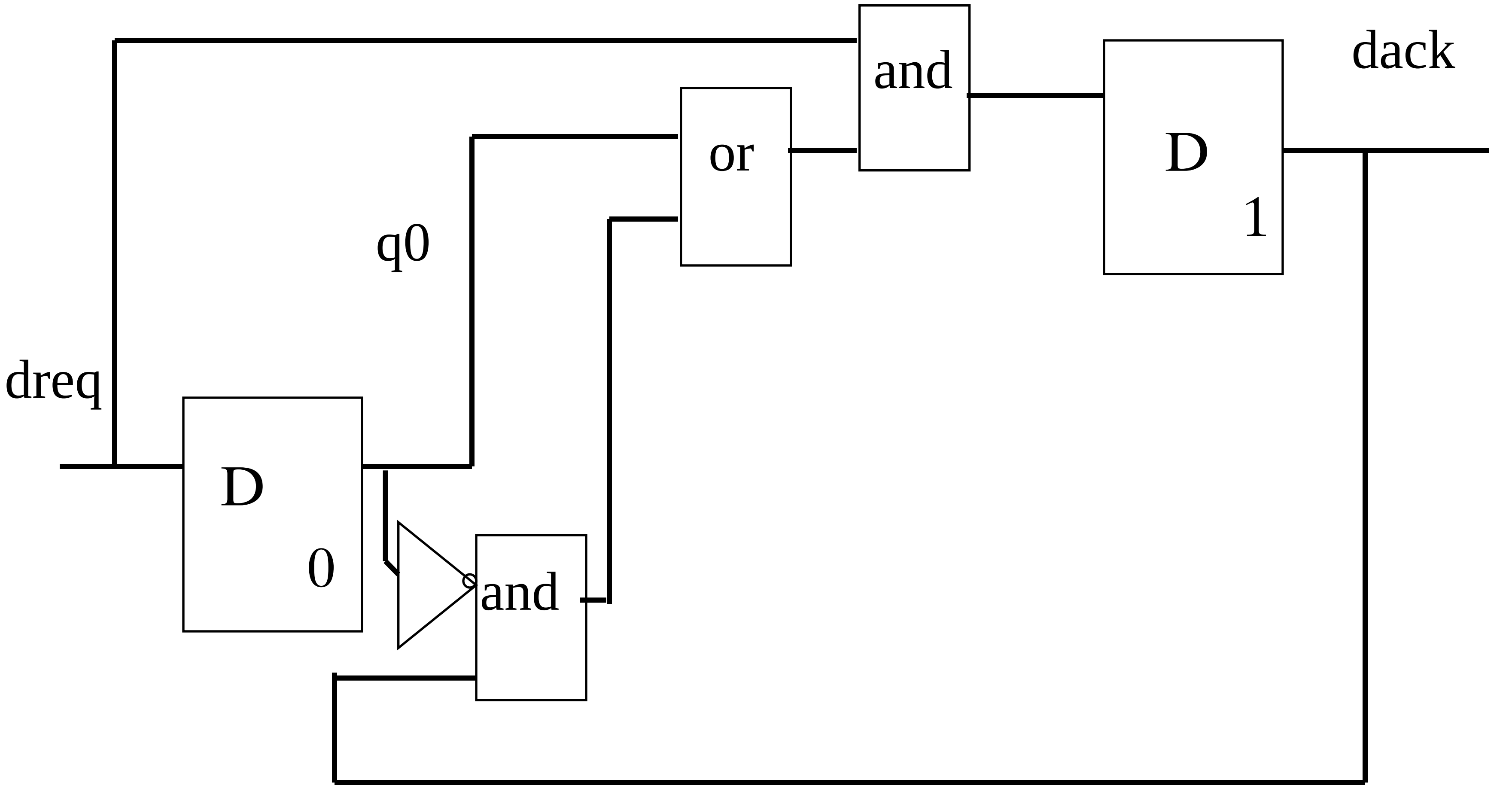
Model Checking I

What are LTL and CTL?



