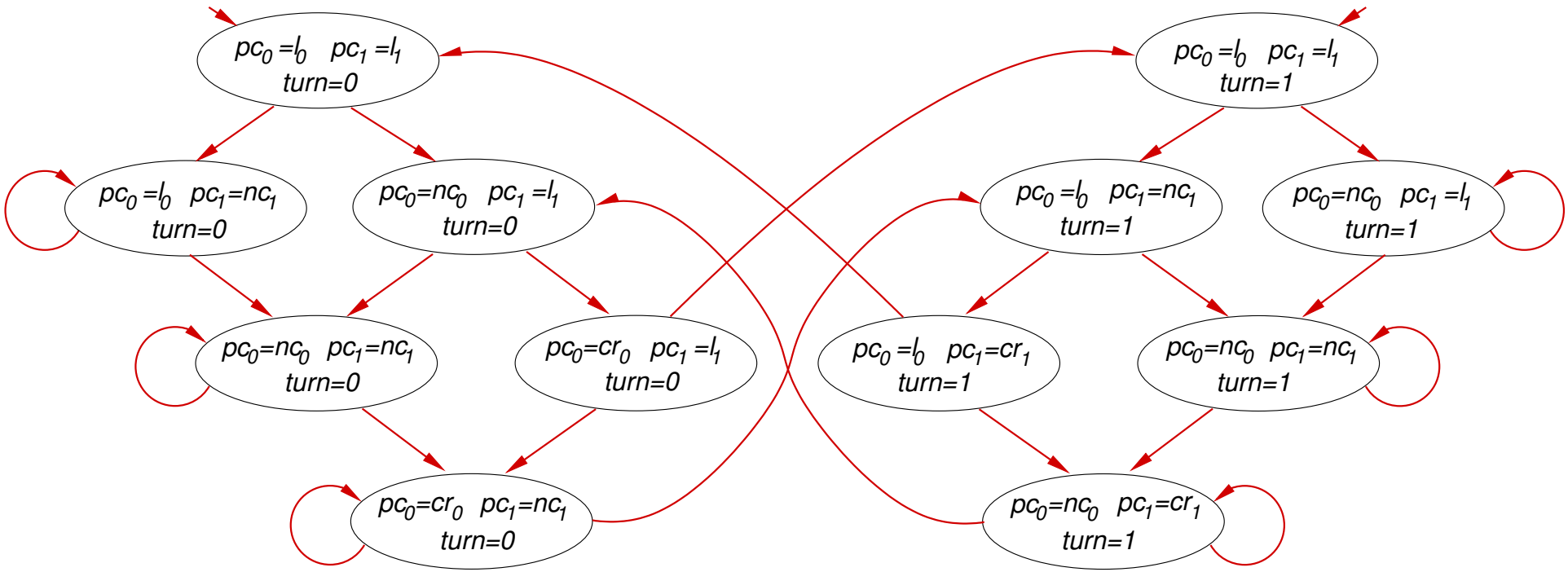
Deadlock : A system model **can** have a deadlock when it has at least one state without outgoing transitions, and the state is not a valid *end* state.

Termination : A system model **can** terminate if it can deadlock on a valid *end* state. Which state is a valid end state depends on the original program; this often corresponds to the state of memory after the *last* statement in the program. A model **may not** terminate if it has at least one cycle.

Boolean propositions on the values of memory: `x <= 8`. You saw this type written as `assert(x <= 8);` in Assignment 1, verified by Spin. **Assertions** are the most basic (and used) software specifications.

In what follows, you learn a language to write more complex properties, which include a basic notion of **time**.

Remember: system models as labelled automata



Paths in Kripke structures

- A **path** π from state s in a Kripke structure is an (in)finite sequence of states

$$\pi = s_0 s_1 s_2 \ldots$$

  so that $s_0 = s$ and $(s_i, s_{i+1}) \in T, \forall i \geq 0$.
- π^i is the **suffix** of π starting at s_i.

