

First some loose ends

Back to BDDs

(from lec. 3, week 1)

First form of FV Equivalence Checking (EC, CEC)

Boolean network comparison, also known as combinational equivalence checking

Straight BDD comparison works for moderately sized circuits. For larger circuits, more sophisticated methods are used.

Invisible to user, automatic, effective

Second form of FV

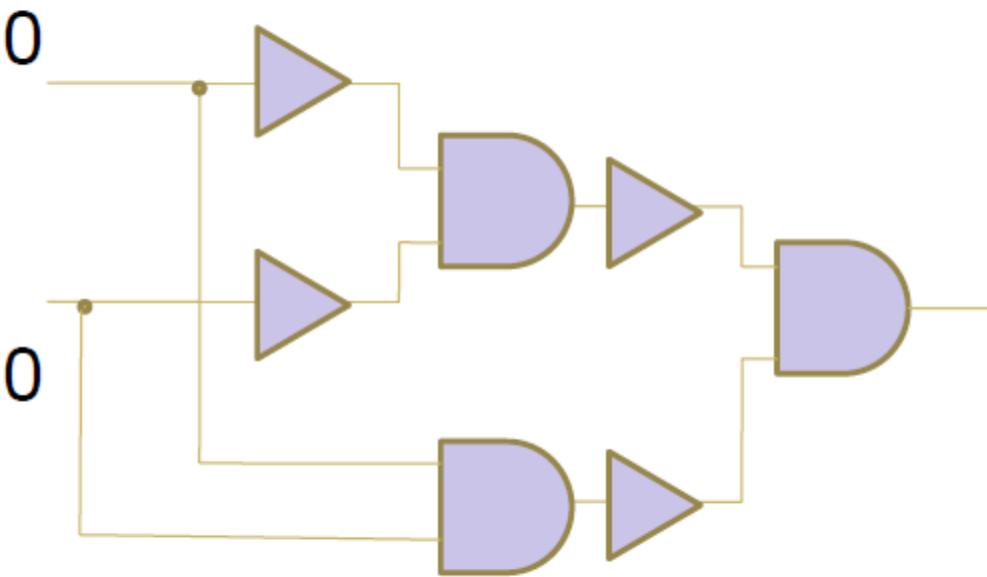
Symbolic simulation

Take a simulator (can be quite low level, accurate one)

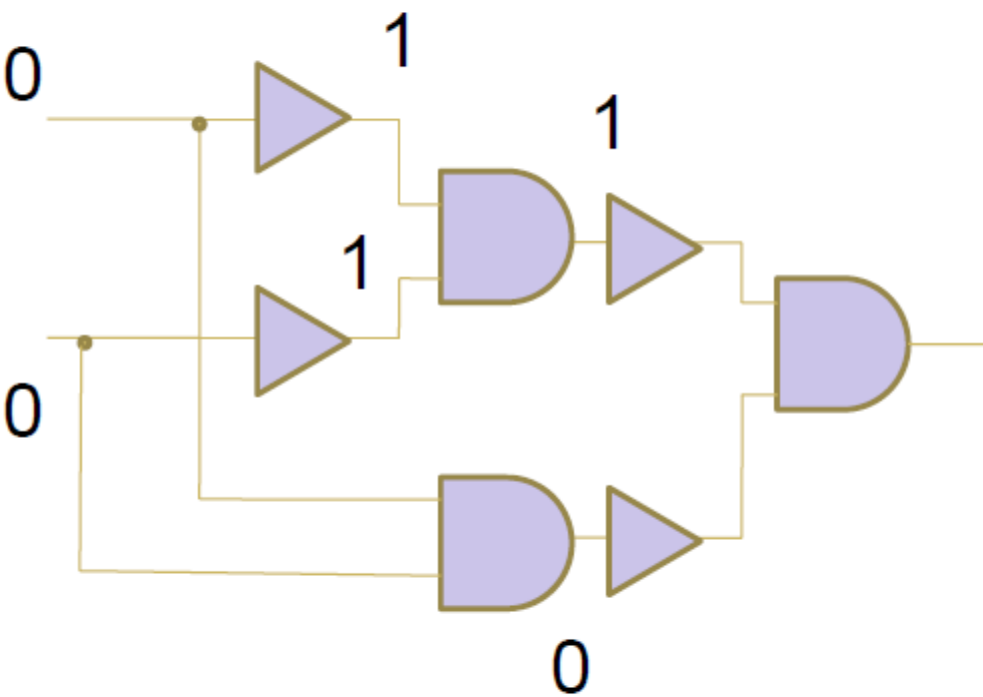
Make it work not only on 0, 1, X (unknown)

(or a larger group of constants) but ALSO on **symbols**

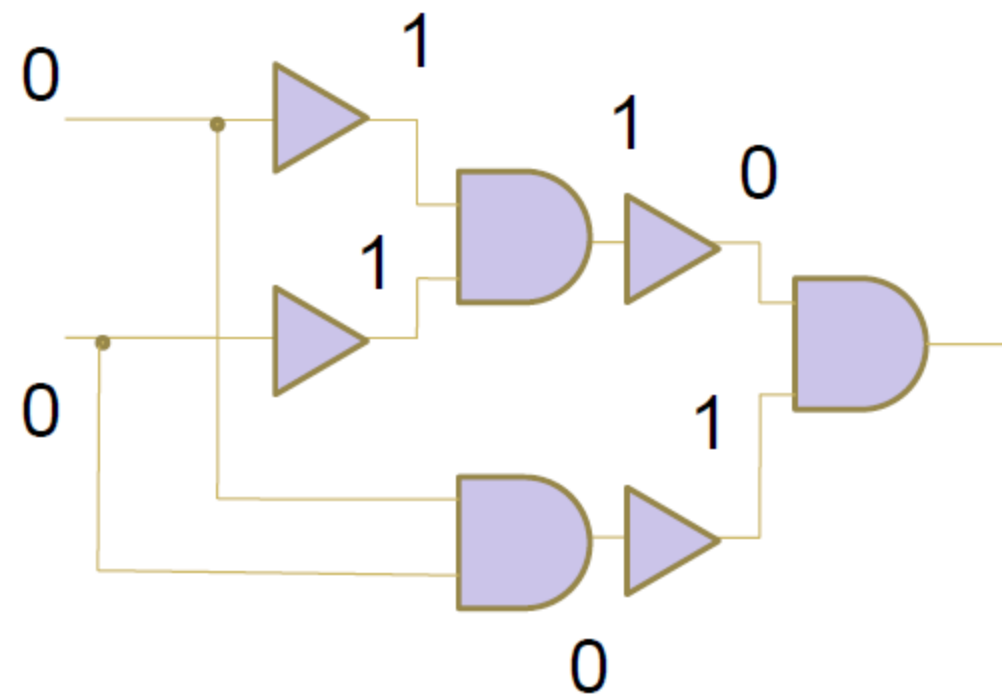
Ordinary simulation xor ?



