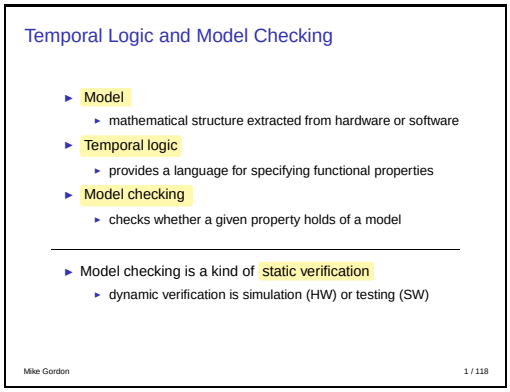
Title:	*Temporal Logic and Model Checking*
Lecturer:	**Mike Gordon**
Class:	**Computer Science Tripos, Part II**
Term:	**Easter Term 2011**
First Lecture:	**12:00 on Thursday, 28 April, 2011**
Location:	**Lecture Theatre 2, WGB**
Duration:	**Eight lectures**

Chapter 1

Introduction and overview



This course is entitled *Temporal Logic and Model Checking*, so we must explain what a model is, what temporal logic is and then what model checking is.

A model, as used here, is a particular kind of mathematical structure representing the functional behaviour of hardware or software. Models are extracted from programs or hardware designs and are intended to capture aspects of their functional behaviour.

Temporal logic is a formal logic for reasoning about temporal behaviour – i.e. behaviour that varies over time. It was originally devised by philosophers to help elucidate logical problems relating to time, but is now used in computer science to express and verify properties of models. Temporal logic consists of a specification language – the sentences of the logic – and a deductive system for proving theorems (i.e. sentences that are true).

Model checking is the process of checking whether properties hold of models. Model checking algorithms (there are many) take as input a property and a model and either confirm that the property holds of the model or, usually, output a counterexample to show that it doesn't.

There are various kinds of model and several different property specification languages. Here we will look mostly at properties specified as sentences in temporal logic, and models represented using a next-state relation. However, we will also look at the verification field more generally and try to locate temporal logic model checking within it.

1.1 Models

The word "model" has many meanings, but for us it is a pair (S, R) consisting of a set S of *states* and a binary relation R on S, called the *transition relation*. I will write $R\ s\ s'$ to mean that s and s' are related by R.[1]

Models are used to represent the behaviour of hardware and software. We illustrate this informally with two examples which we will return to later.

1.1.1 A hardware example: `RCV`

This circuit in Fig 1.1 below was (I think) designed in the Computer Lab many years ago, possibly as part of the old Cambridge Ring. It implements some kind of handshake and the name `RCV` is a shortening of "`RECEIVER`", which maybe suggests what it did ... however, for our purposes, it is just a random example. The wires `dreq`, `q0`, `q0bar`, `a0`, `or0`, `a1` are 1-bit wide.

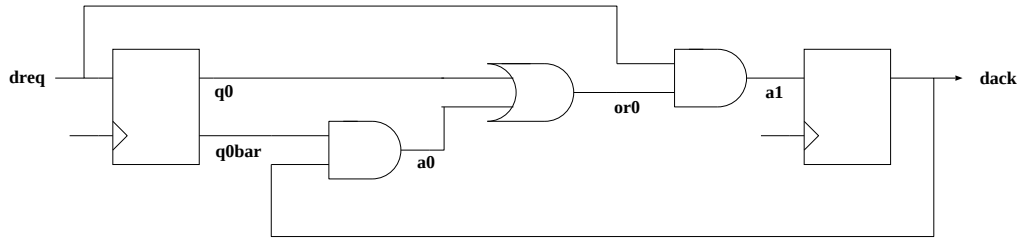


Figure 1.1: `RCV`

We want to construct a model representing the behaviour of `RCV`. This immediately raises some modelling issues: how accurately should we represent the behaviour. A really accurate model might represent the values on the wires as, say, continuously varying voltages and the behaviour of the components as analogue devices, maybe with temperature dependent transfer functions. We, however, will take a crude clocked digital view of behaviour. The state of the wires at any time will be 1 or 0. We assume that when the clock ticks the outputs of the two registers (i.e. `q0` and `dack`) are updated with the values being input (i.e. `dreq` and `a1`, respectively). Also the two outputs of the leftmost register are always complements of each other. The little unlabelled inputs to the two registers are clock lines, but these will not feature in our model. The combinational and-gate and two or-gates are assumed to