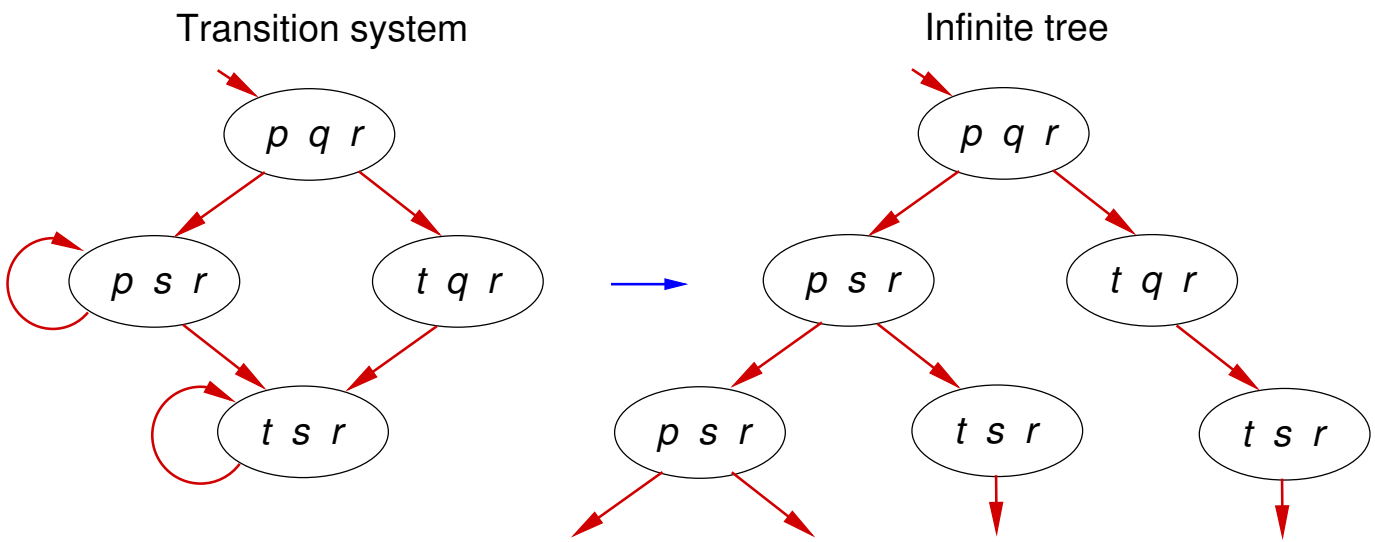
- Clearly, **infinite paths** may exist in finite-state machines.

Temporal logics

Need: specify temporal properties of (in)finite system executions.

System executions are obtained as follows:

- Take a Kripke structure
- ...and *virtually* unwind any loops (from any initial state).
- The result is an infinite tree of all possible executions paths.
- Temporal-logic formulas express properties for such execution paths.



Linear-Time Logic (LTL)

A simple, useful temporal logic is LTL: it writes properties for individual execution paths[1].

LTL path formulas p may be built like this:

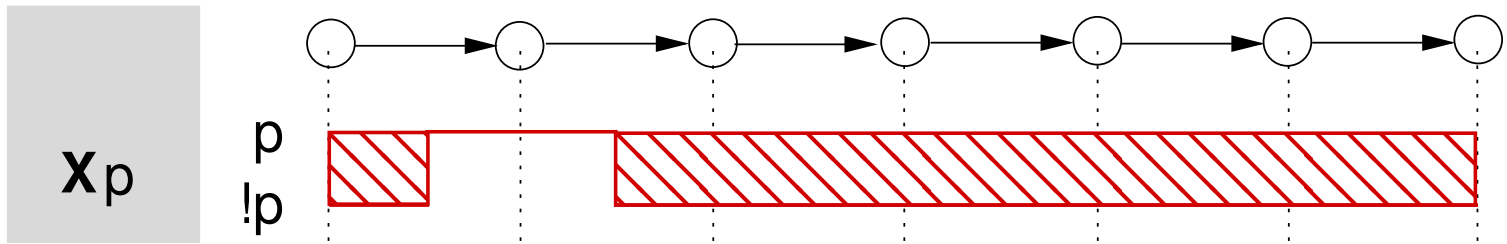
- $p \equiv (\texttt{turn} = 0)$
  The simplest formula: a boolean proposition (e.g., any $p \in \text{AP}$, including the general 'true' and 'false').
  Meaning: On the first state of this path, p should hold.
- $p \equiv (q \wedge \neg r)$
  Standard boolean **logical operators** compose LTL formulas out of other LTL formulas.
- $p \equiv (\mathbf{G}q)$
  Here we add **temporal operators**.
  Meaning: Globally on this path (i.e., in all states), q should hold.

[1] The more general variant is Computation Tree Logic (CTL*): its formulas can write properties for more paths at a time. For software checking, LTL is sufficient.

LTL temporal operators (1)

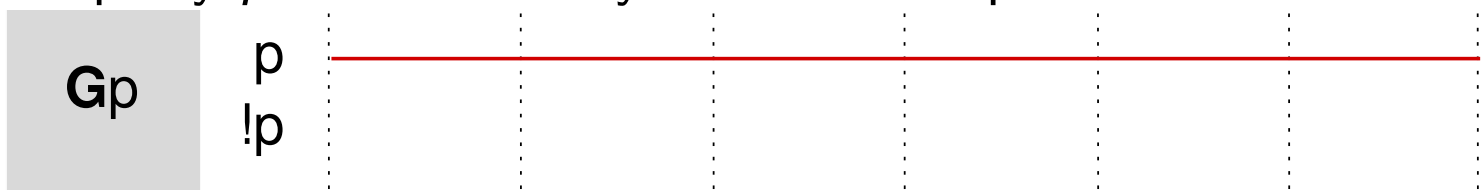
X*p* **"next time"**.

