Model Checking I

What are LTL and CTL?
View circuit as a transition system

(dreq, q0, dack) → (dreq′, q0′, dack′)

q0′ = dreq
dack′ = dreq and (q0 or (not q0 and dack))
Idea

Transition system

+ special temporal logic

+ automatic checking algorithm
Exercise
(from example circuit)

(dreq, q0, dack) →
(dreq’, dreq, dreq and (q0 or (not q0 and dack)))

Draw state transition diagram

Q: How many states for a start?
Hint (partial answer)
Q: how many arrows should there be out of each state? Why so?
Complete the diagram

Write down the corresponding binary relation as a set of pairs of states
Another view

computation tree from a state
Unwinding further
Possible behaviours from state $s$

Transition relation $R$

Relation vs. Function?
path = possible run of the system

Transition relation R
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.
Model Checking

\[ G(p \rightarrow F q) \]

- Property
- Finite-state model
- Algorithm
- Counterexample

(Ken McMillan)
Netlist

dreq

q0

D

0

and

or

and

D

1

dack
input to SMV model checker

MC builds internal representation of transition system

MODULE main
VAR w1 : boolean;
VAR w2 : boolean;
VAR w3 : boolean;
VAR w4 : boolean;
VAR w5 : boolean;
VAR i0 : boolean;
VAR w6 : boolean;
VAR w7 : boolean;
VAR w8 : boolean;
VAR w9 : boolean;
VAR w10 : boolean;
DEFINE w4 := 0;
DEFINE w5 := i0;
ASSIGN init(w3) := w4;
ASSIGN next(w3) := w5;
DEFINE w7 := !(w3);
DEFINE w9 := 1;
DEFINE w10 := w5 & w6;
ASSIGN init(w8) := w9;
ASSIGN next(w8) := w10;
DEFINE w6 := w7 & w8;
DEFINE w2 := w3 | w6;
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

$\pi = s_0 \ s_1 \ s_2 \ \ldots \ \ s_t$
Path in M

\[ s_0 \rightarrow s_1 \rightarrow s_2 \ldots \]

\[ R \]

\[ (s_0, s_1) \in R \]

\[ (s_1, s_2) \in R \]

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 “sometimes”
X “nexttime”
U “until”

about a path
CTL* formulas (syntax)

path formulas

\[ f ::= s \mid \neg f \mid f_1 \lor f_2 \mid X f \mid f_1 U f_2 \]

state formulas (about an individual state)

\[ s ::= a \mid \neg s \mid s_1 \lor s_2 \mid E f \]

atomic formulas
Build up from core

\[ A f \quad \neg E \neg f \]

\[ F f \quad \text{true} \quad U f \]

\[ G f \quad \neg F \neg f \]
Example

G (req -> F ack)
Example

G (req -> F ack)

A request will eventually lead to an acknowledgement

liveness
linear
Example (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 hold.

$E \ (F \ (\text{Started} \ & \ \neg\text{Ready}))$
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

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

\[ s \not\models f \quad \text{not } (s \models f) \]

\[ s \models f_1 \lor f_2 \quad s \models f_1 \quad \text{or} \quad s \models f_2 \]

\[ s \models E \left( g \right) \quad \text{Exists } \pi. \text{ head}(\pi) = s \quad \text{and} \quad \pi \models g \]
Semantics

\[ \pi \models f \]

\[ s \models f \quad \text{and} \quad \text{head}(\pi) = s \]

\[ \pi \models \neg g \quad \text{not} \ (\pi \models g) \]

\[ \pi \models g_1 \lor g_2 \quad \pi \models g_1 \quad \text{or} \quad \pi \models g_2 \]
Semantics

\[\pi \vdash X \ g\]  \quad \text{tail}(\pi) \models g

\[\pi \models g_1 \cup g_2\]

Exists \( k \geq 0 \). \( \text{drop } k \ \pi \models g_2 \) and

Forall \( 0 \leq j < k \). \( \text{drop } j \ \pi \models g_1 \)

(note: I mean tail in the Haskell sense)
CTL

Branching time (remember upside-down tree)
Restrict path formulas (compare with CTL*)

\[ f ::= \neg f \mid s_1 \lor s_2 \mid X s \mid s_1 U s_2 \]

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

\[ f ::= s \mid \neg f \mid f_1 \lor f_2 \mid X f \mid f_1 U f_2 \]

state formulas (about an individual state)

\[ s ::= a \mid \neg s \mid s_1 \lor s_2 \mid E f \]

atomic formulas
Another view is that we just have the combined operators $\text{AU}, \text{AX}, \text{AF}, \text{AG}$ and $\text{EU}, \text{EX}, \text{EF}, \text{EG}$ and only need to think about state formulas.

A operators for necessity
E operators for possibility
f ::= atomic
| ¬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 & f2                     | A (f1 U f2)
| E (f1 U f2)
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

$$EF \ (\text{Started} \ & \ \neg \text{Ready})$$
Examples (Gordon)

If a request Req occurs, then it will eventually be acknowledged by Ack
Examples (Gordon)

If a request \( \text{Req} \) occurs, then it will eventually be acknowledged by \( \text{Ack} \)

\[
\text{AG} \ (\text{Req} \rightarrow \text{AF Ack})
\]
Examples (Gordon)

If a request \( \text{Req} \) 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*)

$$f ::= a \mid \neg f \mid f_1 \lor f_2 \mid X f \mid f_1 U f_2$$
Back to CTL* formulas (syntax)

path formulas

\[ f ::= s | \neg f | f_1 \lor f_2 | X f | f_1 U f_2 \]

state formulas (about an individual state)

\[ s ::= a | \neg s | s_1 \lor s_2 | E f \]

atomic formulas
LTL

finally \( P \)

\[ F P \]

globally \( P \)

\[ G P \]

next \( P \)

\[ X P \]

\( P \) until \( q \)

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

\[
\begin{align*}
G\neg(\text{critical1} \& \text{critical2}) & \quad \text{mutex} \\
\text{FG initialised} & \quad \text{eventually stays initialised} \\
\text{GF myMove} & \quad \text{myMove will always eventually hold} \\
G (\text{req} \rightarrow F \text{ ack}) & \quad \text{request acknowledge pattern}
\end{align*}
\]
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

**AF AG p**
In both

\[ AG (p \rightarrow AF q) \text{ in CTL} \]
\[ G(p \rightarrow F q) \text{ in LTL} \]
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

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

$$\text{AG} \sim (\text{strt} \& \text{EX } E[\sim \text{get} \& \sim \text{kill} \cup \text{get} \& \sim \text{kill} \& \text{EX } E[\sim \text{get} \& \sim \text{kill} \cup \text{get} \& \sim \text{kill} \& E[\sim \text{put} \cup \text{end}] \text{ or } E[\sim \text{put} \& \sim \text{end} \cup (\text{put} \& \sim \text{end} \& \text{EX } E[\sim \text{put} \cup \text{end}])]])$$
Next lecture

How to model check CTL formulas