[1] This notation is a bit non standard: normally a relation R on S would be represented as a set of ordered pairs $R \subseteq S \times S$, and then one would write $(s, s') \in R$ to mean s and s' are related by R. We are treating relations as functions: $R : S \to (S \to \mathbb{B})$, where $\mathbb{B} = \{true, false\}$. When we write $R\ s\ s'$ we mean that this equals *true*, i.e. $R\ s\ s' = true$. This way of regarding relations is just a matter of style aimed at minimising brackets.

have zero delay so, using the italicised names of wires as variables that range over the wire values, we have the Boolean equations below, which show that values of $q0bar$, $a0$, $or0$ and $a1$ are determined by the values of $q0$ and $dack$ (this should also be obvious from the circuit diagram).

$$\begin{array}{lcl} q0bar & = & \neg q0 \\ a0 & = & q0bar \wedge dack \\ or0 & = & q0 \vee a0 \\ a1 & = & dreq \vee or0 \end{array}$$

A state of `RCV` can thus be modelled by a triple of Booleans $(dreq, q0, dack)$, thus we can define a model $(S_{\texttt{RCV}}, R_{\texttt{RCV}})$ by defining the set of states by:

$$S_{\texttt{RCV}} = \mathbb{B} \times \mathbb{B} \times \mathbb{B}$$

and the transition relation by:

$$\begin{array}{l} R_{\texttt{RCV}}\ (dreq, q0, dack)\ (dreq', q0', dack') = \\ (q0' = dreq) \wedge (dack' = (dreq \wedge (q0 \vee (\neg q0 \wedge dack)))) \end{array}$$

The use, as above, of primed variables for the next-state values of the unprimed variables with the same name is a common convention.

Notice that the transition relation $R_{\texttt{RCV}}$ is not a function because the value of $dreq$ is not determined. This is called *input non-determinism*, because `dreq` is an input whose value is determined by 'the environment', not the state of the registers. For any state, there are different successor states corresponding to different values input on `dreq`.

1.1.2 A software example: `DIV`

You might recognise the little program `DIV` in Fig. 1.2 below.

```
0:  R:=X;
1:  Q:=0;
2:  WHILE Y≤R DO
3:  (R:=R-Y;
4:   Q:=Q+1)
5:
```

Figure 1.2: `DIV`

A model $(S_{\texttt{DIV}}, R_{\texttt{DIV}})$ corresponding to `DIV` is obtained by taking a state to be (pc, x, y, r, q) where $pc \in \{0, 1, 2, 3, 4, 5\}$ is the *program counter*, which indicates

the line of the program that is about to be executed, and x, y, r and q are the values of the program variables `X`, `Y`, `R` and `Q`, respectively.

Assuming program variables range over the integers $\mathbb{Z}$ ($\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$) and $[m..n] = \{m, m{+}1, \ldots, n\}$, then:

$$S_{\texttt{DIV}} = [0..5] \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$$

and the transition relation $R_{\texttt{DIV}}$ is characterised by:

$$\begin{array}{l} R_{\texttt{DIV}}\ (0, x, y, r, q)\ (1, x, y, x, q) \\ R_{\texttt{DIV}}\ (1, x, y, r, q)\ (2, x, y, r, 0) \\ R_{\texttt{DIV}}\ (2, x, y, r, q)\ ((\mathit{if}\ y{\leq}r\ \mathit{then}\ 3\ \mathit{else}\ 5), x, y, r, q) \\ R_{\texttt{DIV}}\ (3, x, y, r, q)\ (4, x, y, (r{-}y), q) \\ R_{\texttt{DIV}}\ (4, x, y, r, q)\ (3, x, y, r, (q{+}1)) \end{array}$$

which is just a compact way of writing:

$$\begin{array}{l} (\forall x\ y\ r\ q.\ R_{\texttt{DIV}}\ (0, x, y, r, q)\ (1, x, y, x, q)) \\ \wedge \\ (\forall x\ y\ r\ q.\ R_{\texttt{DIV}}\ (1, x, y, r, q)\ (2, x, y, r, 0)) \\ \wedge \\ (\forall x\ y\ r\ q.\ R_{\texttt{DIV}}\ (2, x, y, r, q)\ ((\mathit{if}\ y{\leq}r\ \mathit{then}\ 3\ \mathit{else}\ 5), x, y, r, q)) \\ \wedge \\ (\forall x\ y\ r\ q.\ R_{\texttt{DIV}}\ (3, x, y, r, q)\ (4, x, y, (r{-}y), q)) \\ \wedge \\ (\forall x\ y\ r\ q.\ R_{\texttt{DIV}}\ (4, x, y, r, q)\ (3, x, y, r, (q{+}1))) \end{array}$$

This doesn't specify any transitions from the last line (`5`) of the program. What to do for the last line is mainly a matter of technical convenience. Possibilities are to have no transitions:

$$\forall pc'\ x'\ y'\ r'\ q'.\ \neg(R_{\texttt{DIV}}\ (5, x, y, r, q)\ (pc', x', y', r', q'))$$

or, if we want the transition relation to be total, to loop:

$$\forall x\ y\ r\ q.\ R_{\texttt{DIV}}\ (5, x, y, r, q)\ (5, x, y, r, q)$$

We will discuss later how to automatically derive a model of a program – i.e. a transition relation – from a formal semantics of the programming language (this is quite straightforward given the right kind of semantics).