Property *p* holds at the second state on the path.



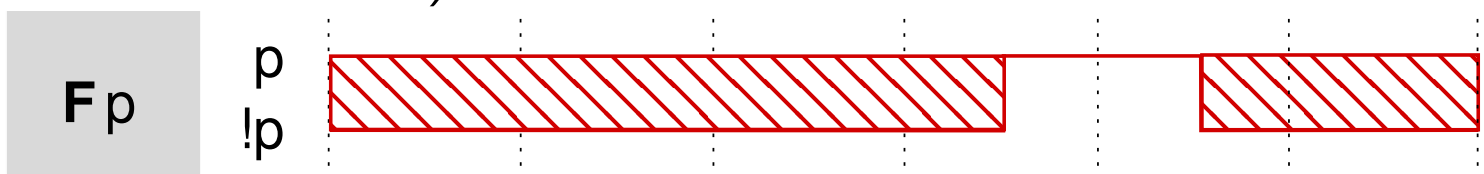
G*p* **"globally"** or **"always"**, also denoted $\Box p$.

Property *p* holds at every state on the path.



F*p* **"in the future"** or **"eventually"**, also denoted $\Diamond p$.

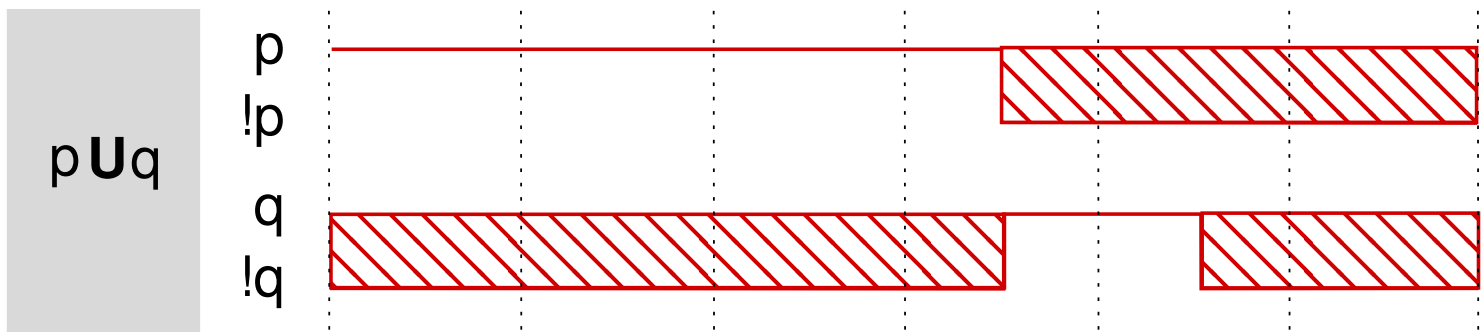
Property *p* holds at some state on the path (after a finite number of states).



LTL temporal operators (2)

$p\mathbf{U}q$ **"until"**.

This holds if eventually q holds,
and at every state before that p holds.