and
dack
or
D
1
q0
dreq
D
0
and

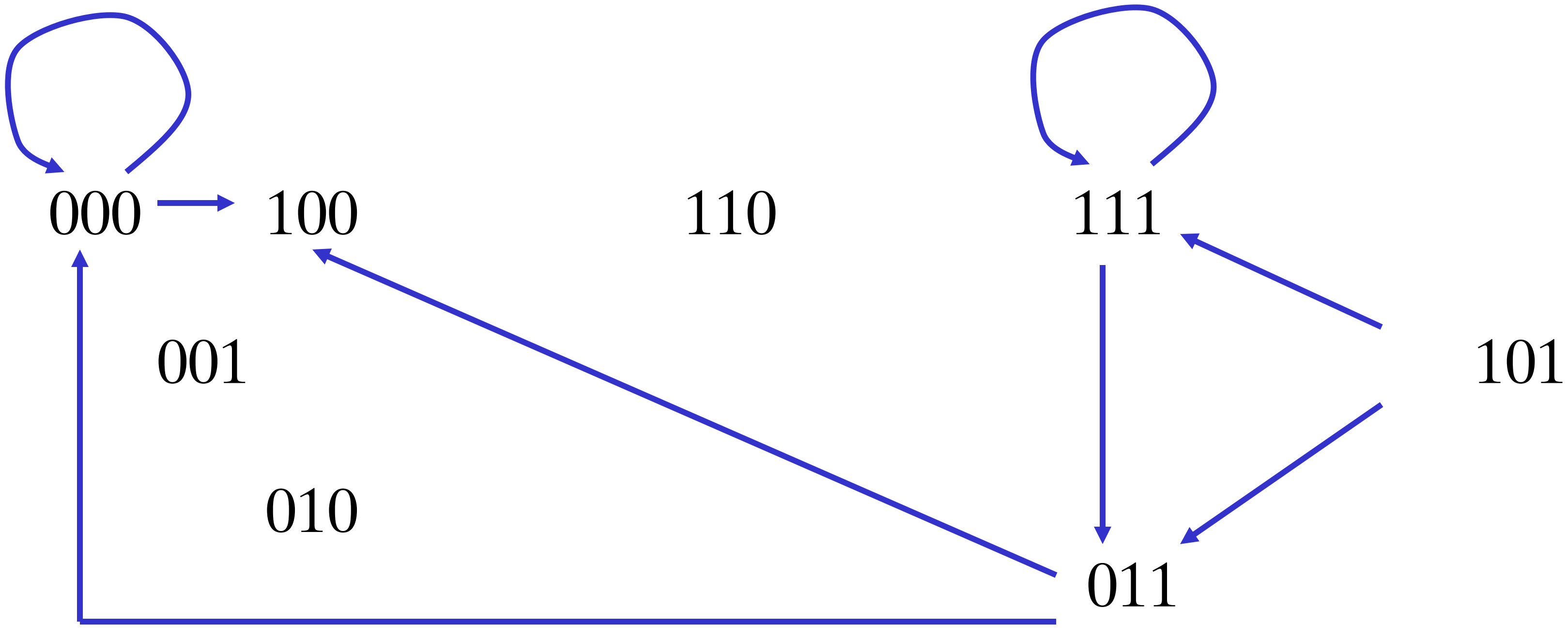
View circuit as a transition system

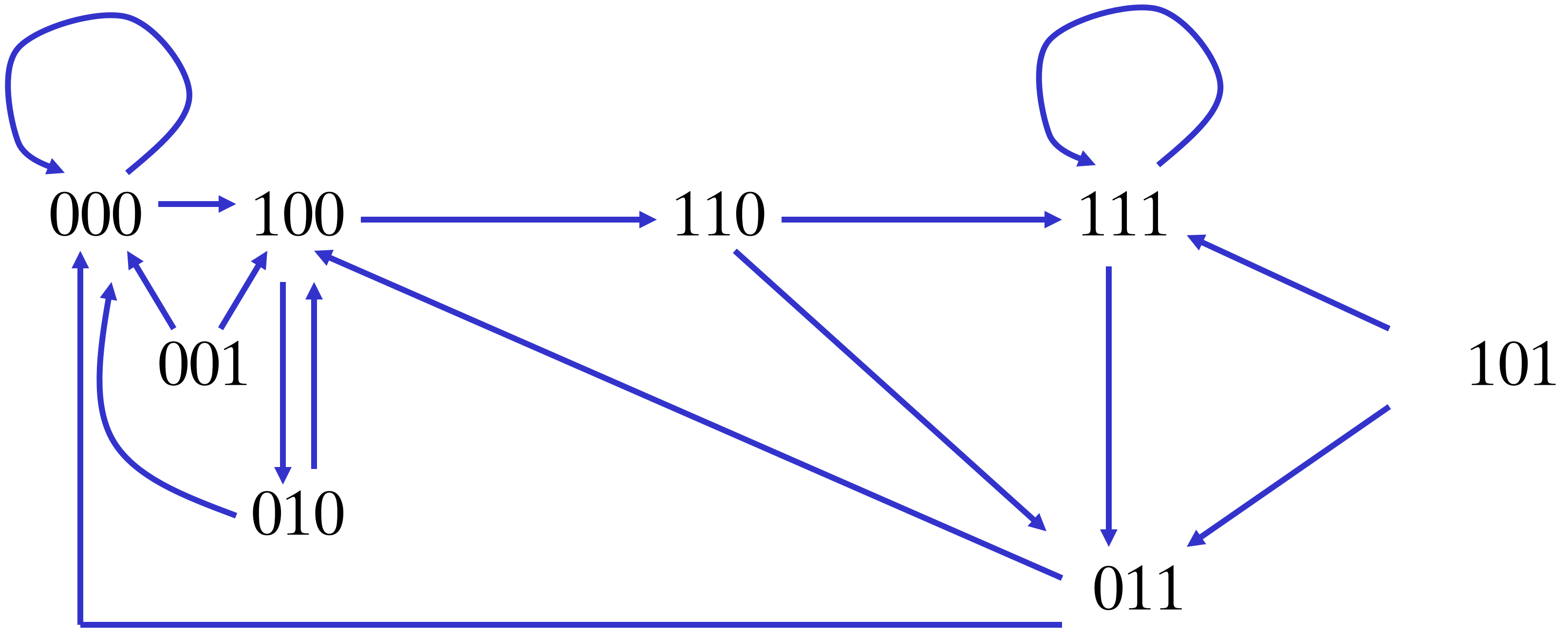
(dreq, q0, dack) → (dreq', q0', dack')