Note that $R_{\texttt{DIV}}$ is *deterministic* in that for any state s there is at most one state s' such that $R_{\texttt{DIV}}\ s\ s'$. Deterministic models normally arise from sequential programs. Programs with concurrency can give rise to non-deterministic models (reflecting different interleavings). An example with concurrency is the simple program below, called `JM1`, which is adapted from Jhala and Majumdar's highly recommended tutorial "Software Model Checking" [1, Fig. 1].

```
Thread 1                          Thread 2
0:  IF LOCK=0 THEN LOCK:=1;       0:  IF LOCK=0 THEN LOCK:=1;
1:  X:=1;                         1:  X:=2;
2:  IF LOCK=1 THEN LOCK:=0;       2:  IF LOCK=1 THEN LOCK:=0;
3:                                3:
```

Figure 1.3: `JM1` (from Fig. 1 in Jhala & Majumdar's tutorial)

This has two threads executing in parallel and has behaviour represented by a non-deterministic model with state $(pc_1, pc_2, lock, x)$, where pc_i is the value of the program counter of thread i (where $i = 1$ or $i = 2$). Thus:

$$S_{\texttt{JM1}} \ = \ [0..3] \times [0..3] \times \mathbb{Z} \times \mathbb{Z}$$

For an example this simple one can specify the transition relation 'by inspection'.

$$\begin{array}{ll}
R_{\texttt{JM1}}\ (0, pc_2, 0, x)\ (1, pc_2, 1, x) & R_{\texttt{JM1}}\ (pc_1, 0, 0, x)\ (pc_1, 1, 1, x) \\
R_{\texttt{JM1}}\ (1, pc_2, lock, x)\ (2, pc_2, lock, 1) & R_{\texttt{JM1}}\ (pc_1, 1, lock, x)\ (pc_1, 2, lock, 2) \\
R_{\texttt{JM1}}\ (2, pc_2, 1, x)\ (3, pc_2, 0, x) & R_{\texttt{JM1}}\ (pc_1, 2, 1, x)\ (pc_1, 3, 0, x)
\end{array}$$

For more complex examples the model needs to be extracted using the semantics of the programming language. This will be discussed later.

Another source of non-determinism in models is abstraction: a model derived by abstraction be might be non-deterministic, even if the original program is purely sequential. This is discussed later.

1.2 Properties

Atomic properties are properties of states: if P is an atomic property of a model (S, R) then $P : S \to \mathbb{B}$. Atomic properties do not depend on the transition relation.

For example, for the model `RCV` we can define atomic properties `Dreq`, `Q0` and `Dack` that are true if the corresponding components of the state are *true*. For hardware examples like `RCV` we will write 1, 0 for *true*, *false*, respectively and say that a wire

is *high* if it has value 1 (i.e. *true*) and is *low* if it has value 0 (i.e. *false*). We will use capitalised names in `teletype font` for properties. Thus:

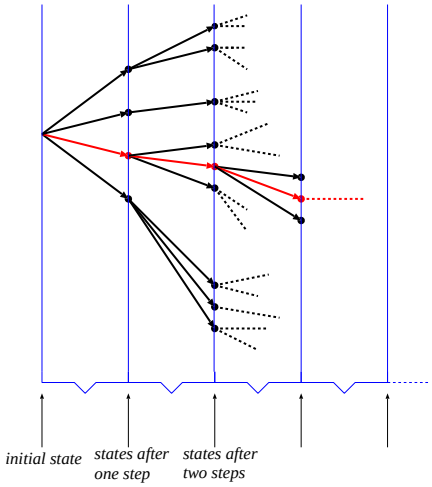
$$\begin{aligned} \texttt{Dreq}(dreq, q0, dack) &= (dreq = 1) \\ \texttt{Q0}(dreq, q0, dack) &= (q0 = 1) \\ \texttt{Dack}(dreq, q0, dack) &= (dack = 1) \end{aligned}$$

Examples of atomic properties for the model `DIV` are:

$$\begin{aligned} \texttt{AtEnd}\ (pc, x, y, r, q) &= (pc = 5) \\ \texttt{InLoop}\ (pc, x, y, r, q) &= (pc \in \{3, 4\}) \\ \texttt{YleqR}\ (pc, x, y, r, q) &= (y \leq r) \\ \texttt{Invariant}\ (pc, x, y, r, q) &= (x = r + (y \times q)) \end{aligned}$$

`AtEnd` is true of states at the end of the program, `InLoop` is true if the program counter is inside the body of the While-loop, `YleqR` is true of states where $y \leq r$ and `Invariant` is true of states where $x = r + (y \times q)$.