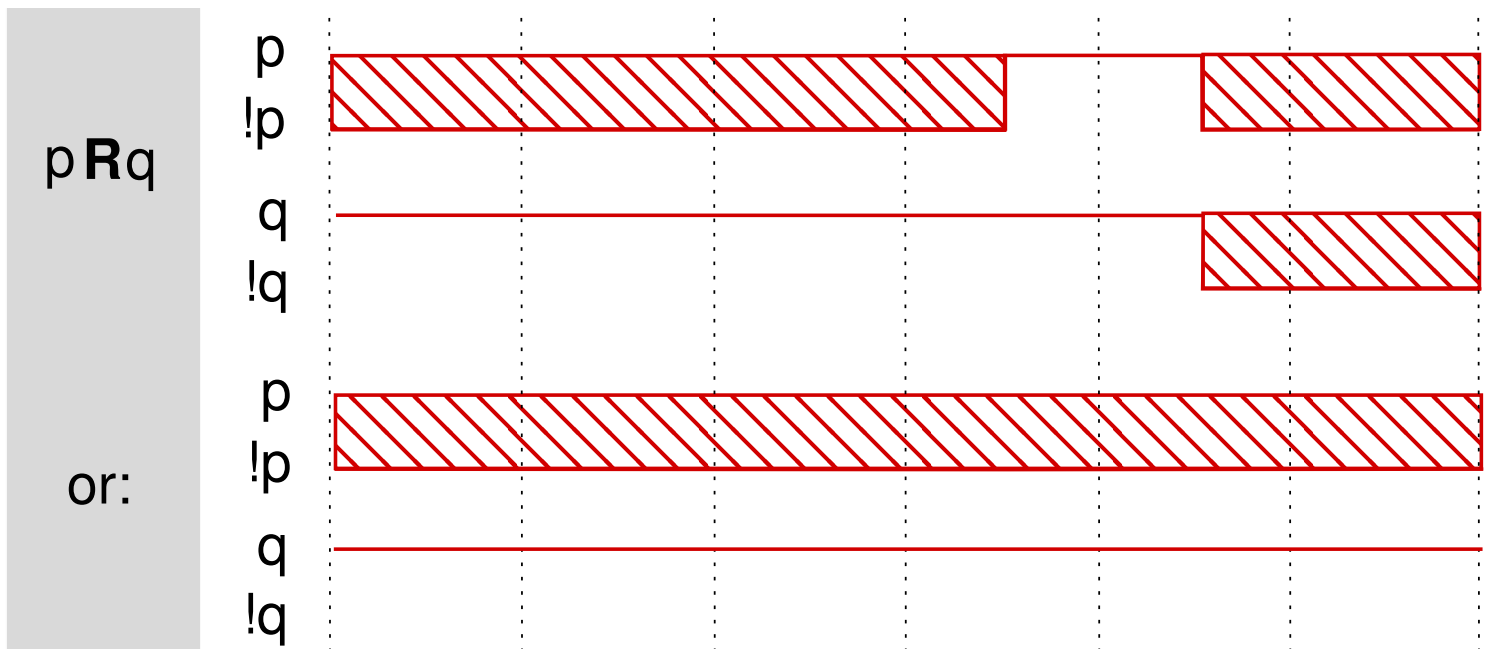


LTL temporal operators (3)

$p\mathbf{R}q$ **"release"**.

Dual of "until".

This holds if eventually p holds, and at every state before that (also including that state) q holds.
Alternatively, this holds if q holds all the time.



LTL formulas

Definition (LTL formula)

1. If $p \in AP$, then p is a **state formula**.
2. If f_1 and f_2 are state formulas, then $\neg f_1, f_1 \vee f_2, f_1 \wedge f_2$ are state formulas.
3. A state formula is also a **path formula**.
4. If g_1 and g_2 are path formulas, then $\neg g_1, g_1 \vee g_2, g_1 \wedge g_2$, $\mathbf{X} g_1, \mathbf{F} g_1, \mathbf{G} g_1, g_1 \mathbf{U} g_2, g_1 \mathbf{R} g_2$ are path formulas.

Note: Logical implication and equivalence are the usual shorthands:
$p \rightarrow q$ is $(\neg p) \vee q$, and
$p \leftrightarrow q$ is $(p \rightarrow q) \wedge (q \rightarrow p)$.

Induction rules to prove LTL: state formulas

Say M is a Kripke structure, f_i state formulas and g_i path formulas. We write $s \models f_1$ to say that f_1 holds on state s in M, and $\pi \models g_1$ to say that g_1 holds along path π in M.

(Reminder: you already know, informally, how to check $s \models f_1$, i.e., an **assertion** over a **state** s: you check whether that state's label violates f_1).

You can imagine that $s \models$ in LTL is formalized simply:

$$
\begin{array}{lcl}
s \models p & \Leftrightarrow & p \in L(s) \\
s \models \neg f_1 & \Leftrightarrow & s \not\models f_1 \\
s \models f_1 \wedge f_2 & \Leftrightarrow & s \models f_1 \text{ and } s \models f_2 \\
s \models f_1 \vee f_2 & \Leftrightarrow & s \models f_1 \text{ or } s \models f_2
\end{array}
$$

Induction rules to prove LTL: path formulas

$\pi \models$ is defined inductively:

$$
\begin{array}{lcl}
\pi \models f_1 & \Leftrightarrow & \pi \text{ starts in } s \text{ and } s \models f_1 \\
\pi \models \neg g_1 & \Leftrightarrow & \pi \not\models g_1 \\
\pi \models g_1 \wedge g_2 & \Leftrightarrow & \pi \models g_1 \text{ and } \pi \models g_2 \\
\pi \models g_1 \vee g_2 & \Leftrightarrow & \pi \models g_1 \text{ or } \pi \models g_2
\end{array}
$$

$$
\begin{array}{lcl}
\pi \models \mathbf{X} g_1 & \Leftrightarrow & \pi^1 \models g_1 \\
\pi \models \mathbf{F} g_1 & \Leftrightarrow & \exists k \geq 0 . \pi^k \models g_1 \\
\pi \models \mathbf{G} g_1 & \Leftrightarrow & \forall k \geq 0 . \pi^k \models g_1
\end{array}
$$

$$
\begin{array}{lcl}
\pi \models g_1 \mathbf{U} g_2 & \Leftrightarrow & \exists k \geq 0 . \pi^k \models g_2 \text{ and } \forall i, 0 \leq i < k . \pi^i \models g_1 \\
\pi \models g_2 \mathbf{R} g_1 & \Leftrightarrow & \forall j \geq 0 . \text{ if } \forall i \leq j . \pi^i \models g_1 \text{ then } \pi^j \models g_2
\end{array}
$$