q0' = dreq

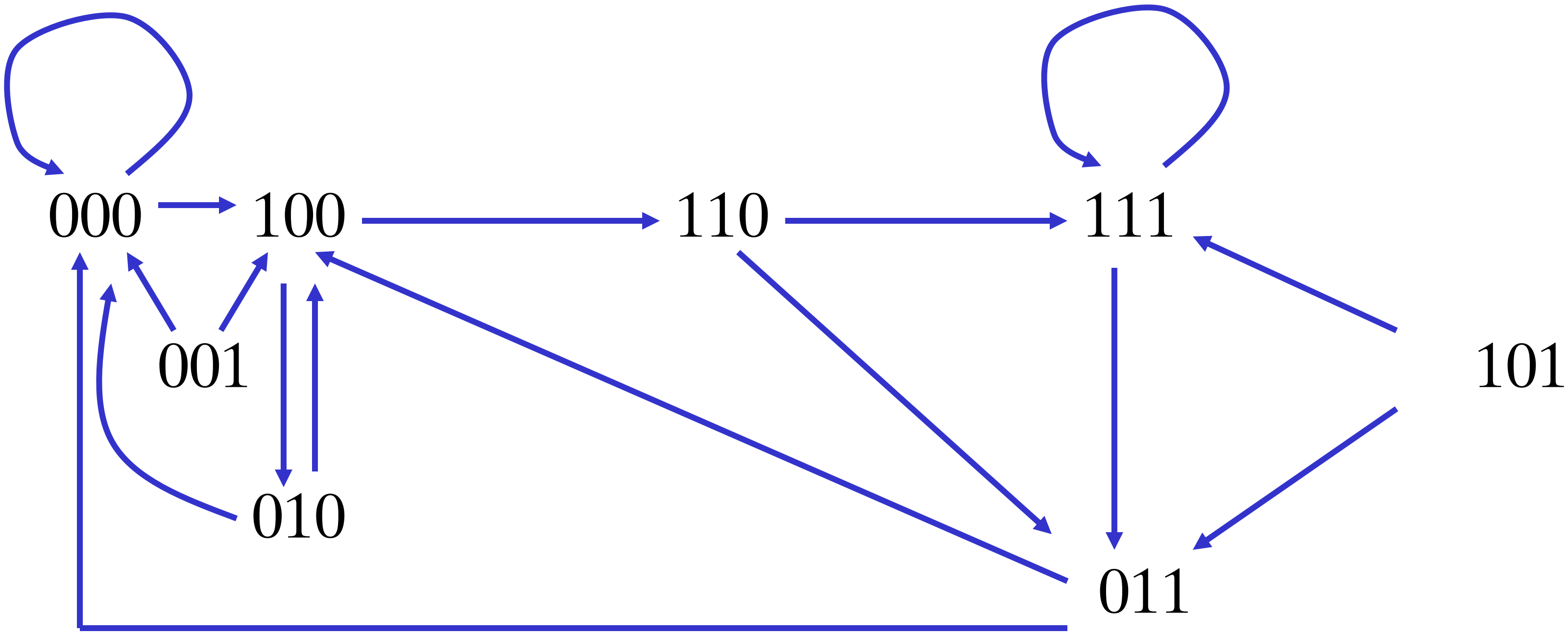
dack' = dreq and (q0 or (not q0 and dack))

exercise





000
100
110
111
001
101
010
011

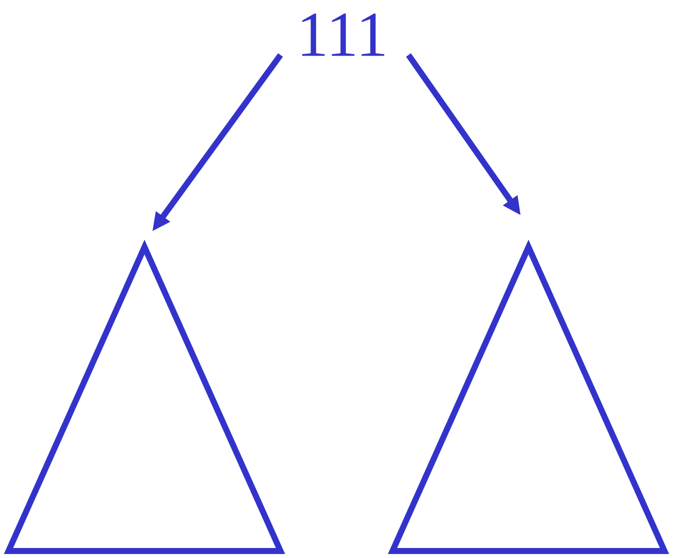


can also view transitition relation as a set of pairs of states, one pair per arrow