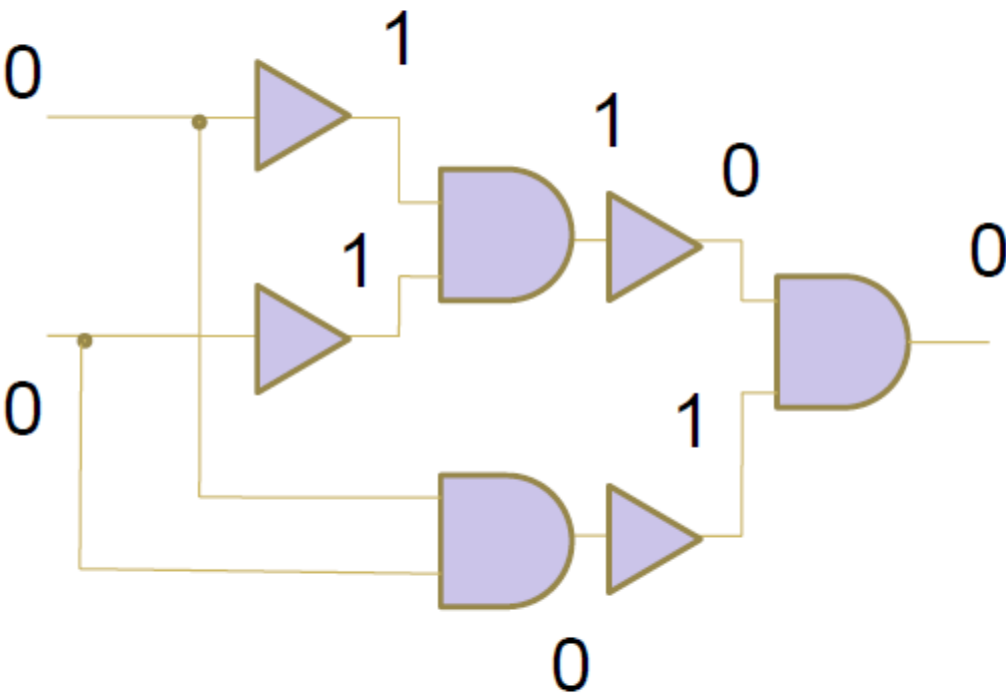
simulation



simulation



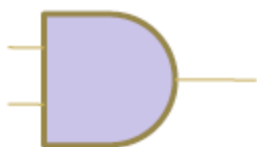
simulation



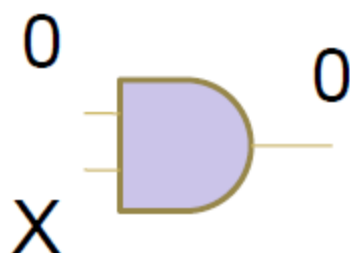
4 runs to check exhaustively

Q: how many for n inputs?

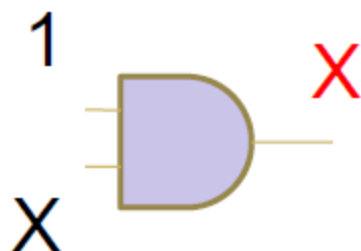
Symbolic simulation Idea 1



Use X values



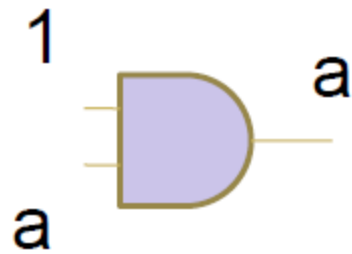
Halves number of sim. runs!



BUT may lose information

(try on xor example)

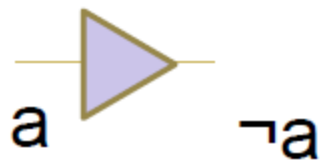
Symbolic simulation Idea 2



Use **symbolic** values

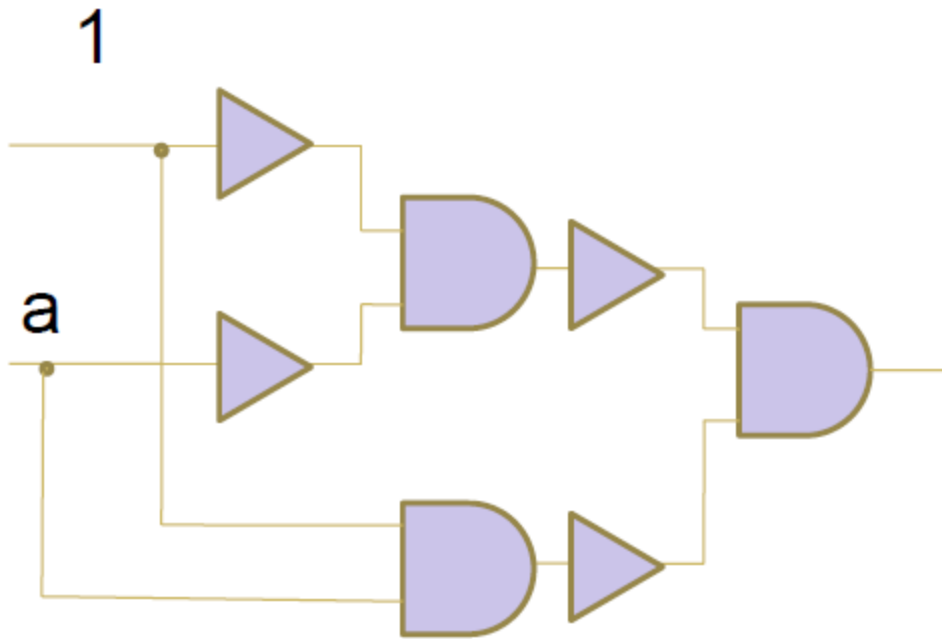
Think of giving input values names rather than constant values