Some intuition

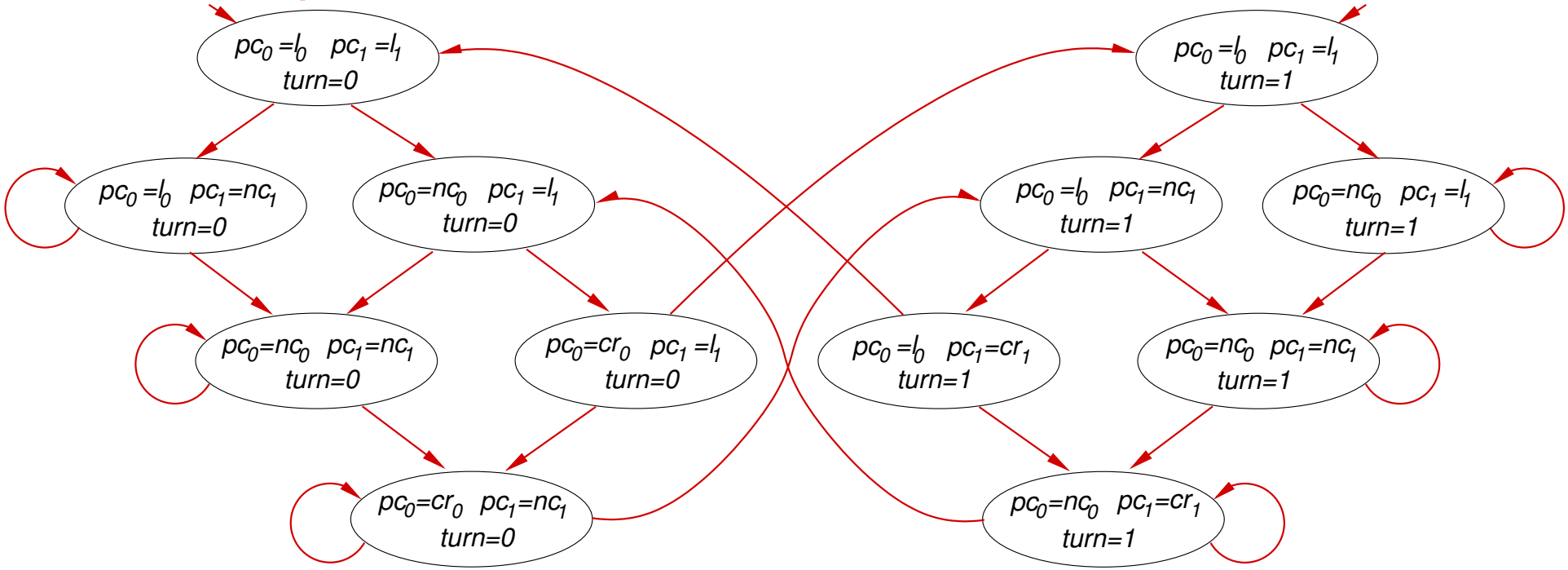
Question: An unwound Kripke structure M has any number of different executions. How can a LTL formula, say, $\mathbf{F}(pc_0 = cr_0 \wedge pc_1 = cr_1)$ (which describes one path) specify the entire M?

Answer: It will if we add a **path quantifier** to f, and write $\mathbf{A}f$: "For all paths in M, f".

A ("for all paths") and **E** ("for some path") are path quantifiers, part of the Computation Tree Logic (CTL*), a superset of LTL. They act like the logical quantifiers $\forall x.P$ and $\exists x.P$, but on paths instead of on variables. (One **A** or **E** is sufficient to write any formula.)

$\mathbf{A}f$ is a state formula.

Some examples



With the initial state where `turn=0`, are the following properties true for all paths **A**?

$$\mathbf{XX}(turn = 0)$$
$$\mathbf{G}\neg((pc_0 = cr_0) \wedge (pc_1 = cr_1))$$
$$(turn = 0)\mathbf{U}(turn = 1)$$
$$\mathbf{GF}(pc_1 = cr_1)$$

Operator minimality

The set of temporal operators in LTL is not minimal:

- $\mathbf{F}p$ is shorthand for: true $\mathbf{U}$ p
  ('true' is a state label which is true in any state).
- $\mathbf{G}p$ is shorthand for: p $\mathbf{W}$ false[2]
  ('false' is a state label which is false in any state);

[2] This new operator **W** is called a 'Weak until', and is slightly different than Until; can you tell how?