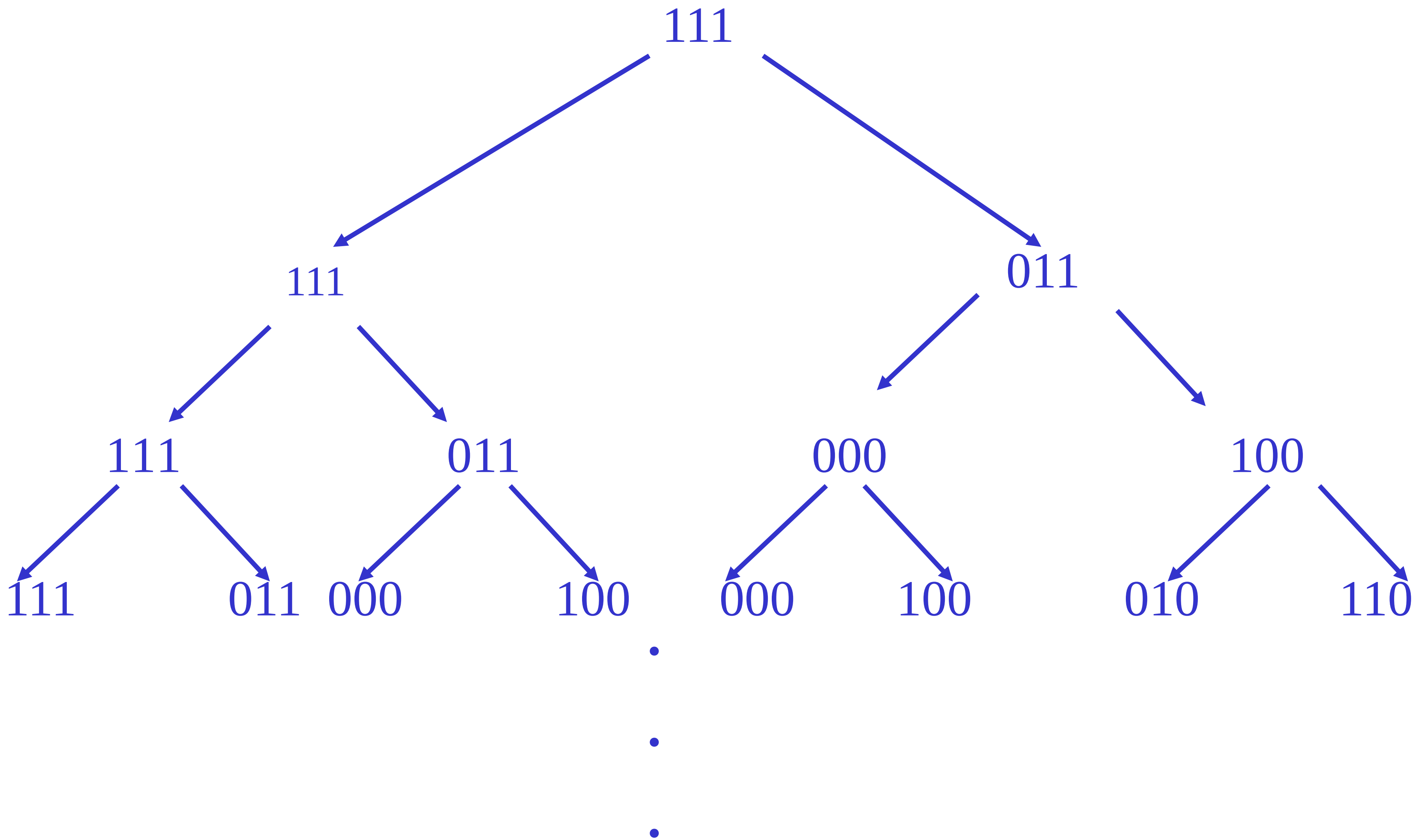
{(000,000), (000,100), (001,000), (001,100),
(010,000), (010,100), (011,000), (011,100),
(100,010), (100,110), (101,011), (101,111),
(110,011), (110,111), (111,011), (111,111)}