Build up an expression in terms of (some of the) inputs

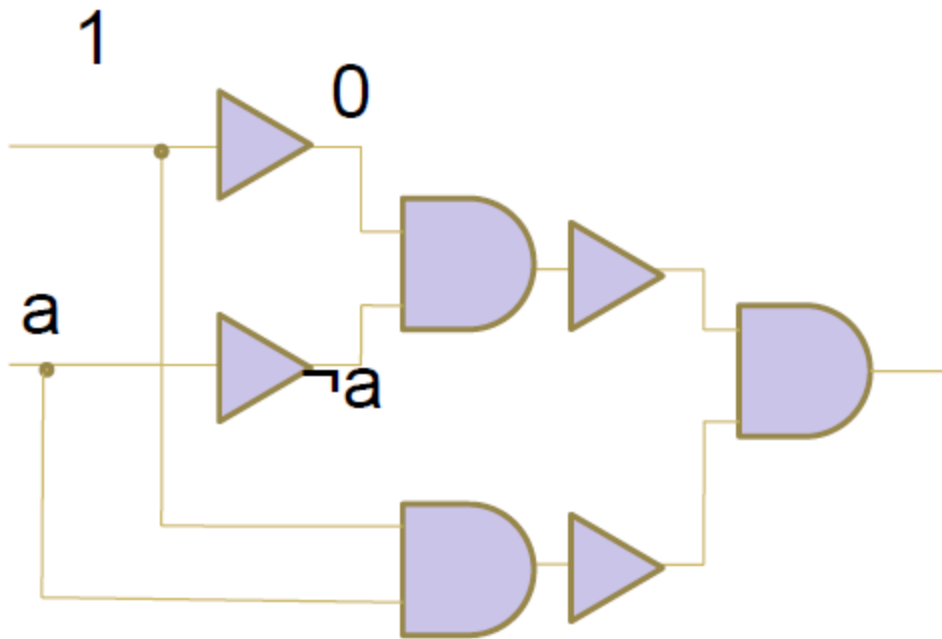


May Rep. Using Binary Decision Diagrams (BDDs)

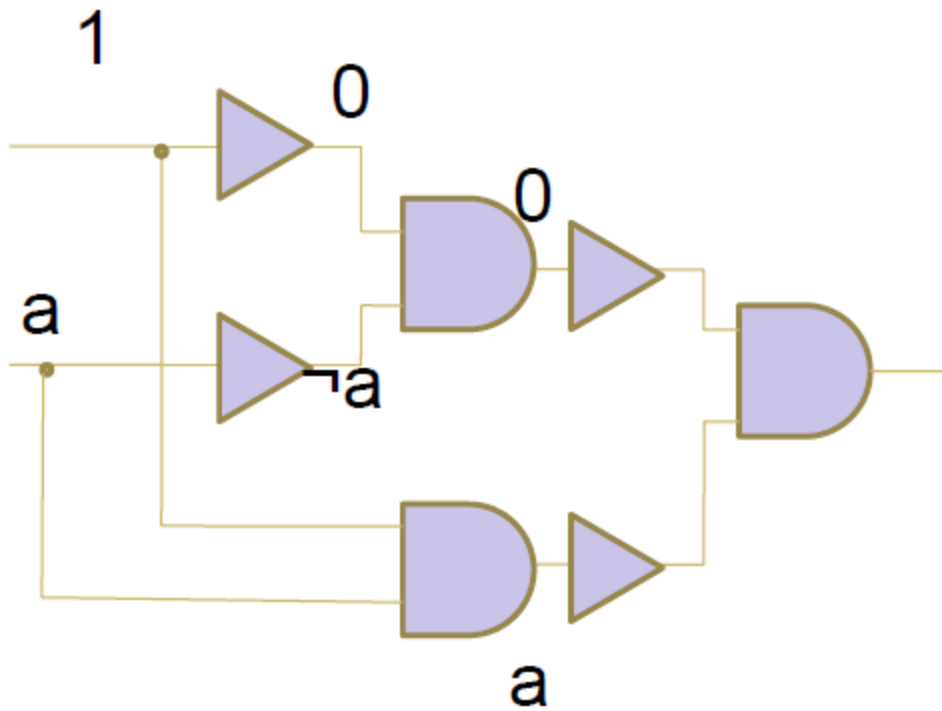
Symbolic simulation



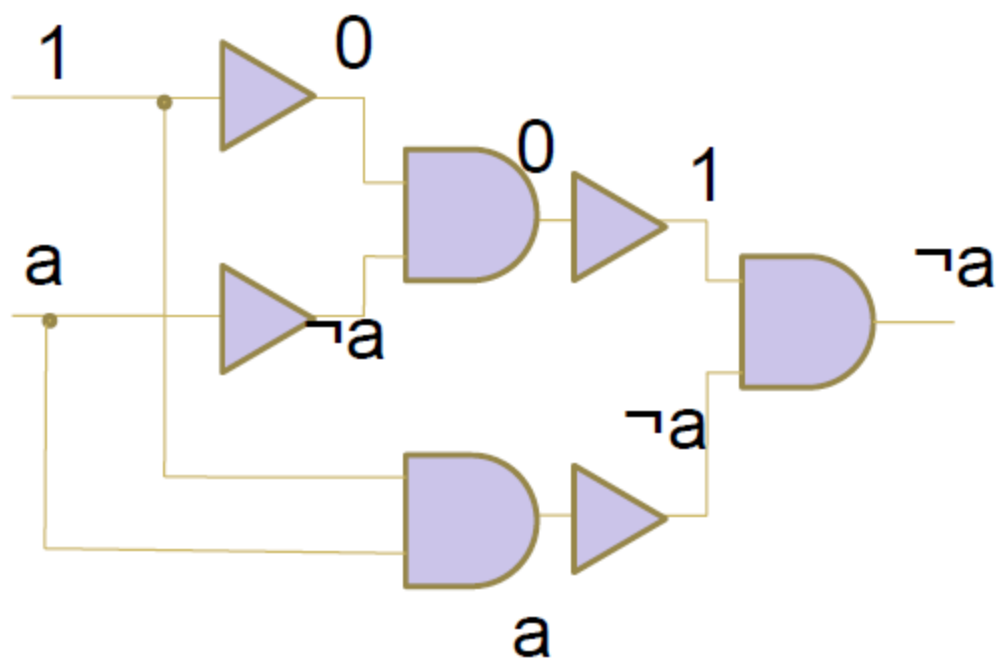
Symbolic simulation



Symbolic simulation



Symbolic simulation



| | |
|----|----------|
| 1X | X |
| 1a | $\neg a$ |

Symbolic simulation

Widely used (applies also to sequential circuits)

Forms basis of model checking method called Symbolic Trajectory Evaluation (STE)

User must make judicious choice of $0, 1, X, a, b, \dots$

X halves sim runs, but may result in X at a point vital to the verification

Symbolic variable halves sim. runs without losing info.
BUT BDD somewhere in the sim. may grow too big