Some interesting dualities

You can prove that:

$$
\begin{aligned}
\neg \mathbf{G} p &\Leftrightarrow \mathbf{F} \neg p \\
\mathbf{G} p &\Leftrightarrow \mathbf{G}\mathbf{G} p \\
\mathbf{F} p &\Leftrightarrow \mathbf{F}\mathbf{F} p \\
\mathbf{G}(p \wedge q) &\Leftrightarrow \mathbf{G} p \wedge \mathbf{G} q \\
\mathbf{F}(p \vee q) &\Leftrightarrow \mathbf{F} p \vee \mathbf{F} q \\
\\
\mathbf{G}\mathbf{F}(p \vee q) &\Leftrightarrow \mathbf{G}\mathbf{F} p \vee \mathbf{G}\mathbf{F} q \\
p \mathbf{U} q &\Leftrightarrow p \mathbf{U} (p \mathbf{U} q) \\
p \mathbf{U} q &\Leftrightarrow (p \mathbf{U} q) \mathbf{U} q \\
\neg (p \mathbf{U} q) &\Leftrightarrow (\neg q) \mathbf{U} (\neg p \wedge \neg q) \\
\\
\mathbf{A} p &\Leftrightarrow \neg \mathbf{E} \neg p
\end{aligned}
$$

Writing LTL from natural-language specifications...

...can be daunting.

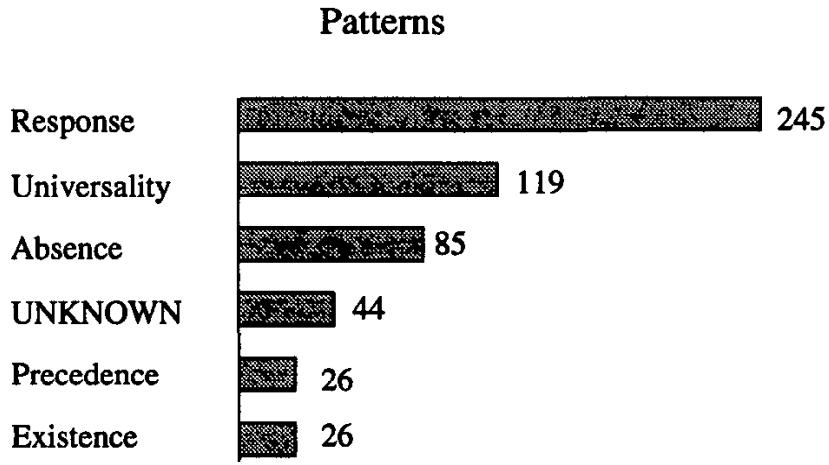
The specification "Between the time an elevator is called at a floor and the time it opens its doors at that floor, the elevator can arrive at that floor at most twice"[3] is in LTL:

$$\begin{aligned}
&\Box((\text{call} \wedge \Diamond \text{open}) \rightarrow \\
&\quad ((\neg \text{atfloor} \wedge \neg \text{open}) \; \mathcal{U} \\
&\qquad (\text{open} \vee ((\text{atfloor} \wedge \neg \text{open}) \; \mathcal{U} \\
&\qquad\quad (\text{open} \vee ((\neg \text{atfloor} \wedge \neg \text{open}) \; \mathcal{U} \\
&\qquad\qquad (\text{open} \vee ((\text{atfloor} \wedge \neg \text{open}) \; \mathcal{U} \\
&\qquad\qquad\quad (\text{open} \vee (\neg \text{atfloor} \; \mathcal{U} \; \text{open})))))))))
\end{aligned}$$

[3] From: M. B. Dwyer, G. S. Avrunin, J. C. Corbett. *Patterns in property specifications for finite-state verification.* In Proc. Int. Conf. on Software Engineering (ICSE '99). Check this paper out: it's fun. The plots on the next few slides originate here. `http://patterns.projects.cis.ksu.edu/documentation/patterns.shtml` (same authors) is also more informative than this overview.