Another view

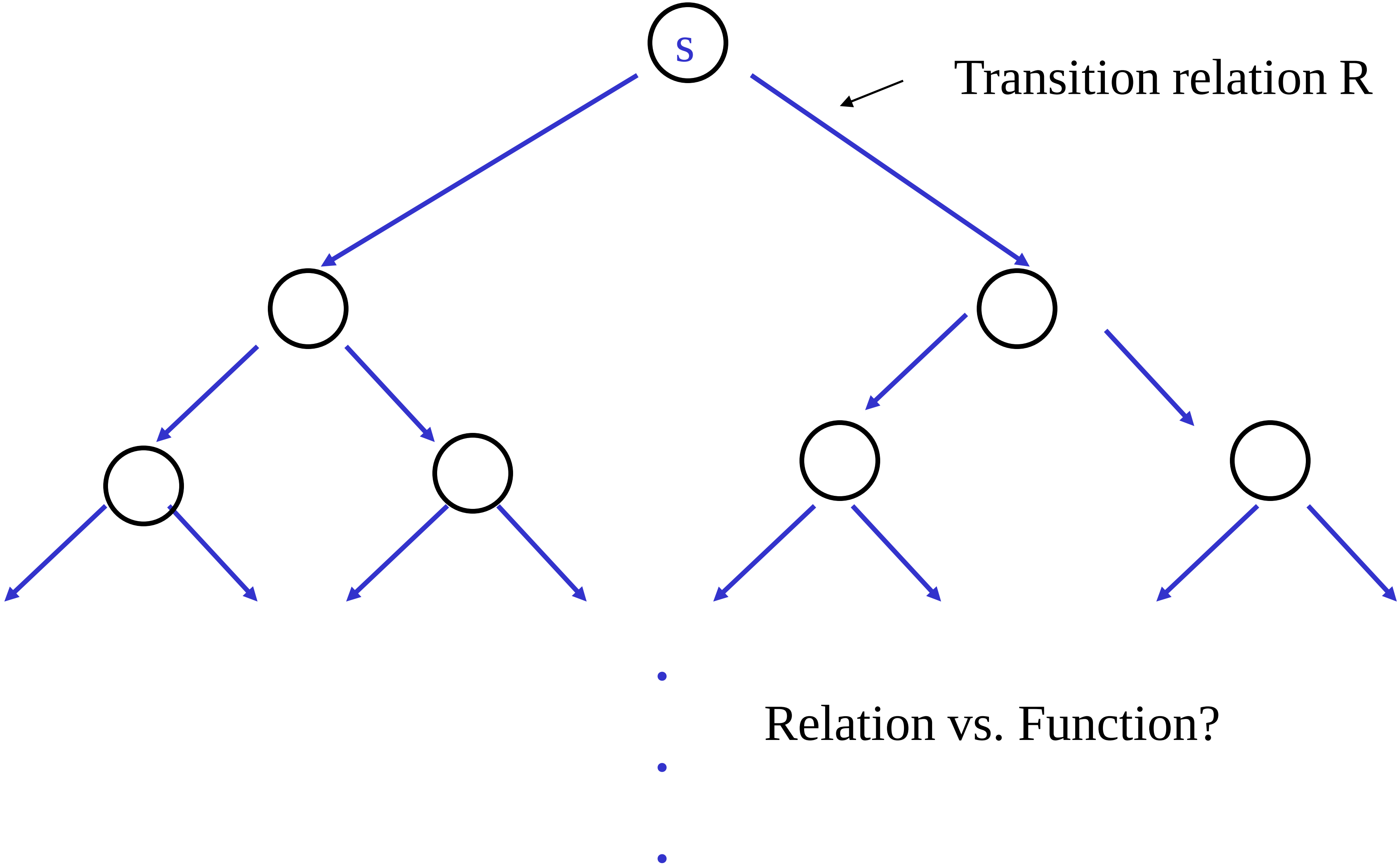
computation tree from a state



Unwinding further

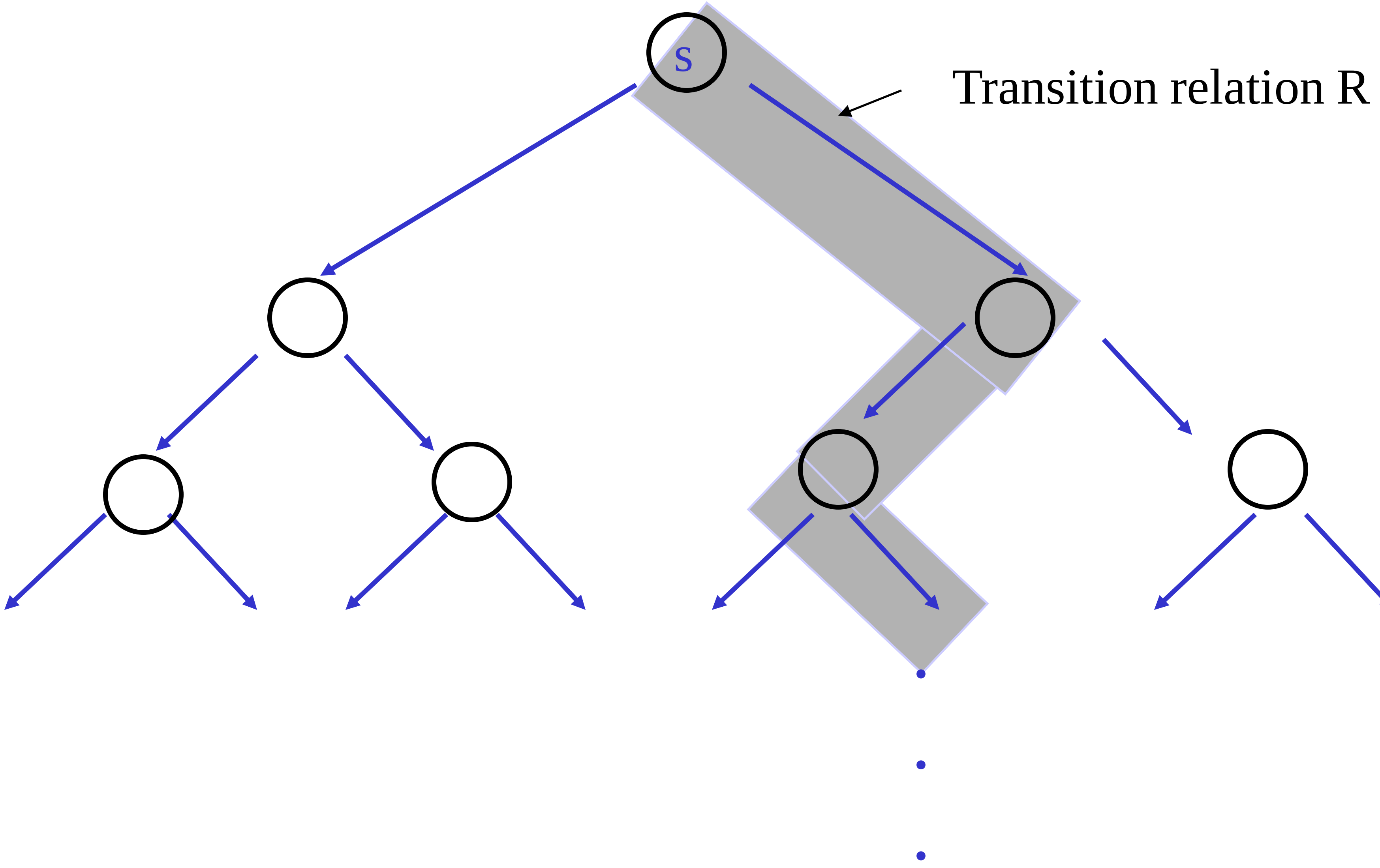


Possible behaviours from state s



Relation vs. Function?

path = possible run of the system



Points to note

Transition system models circuit behaviour

We chose the tick of the transition system to be the same as one clock cycle. Gates have zero delay – a very standard choice for synchronous circuits

Could have had a finer degree of modelling of time (with delays in gates). Choices here determine what properties can be analysed

Model checker starts with transition system. It doesn't matter where it came from

Transition system M

S set of states (finite)

R binary relation on states

assumed total, each state has at least one arrow out

A set of atomic formulas

L function S -> set of atomic formulas that hold in that state

Lars backwards ☺ finite Kripke structure

Path in M

Infinite sequence of states

$\pi = s0\ s1\ s2 \ldots$ such that

Path in M

s0 → s1 → s2 → ...

→

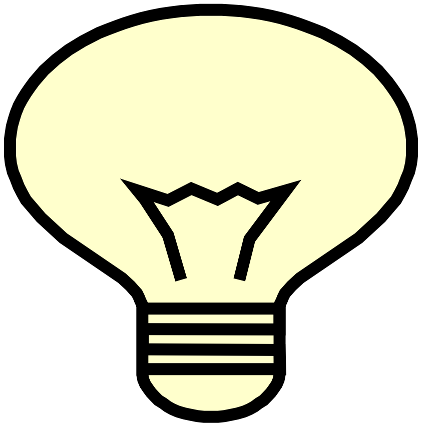
R

(s0,s1) ϵ R

(s1,s2) ϵ R

etc

Idea



Transition system

+ special temporal logic

+ automatic checking algorithm

Properties

Express desired behaviour over time using special logic

LTL (linear temporal logic)

CTL (computation tree logic)

CTL* (more expressive logic with both LTL and CTL as subsets)

Questions?

CTL*

path quantifers

A “for all computation paths”

E “for some computation path”

can prefix assertions made from

Linear operators

G “globally=always”

F “sometime”

X “nexttime”

U “until”

} about a path

CTL* formulas (syntax)

path formulas

$f ::= s \mid \neg f \mid f1 \vee f2 \mid X\, f \mid f1\; U\; f2$

state formulas (about an individual state)