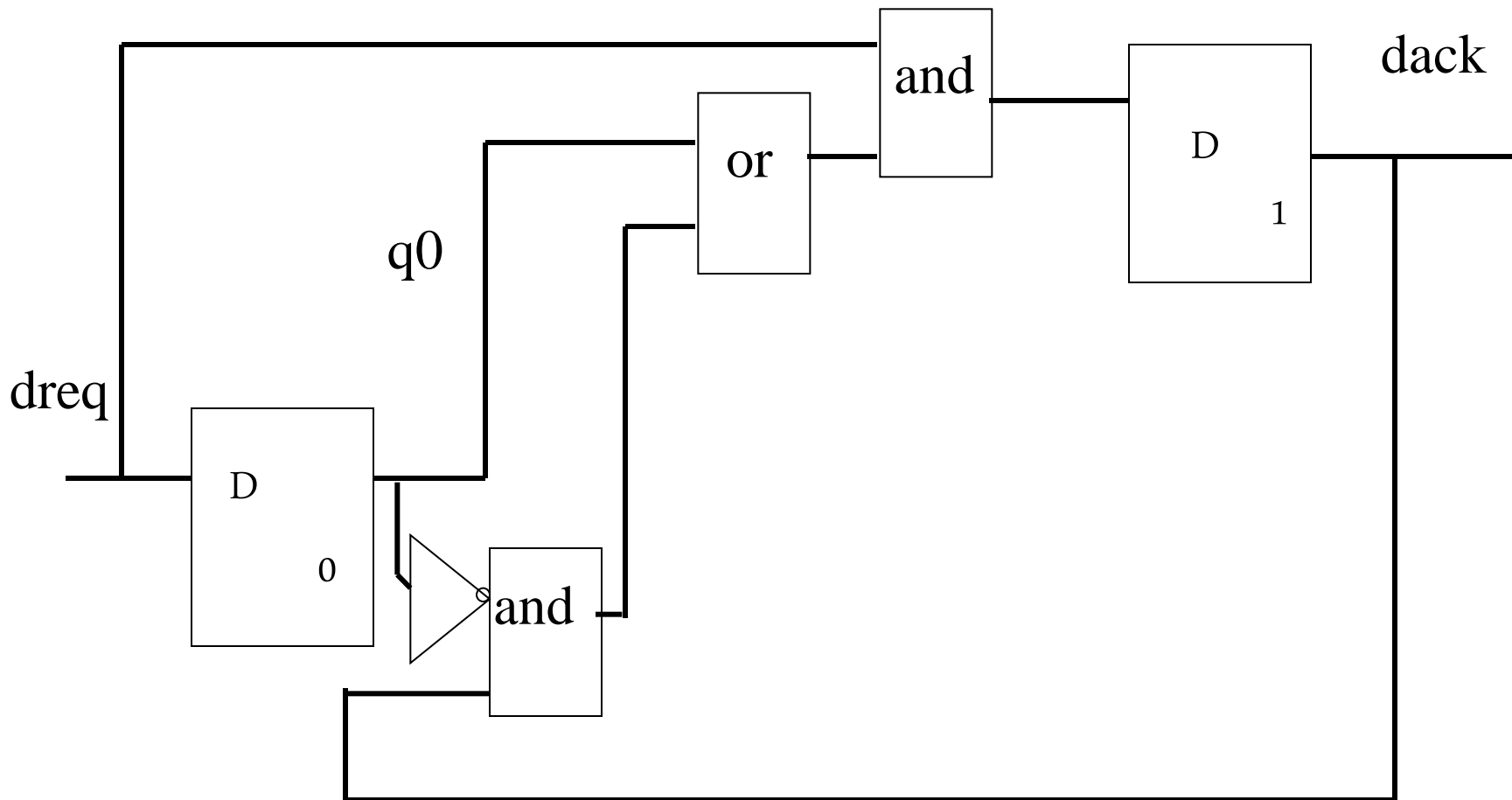
Pro and Cons of BDDs

- + Powerful operations (create, manipulate, test)
polynomial complexity, composable
- + Usually stay small enough
given good variable order
- + Provide quantification operations (unlike plain SAT) (see MC!)
- sometimes explode in size
- important circuits (multipliers and shifters) are problematic =>
yet more special hacks in the tools
- variable ordering problem is NP-complete

In practice used together with SAT and other engines

Model Checking I

What are LTL and CTL?