Patterns and scopes to the rescue

Specifications naturally fall into patterns; on the right, a survey of 500+ collected specifications:



Formula	(with p a state formula)	**Pattern names**
$\mathbf{G}p$	p is always true	**universality/absence/invariant**
$\mathbf{F}p$	p is eventually true	**existence/guarantee**
$\mathbf{G}(p \rightarrow \mathbf{F}q)$	p implies eventually q	**response**
$\neg q \mathbf{W} p$	p precedes q	**precedence**
$\mathbf{GF}p$	always eventually p	**recurrence (progress)**
$\mathbf{FG}p$	eventually always p	**stability (non-progress)**
$\mathbf{F}p \rightarrow \mathbf{F}q$	eventually p implies eventually q	**correlation**

Frequently used scopes

For more complex properties, place the previous patterns into scopes.

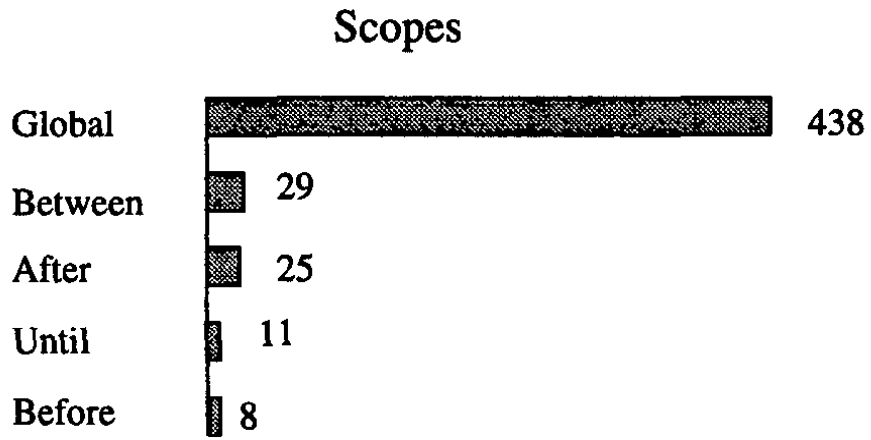
Pattern P happens:

globally:
$\mathbf{G}P$

before Q:
$\mathbf{F}Q \rightarrow (P\mathbf{U}Q)$

after Q:
$\mathbf{G}(Q \rightarrow \mathbf{G}P)$

between Q and R:
$\mathbf{G}((Q \wedge \neg R \wedge \mathbf{F}R) \rightarrow (P\mathbf{U}R))$



Safety specifications

"Nothing bad should happen".

Examples:

- Mutual exclusion, e.g. $\mathbf{G}\neg(pc_0 = cr_0 \wedge pc_1 = cr_1)$;
- Deadlock freedom;
- (Generally, any **invariant** $\mathbf{G}p$);
- But also: any property for which an error trace contains a finite bad prefix, such as **precedence**. I.e., safety properties are **violated in finite time**.

Liveness specifications

"Something good will eventually happen".

Examples:

- A **guarantee** $\mathbf{F}p$ or **progress** $\mathbf{GF}p$ capture liveness;
- Starvation freedom (a form of fairness): each waiting process will eventually enter its critical section;
- Assume a system with infinite executions. Liveness properties are **violated in infinite time**.
- Safety and liveness properties are disjoint.
- Any LTL formula can be rewritten as a conjunction of safety and liveness properties.

Fairness

A **fair** path is, intuitively, one on which certain states do appear regularly, instead of being ignored forever by the program scheduler. Can be expressed with a **progress** formula.

An example: for a communication channel, a fairness constraint is a state in which a message is received at an end of the channel, if it was sent at the other.

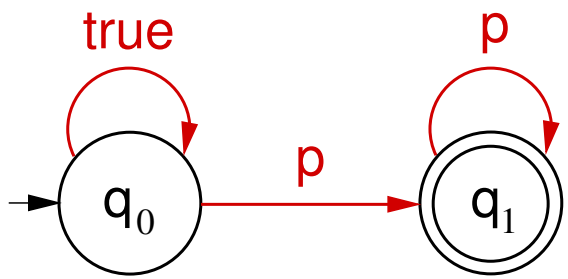
Definition (Fair Kripke structure)

A fair Kripke structure is $M = (S, S_0, T, L, F)$, where all is as before besides the new $F \subseteq 2^S$.
F is the set of fairness constraints[4].
π is called a fair path if every state from F appears in π infinitely often.

[4] Fairness constraints are often called generalized Büchi acceptance conditions. We see about Büchi automata next time.

Alternative: Automata as specifications

You can also express these (and more) specifications as automata[5].



automaton for **FG** p

However, the complexity of expressing a specification with automata is higher than with temporal logics. The same goes for operating on a specification (think how hard it is to complement an automaton). This is why temporal logics are more widely used as specification formalisms.

Nevertheless, automata are one of the main implementation techniques for temporal-logic model checking. We see this next time.

[5] Also, with other logics, calculi, regular expressions.

PROMELA constructs

PROMELA also recently expresses LTL (latest version); in general, you can specify:

Assertions `assert`(executable state formula p)
Added at a particular program state; if state is reachable, p should not evaluate to false. The most commonly used specifications in practice, by far.