$s ::= a \mid \neg s \mid s1 \vee s2 \mid E\, f$

↑

atomic formulas

Build up from core

$A\,f = \neg E \neg f$

$F\ f = true\ U\ f$

$G\ f = \neg F \neg f$

Example

G (req -> F ack)

Example

G (req -> F ack)

A request will eventually lead to an acknowledgement

liveness

linear

Example

G (req ->

A reques
acknow

liveness

linear

Liveness property

can only be proved false by exhibiting an infinite path (run). Any finite path can be extended to satisfy the eventuality condition

Example

G (req ->

A request
acknow

Safety and liveness in this sense introduced by Lamport in a 1976 paper (about manual proof of his Bakery algorithm)

liveness

linear

Example (Gordon)

It is possible to get to a state where Started holds but Ready does not

Example (Gordon)

It is possible to get to a state where Started holds but Ready does not

E (F (Started & ¬Ready))

Next: semantics

Questions?

Semantics

M = (L,A,R,S)

$M, s \models f$ f holds at state s in M
(and omit M if it is clear which M
we are talking about)

$M, \pi \models g$ g holds for path π in M

Semantics

Back to syntax and write down each case

$s \models a$ $\quad$ a in L(s) (atomic)

$s \models \neg f$ $\quad$ not $(s \models f)$

$s \models f1 \vee f2$ $\quad$ $s \models f1$ or $s \models f2$

$s \models E\ (g)$ $\quad$ Exists π. head $(\pi) = s$ and $\pi \models g$

Semantics

Back to syntax and write down each case

$s \models a$ a in L(s) (atomic)

$s \models \neg f$

$s \models f1 \vee$

$s \models E\ (g)$

English

a holds in state s if and only if a is in the set of atomic propositions associated with s

English

$\neg$f holds in s if and only if it is not the case that f holds in s

Semantics of a formula expressed in terms of semantics of its parts. Recursive, with base case being the rule about atomic formulas