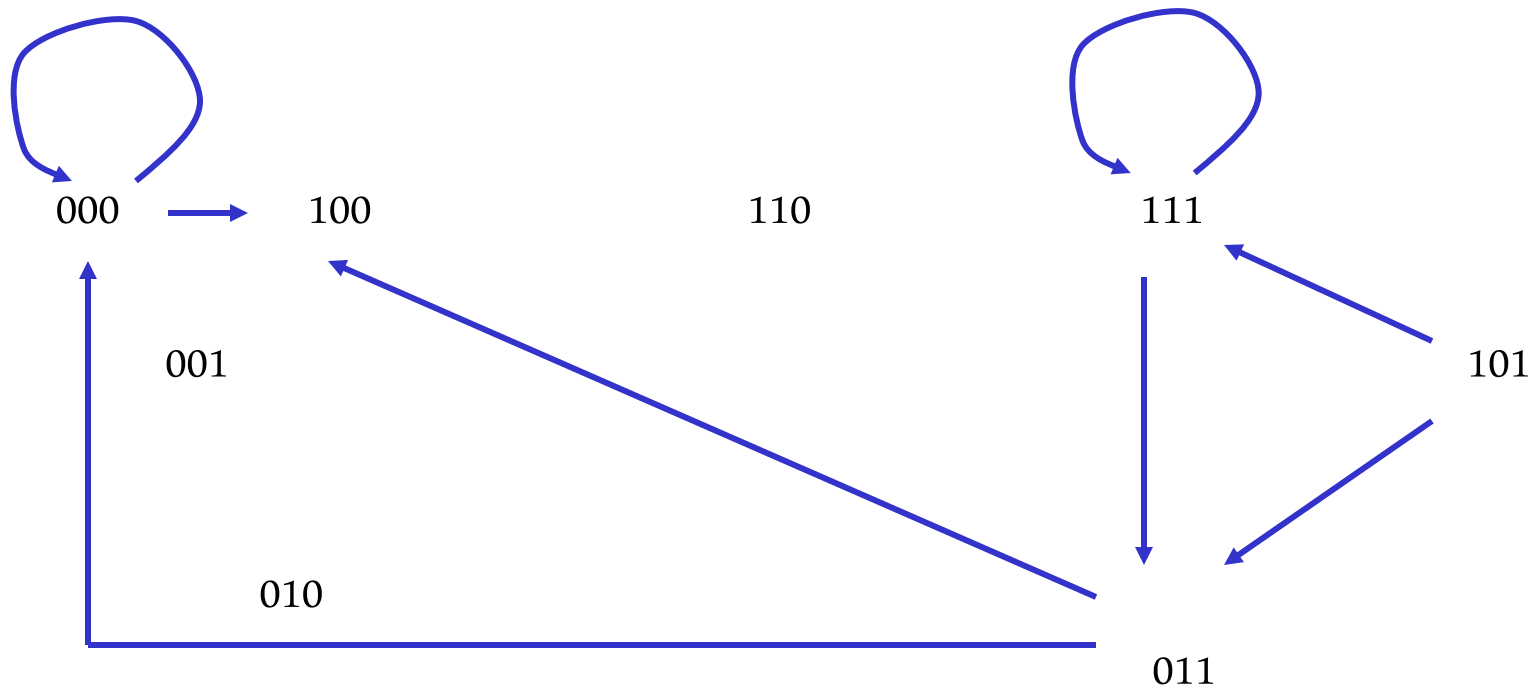
View circuit as a transition system

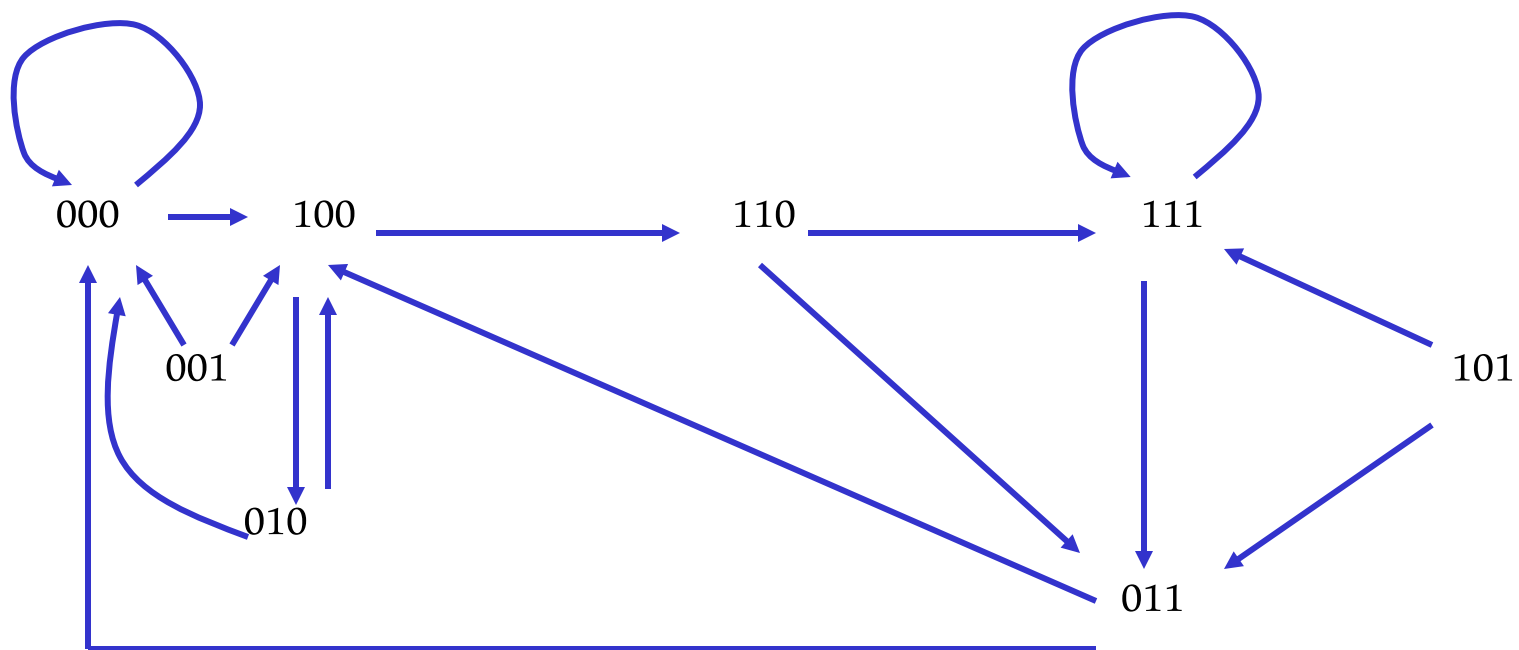
$$(\text{dreq}, q_0, \text{dack}) \rightarrow (\text{dreq}', q_0', \text{dack}')$$

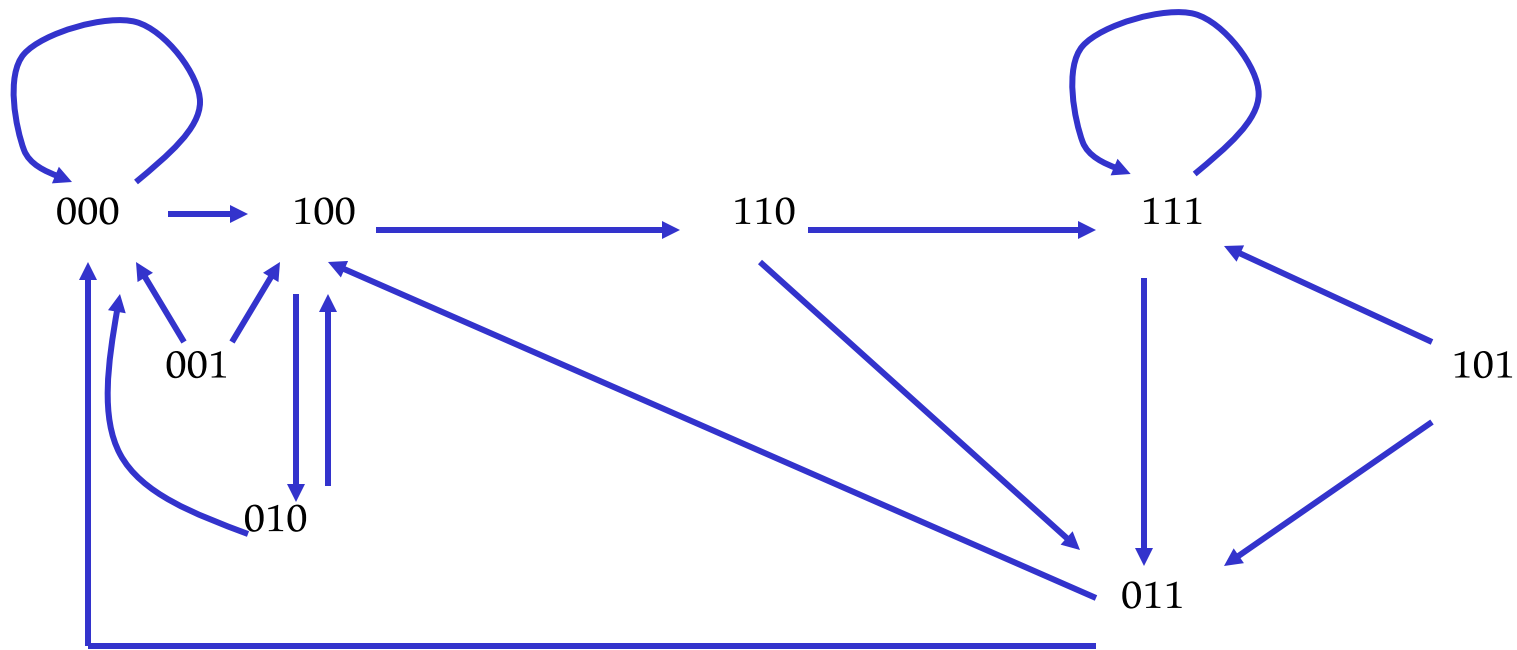
$$q_0' = \text{dreq}$$

$$\text{dack}' = \text{dreq} \text{ and } (q_0 \text{ or } (\text{not } q_0 \text{ and } \text{dack}))$$