Labels for end states and progress states;

LTL formulas with the `ltl` keyword;
they can reference the program counter of a process with atomic propositions such as `process@line`.

Never claims i.e., automata; SPIN translates LTL formulas into `never` claims internally anyway, so we'll prefer to write the much more concise LTL.

Traces express allowed sequences of channel operations.

Historical notes

Amir Pnueli first proposed the use of *temporal logics* in the analysis of distributed systems in 1977; it took another decade for this to become mainstream.

Since then, various extensions of LTL have been found of use. A flavour of LTL with *past operators* can write more succinct formulas than standard LTL. LTL is the basis for the *Property Specification Language*[6] and the *Process Specification Language*[7], standardized industrial specification languages for various types of computer systems.

The terms *safety* and *liveness* were first formalized by Leslie Lamport (yes, the LaTeXcreator, Turing Award 2013) in 1982[8].

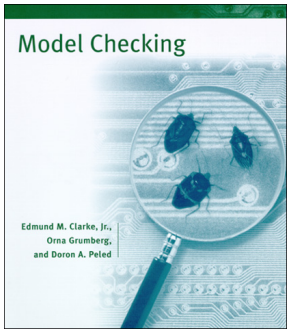
---

[6] `https://en.wikipedia.org/wiki/Property_Specification_Language`

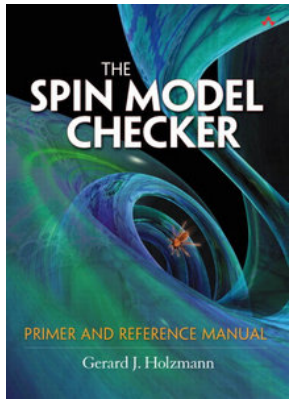
[7] `http://en.wikipedia.org/wiki/ISO_18629`

[8] See "Proving Liveness Properties of Concurrent Programs", ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982, Pages 155-495.

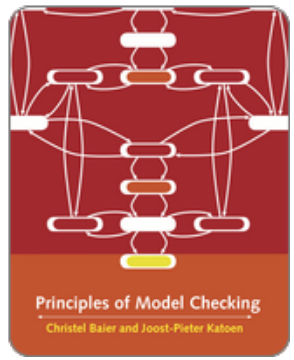
Bibliographical notes



The content in this lecture covers Ch. 3, *Temporal Logics* (the bits concerning LTL), from *Model Checking*, E. M. Clarke, O. Grumberg, D. A. Peled.



Go for the Temporal Logic section in Ch. 6, *Automata and Logic*, then Ch. 4, *Defining Correctness Claims* from *The* SPIN *Model Checker*, G. J. Holzmann.



You may also read on the topic from Ch. 3, *Linear Time Properties*, from *Principles of Model Checking*, C. Baier, J.-P. Katoen.

Assignment

(70) Do **[LMC'15] Assignment 3**, all four points. This has you practice your temporal logic.

(20) Prove any two of the "interesting dualities" on slide 17.

(20) The Pathfinder released a robot to roam on Mars in 1997. The robot's software controls occasionally failed during the mission[9], causing loss of contact. The essence of the software was modelled in PROMELA: see the SPIN sources, `pathfinder.pml`. The thread priorities are modelled with a `provided`[10], and the "valid" end states with `end` labels[11].

Run SPIN and display the trail(s) to any deadlock state. Draw the Kripke structure and confirm the results. Explain why any deadlock states can appear.

---

[9] For more details, see `http://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft` .

[10] See the manual, `http://spinroot.com/spin/Man/provided.html`

[11] See the manual, `.../Man/end.html`

Assignment (continued)

(30) A piece of concurrent software which claims to ensure mutual exclusion among processes should have the following properties:

1. Mutual exclusion: Two processes are never both in the critical section.
2. Progress: If neither process is in the critical section, and both processes try to enter, then one of them will eventually succeed.

Take the following very simple (and intuitive) attempt at mutual exclusion. Rewrite it in Promela and check whether it satisfies the properties above (which you can write in LTL[12], then use `./pan -a` to verify).

```
bool want1 = false, want2 = false;     // global variables

// process 1                           // process 2
while (true)                           while (true)
   ...                                    ...
   want1 = true                           want2 = true
   wait_until (want2 == false)            wait_until (want1 == false)
   ...   // critical section here        ...   // critical section here
   want1 = false                          want2 = false
```

[12] http://spinroot.com/spin/Man/ltl.html

Assignment administration

Handing in:

- in lab session or electronically;
- either way, before the end of Fri Dec 4.

Grading:

- this assignment adds up to 140  , or 1.4 grade points.