Back to

s $\models$ a

s $\models$ $\neg$f   not (s $\models$ f)

s $\models$ f1 $\vee$ f2   s $\models$ f1 or s $\models$ f2

s $\models$ E (g)   Exists π. head (π) = s and $\pi \models$ g

Semantics

$\pi \models f \qquad s \models f$ and $head(\pi) = s$

$\pi \models \neg g \qquad$ not $(\pi \models g)$

$\pi \models g1 \vee g2 \qquad \pi \models g1$ or $\pi \models g2$

Semantics

$\pi \models X\, g$ $tail(\pi) \models g$

$\pi \models g1\ U\ g2$

Exists $k \geq 0$. $drop\ k\ \pi \models g2$ and

Forall $0 \leq j < k$. $drop\ j\ \pi \models g1$

(note: I mean tail in the Haskell sense)

CTL

Restrict path formulas (compare with CTL*)

$f ::= \neg f \;\mid\; s1 \vee s2 \;\mid\; X\, s \;\mid\; s1\, U\, s2$

state formulas

Linear time ops (X,U,F,G) must be wrapped up in a path quantifier (A,E).

Back to CTL* formulas (syntax)

path formulas

$f ::= s \mid \neg f \mid f1 \vee f2 \mid X\, f \mid f1\ U\ f2$

state formulas (about an individual state)

$s ::= a \mid \neg s \mid s1 \vee s2 \mid E\, f$

↑

atomic formulas

CTL* yes CTL ?

E X X f

E (f U (g U j))

A (f U Xg)

A (f U g) ∨ EF k

CTL* yes CTL ?

$E\ X\ X\ f$

$E\ (f\ U\ (g\ U\ j))$

$A\ (f\ U\ Xg)$

$A\ (f\ U\ g) \lor EF\ k$ Yes

CTL

Another view is that we just have the combined operators

AU, AX, AF, AG and EU, EX, EF, EG

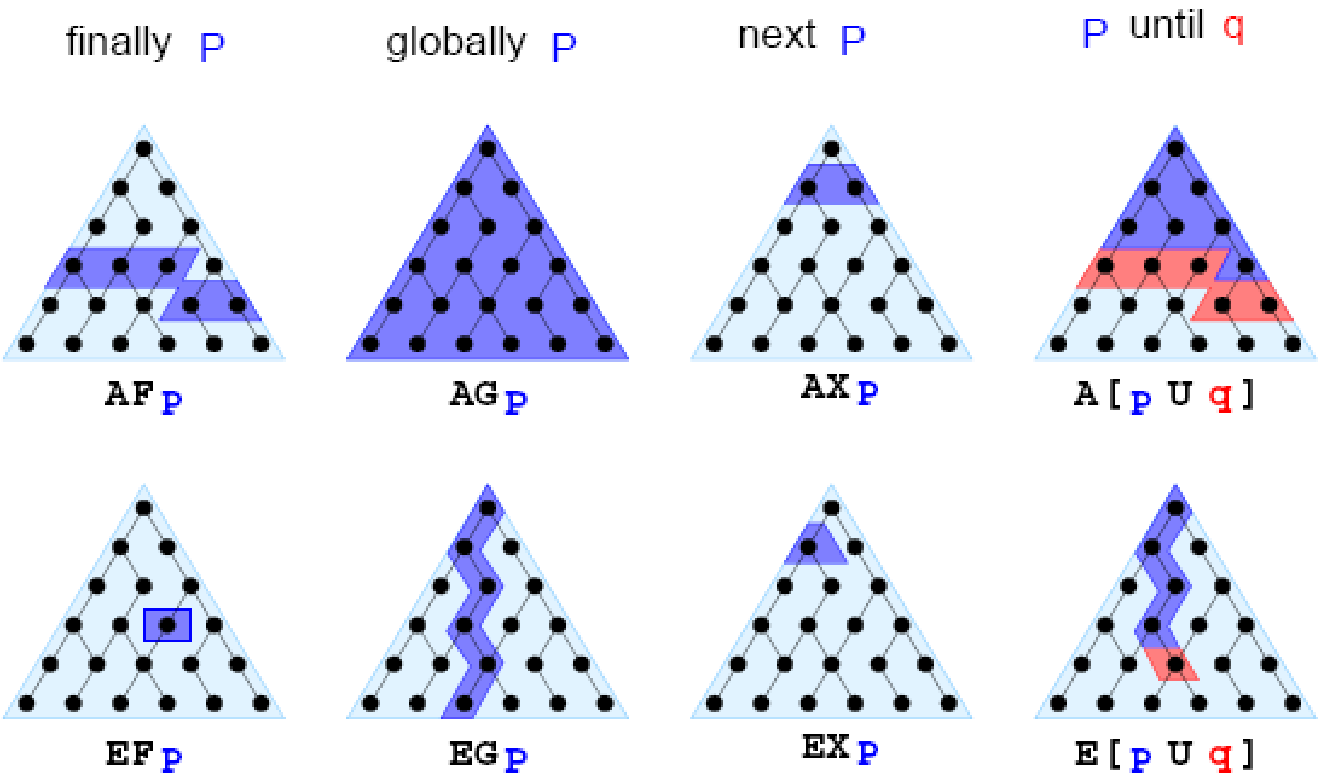
and only need to think about state formulas