exercise



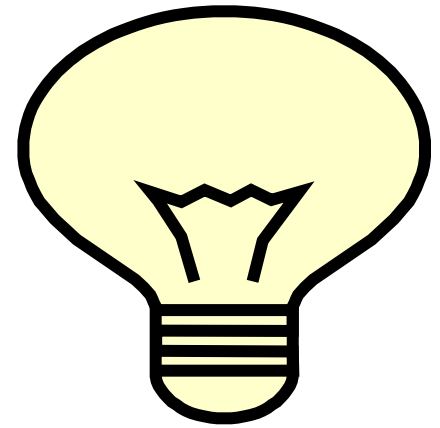




can also view transitiion relation as a set of pairs of states, one pair per arrow

{(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)}

Idea



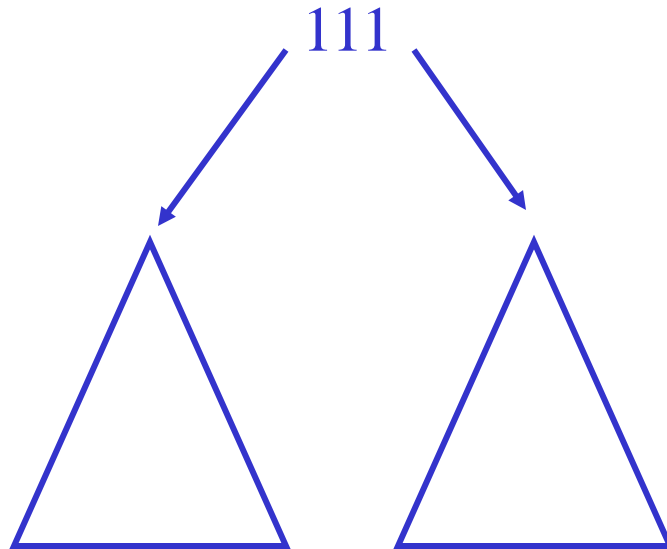
Transition system

+ special temporal logic

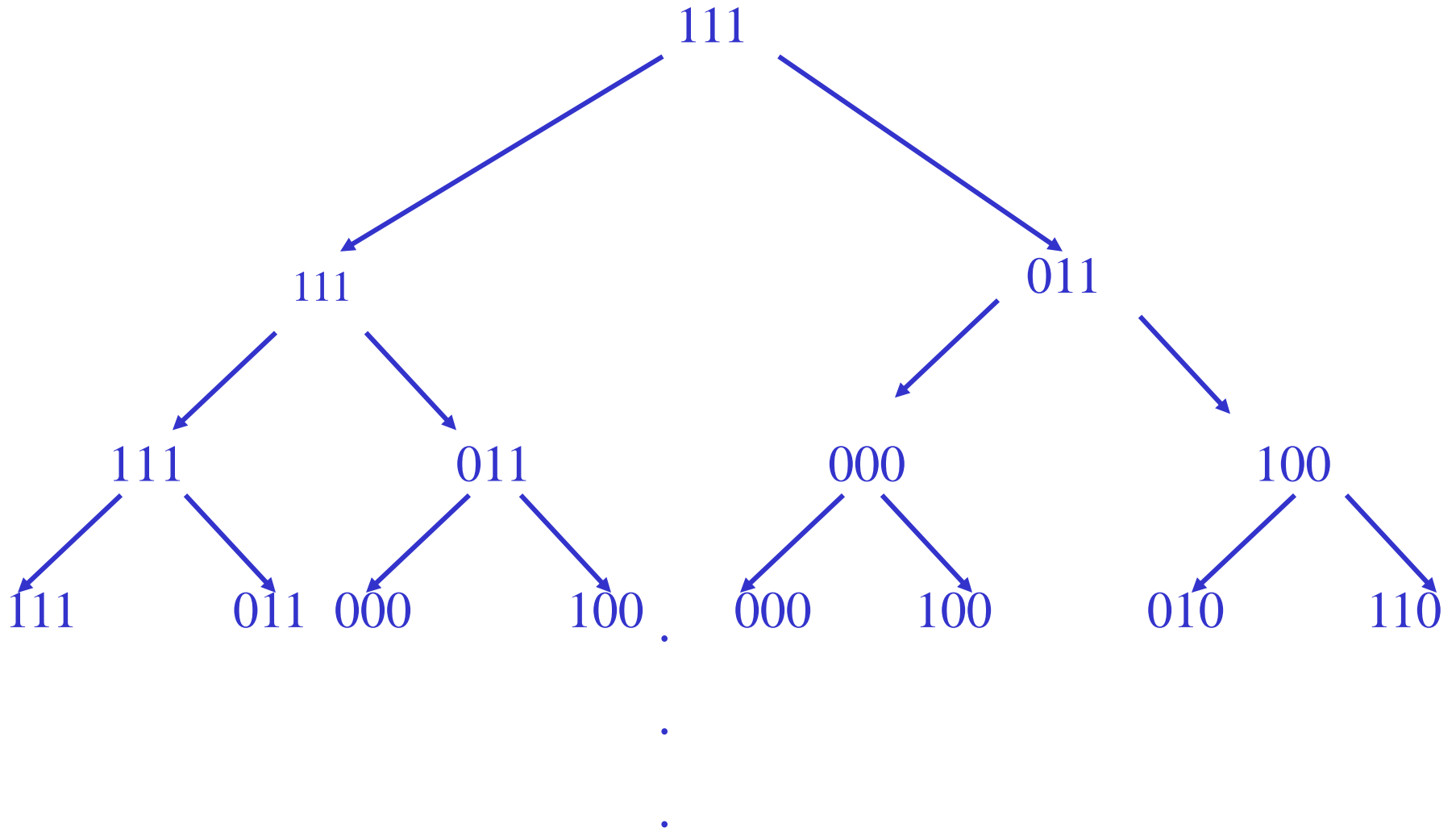
+ **automatic** checking algorithm

Another view

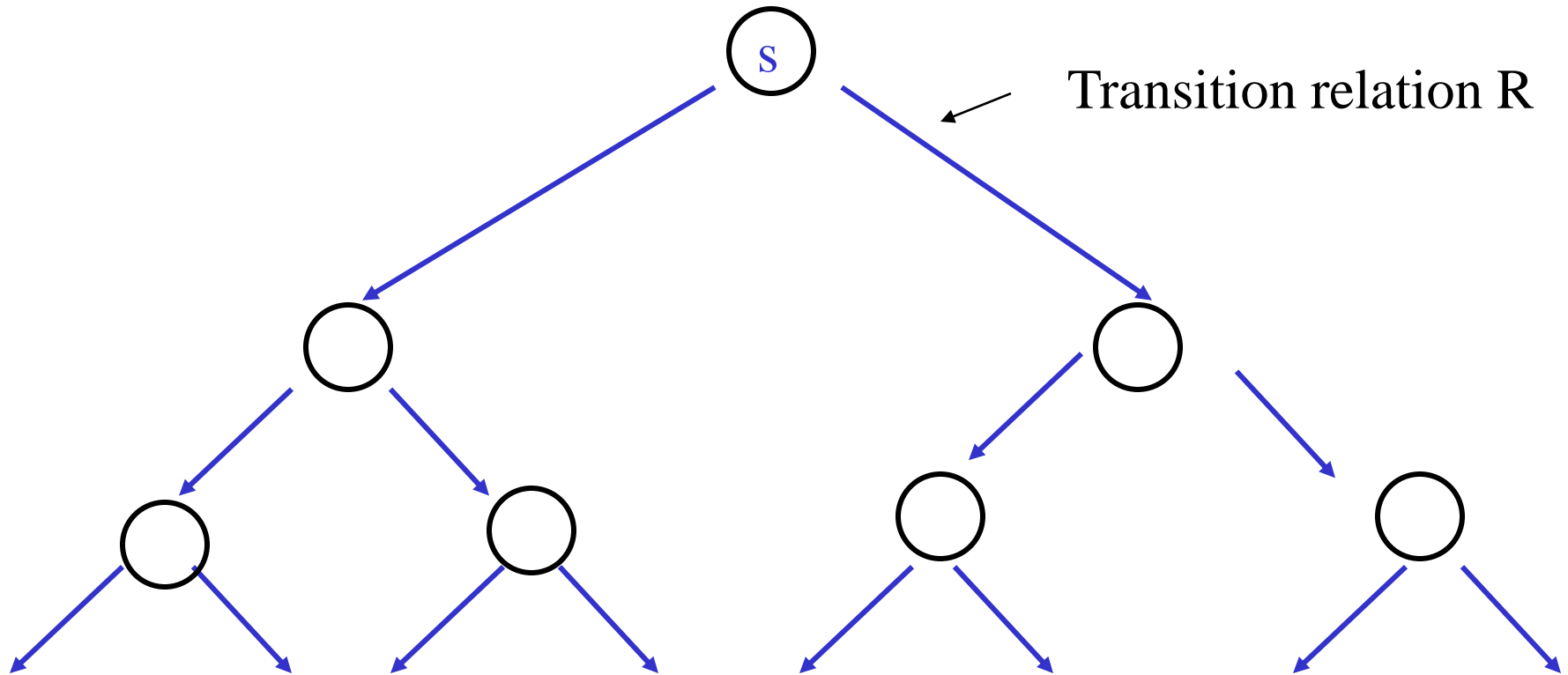
computation tree from a state



Unwinding further



Possible behaviours from state s



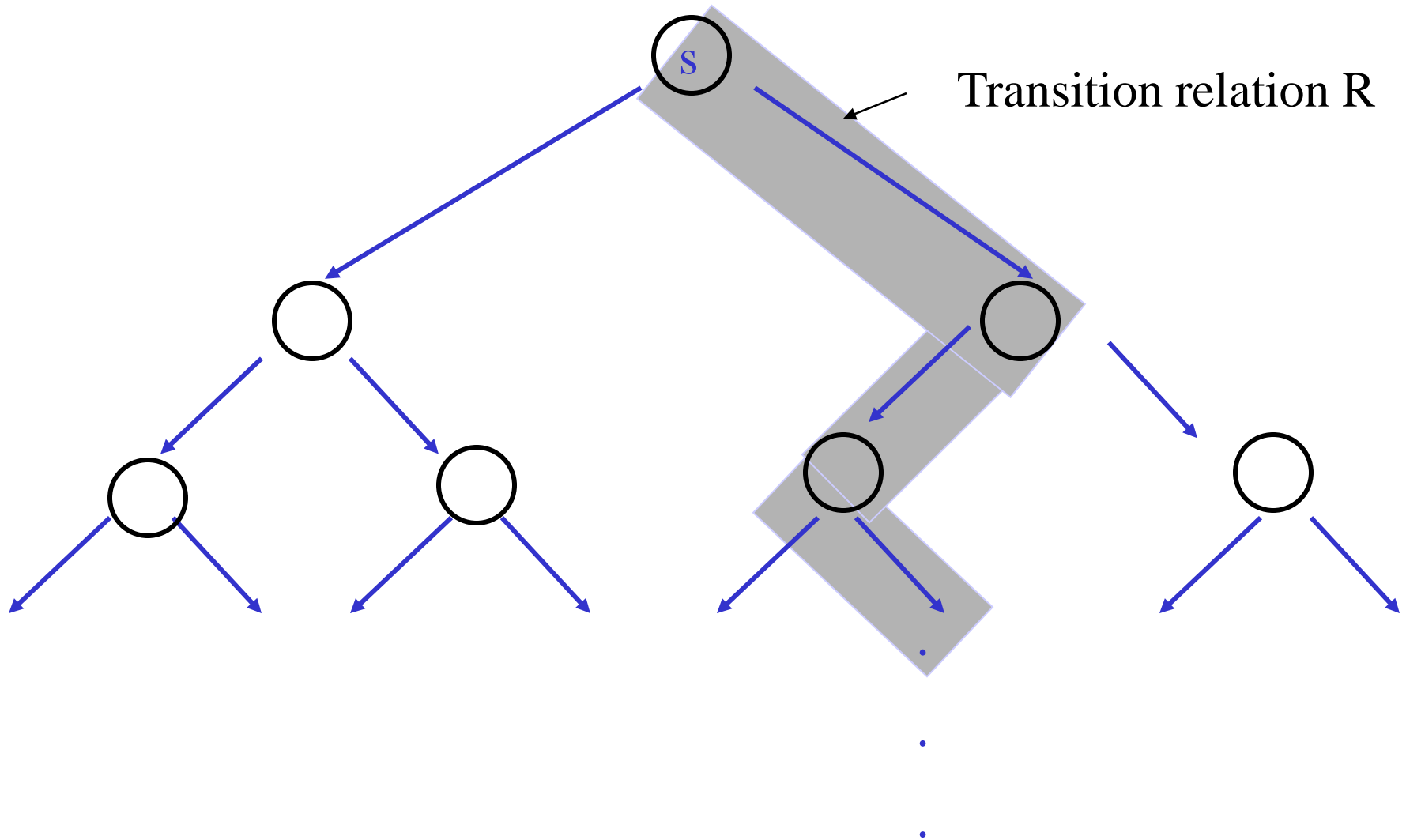
.