A operators for necessity

E operators for possibility

f	::=	atomic
		$\mid \neg f$
All immediate successors		$\mid$ AX f
Some immediate succesor		$\mid$ EX f
All paths always		$\mid$ AG f
Some path always		$\mid$ EG f
All paths eventually		$\mid$ AF f
Some path eventually		$\mid$ EF f
		$\mid$ f1 $\vee$ f2
		$\mid$ A (f1 U f2)
		$\mid$ E (f1 U f2)

CTL


finally p
globally p
next p
p until q
AF p
AG p
AX p
A [p U q]
EF p
EG p
EX p
E [p U q]

Examples (Gordon)

It is possible to get to a state where Started holds but Ready does not

Examples (Gordon)

It is possible to get to a state where Started holds but Ready does not

EF (Started & ¬Ready)

Examples (Gordon)

If a request Req occurs, then it will eventually be acknowledged by Ack

Examples (Gordon)

If a request Req occurs, then it will eventually be acknowledged by Ack

AG (Req -> AF Ack)

Examples (Gordon)

If a request Req occurs, then it continues to hold, until it is eventually acknowledged

Examples (Gordon)

If a request Req occurs, then it continues to hold, until it is eventually acknowledged

AG (Req -> A [Req U Ack])

LTL

LTL formula is of form A f where f is a path formula with subformulas that are atomic

(The f is what we write down. The A is implicit.)

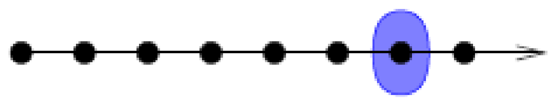
Restrict path formulas (compare with CTL*)

$$f ::= a \mid \neg f \mid f1 \vee f2 \mid X f \mid f1 \, U \, f2$$

↑ atomic formulas (Talk about a single state)

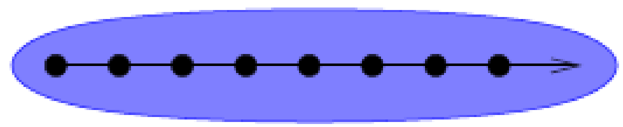
LTL

finally p



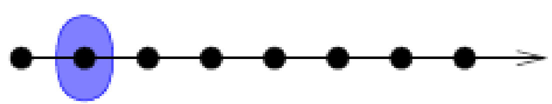
F p

globally p



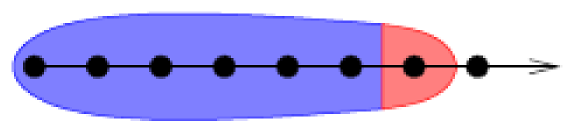
G p

next p



X p

p until q



p U q

LTL

It is the restricted path formulas that we think of as LTL specifications (See P&R again)

$G \neg(critical1 \;\&\; critical2)$ mutex

FG initialised eventually stays initialised

GF myMove myMove will always eventually hold

G (req -> F ack) request acknowledge pattern

LTL

Responsiveness (more examples)

G (req -> XF ack)

G (req -> X(req U ack))

G (req -> X((req & ¬ ack) U (¬ req & ack)))

LTL

p holds at the even states and does not hold at the odd states

$p \,\&\, G\,(p \leftrightarrow \neg (X\,p))$

It is not possible to express that p holds in the even states (while not saying anything about the odd states) in LTL

In CTL but not LTL

AG EF start

Regardless of what state the program enters, there exists a computation leading back to the start state

In CTL but not LTL

$AG\ (R \rightarrow EX\ S)$

”non-blocking”

Even $EX\ P$ is an example

In both

$AG\ (p \rightarrow AF\ q)$ in CTL

$G(p \rightarrow F\ q)$ in LTL

In LTL but not CTL

$G\ F\ p \rightarrow F\ q$

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

$F\ G\ p$

In CTL* but not in LTL or CTL

E [G F p]

there is a path with infinitely many p

Further reading

Ed Clarke's course on **Bug Catching: Automated Program Verification and Testing**

complete with moving bug on the home page!

Covers model checking relevant to hardware too.

http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15414-f06/www/index.html

The sub-page called Reading has slides and paper links

For some history (by the inventors themselves) see this workshop celebrating 25 years of MC http://www.easychair.org/FLoC-06/25MC-day227.html

Example revisited

A sequence beginning with the assertion of signal strt, and containing two not necessarily consecutive assertions of signal get, during which signal kill is not asserted, must be followed by a sequence containing two assertions of signal put before signal end can be asserted

AG~(strt & EX E[~get & ~kill U get & ~kill & EX E[~get & ~kill U get & ~kill & E[~put U end] or E[~put & ~end U (put & ~end & EX E[~put U end])]]])

AG ~ ...

strt & EX E[~get & ~kill U get & ~kill & ...]

EX E [~get & ~kill U get & ~kill & ...]

E[~put U end] or

E[~put & ~end U (put & ~end & EX E[~put U end])]]

AG ~ ...

strt & EX E[~get & ~kill U get & ~kill & ...]

EX E [~get & ~kill U get & ~kill & ...]

zero puts

E[~put U end] or

E[~put & ~end U (put & ~end & EX E[~put U end])]]

AG ~ ...

strt & EX E[~get & ~kill U get & ~kill & ...]

EX E [~get & ~kill U get & ~kill & ...]

one put

E[~put U end] or

E[~put & ~end U (put & ~end & EX E[~put U end])]]

Next lecture

How to model check CTL formulas