.

.

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 $S \rightarrow$ set of atomic formulas that hold
in that state

Lars backwards ☺ finite Kripke structure

Path in M

Infinite sequence of states

$\pi = s_0 s_1 s_2 \dots$ such that

Path in M

$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$



R

$(s_0, s_1) \in R$

$(s_1, s_2) \in R$

etc

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)

CTL*

path quantifiers

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

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

$$F f = \text{true} \cup f$$

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

Example

$G \text{ (req} \rightarrow \text{F ack)}$

Example

$G(\text{req} \rightarrow F \text{ ack})$

A request will eventually lead to an
acknowledgement

liveness

linear

Example

$G (\text{req} \rightarrow \text{ack})$

A request
acknowledgment

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 \rightarrow

A request
acknowledged

liveness

linear

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

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 & \neg Ready))

Semantics

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

$$M, s \models f \quad f \text{ holds at state } s \text{ in } M$$

(and omit M if it is clear which M we are talking about)

$$M, \pi \models g \quad g \text{ holds for path } \pi \text{ in } M$$

Semantics

Back to syntax and write down each case

$$s \models a \quad a \text{ in } L(s) \quad (\text{atomic})$$

$$s \models \neg f \quad \text{not } (s \models f)$$

$$s \models f1 \vee f2 \quad s \models f1 \quad \text{or} \quad s \models f2$$

$$s \models E(g) \quad \text{Exists } \pi. \text{head}(\pi) = s \quad \text{and } \pi \models g$$

Semantics

Back to syntax and write down each case

$s \models a$ $a \text{ in } L(s)$ (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

$$s \models a$$

$$s \models \neg f \quad \text{not } (s \models f)$$

$$s \models f1 \vee f2 \quad s \models f1 \quad \text{or} \quad s \models f2$$

$$s \models E(g) \quad \text{Exists } \pi. \text{head}(\pi) = s \quad \text{and } \pi \models g$$

Semantics

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

$$\pi \models \neg g \qquad \text{not } (\pi \models g)$$

$$\pi \models g1 \vee g2 \qquad \pi \models g1 \text{ or } \pi \models g2$$

Semantics

$$\pi \models X \ g \qquad \text{tail}(\pi) \models g$$

$$\pi \models g1 \ U \ g2$$

$$\text{Exists } k \geq 0. \quad \text{drop } k \ \pi \models g2 \qquad \text{and}$$

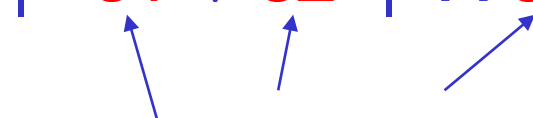
$$\text{Forall } 0 \leq j < k. \quad \text{drop } j \ \pi \models g1$$

(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) \vee E k

CTL* yes

CTL ?

E X X f

E (f U (g U j))

A (f U Xg)

A (f U g) \vee E k

Yes

CTL

Another view is that we just have the combined operators

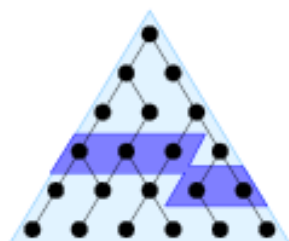
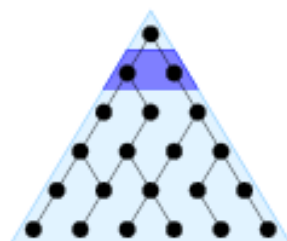
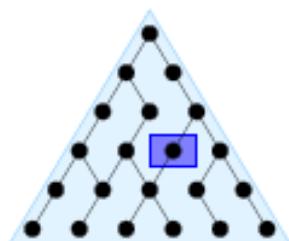
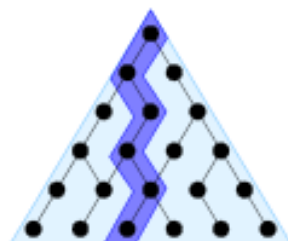
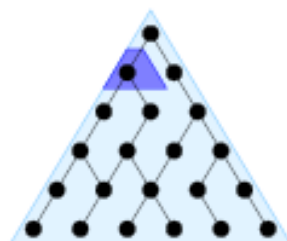
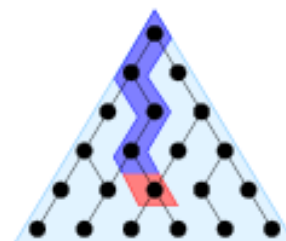
AU, AX, AF, AG and **EU, EX, EF, EG**
and only need to think about state formulas

A operators for necessity

E operators for possibility

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

CTL

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

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 & \neg 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**

$AG (Req \rightarrow AF Ack)$

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

$AG (Req \rightarrow A [Req \ U \ Ack])$

$\exists x \exists E (f \cup g)$

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.)

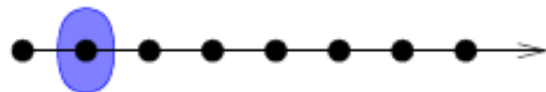
Restrict path formulas (compare with CTL*)

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



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)

$G \neg(\text{critical1} \ \& \ \text{critical2})$ mutex

$FG \text{ initialised}$ eventually stays initialised

$GF \text{ myMove}$ myMove will always eventually hold

$G (\text{req} \rightarrow F \text{ ack})$ request acknowledge pattern

LTl

Responsiveness (more examples)

$G (\text{req} \rightarrow XF \text{ack})$

$G (\text{req} \rightarrow X(\text{req} U \text{ack}))$

$G (\text{req} \rightarrow X((\text{req} \ \& \ \neg \text{ack}) U (\neg \text{req} \ \& \ \text{ack}))$

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

$AG \sim (strt \ \& \ EX \ E[\sim get \ \& \ \sim kill \ U \ get \ \& \ \sim kill \ \& \ EX \ E[\sim get \ \& \ \sim kill \ U \ get \ \& \ \sim kill \ \& \ E[\sim put \ U \ end] \ or \ E[\sim put \ \& \ \sim end \ U \ (put \ \& \ \sim end \ \& \ EX \ E[\sim put \ U \ end])]]])$

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