Atomic properties are true or false of individual states. General properties depend on the whole behaviour specified by the transition relation. The behaviour of a model (S, R) starting from an initial state $s \in S$ can be visualised as a tree:



This is called a *computation tree*. If the model is deterministic, then the tree will be linear – i.e. will just be a *path*, where a path is defined to be a sequence of states $s_0 s_1 s_2 \cdots$ such that for all i: $r\ s_i\ s_{i+1}$. Paths are also called *traces*.

Properties can be defined on paths (examples soon) and thus paths are a key concept for temporal logic and model checking. Sometimes paths are allowed to be finite and sometimes they are required to be infinite. We will require paths to be infinite because it makes some technical details nicer for us (but for other purposes the

details can be nicer if finite paths are allowed and, furthermore, finite paths are important in some practical applications). We generally use π to range over paths, which we represent as functions from the natural numbers $\mathbb{N}$ to states. Thus a path of a model (S, R) is a function $\pi : \mathbb{N} \to S$, and hence we write $\pi(i)$ for the ith element of π.

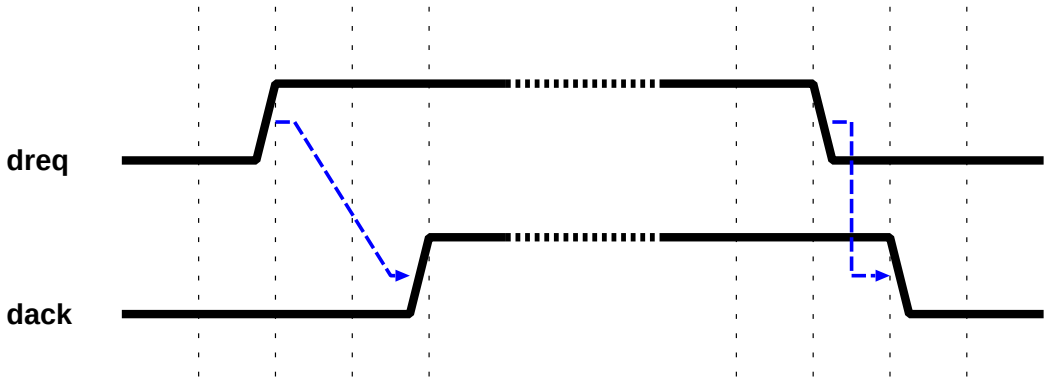
We define a predicate Path so that $\mathsf{Path}\ R\ s\ \pi$ is true if and only if π is a path starting at s:

$$\mathsf{Path}\ R\ s\ \pi\ =\ (\pi(0) = s)\ \wedge\ \forall i.\ R\ (\pi(i))\ (\pi(i{+}1))$$

Mathematically Path is a function:

$$\mathsf{Path} : \underbrace{(S \to S \to \mathbb{B})}_{\text{transition relation}} \to S \to \underbrace{(\mathbb{N} \to S)}_{\text{path}} \to \mathbb{B}$$

Many properties of the behaviour of models can be expressed in terms of paths. For example, consider the following timing diagram for the hardware model `RCV`:



Here are two possible properties, roughly expressing a fragment of handshake behaviour. The first property below corresponds to the left dotted arrow in the diagram and the second property corresponds to the right dotted arrow.

- If `dreq` rises, then it continues high, until it is acknowledged by a rise on `dack`.
- If `dreq` falls, then it will continue low until `dack` false.

To formalise these properties we need a *property language* such as temporal logic. We shall cover several different temporal logics in this course.

Here are some example properties for the software model `DIV`.

- On every path if `AtEnd` is true then `Invariant` is true and `YleqR` is not true.

- On every path there is a state where `AtEnd` it true.

For those of you familiar with the `DIV` example from the course on Hoare logic, note that these properties correspond to partial and total correctness, respectively. However, one can have properties than do not correspond to anything expressible in Hoare logic, for example:

- On any path if there exists a state where `YleqR` is true then there is also a state where `InLoop` is true.

An example property of the `JM1` example is:

- If initially `LOCK` $= 0$ and `X` $= 0$ then the model can never get into a state in which $pc_1 = 1$ and $pc_2 = 1$.

1.3 Model checking

Model checking is checking that a model has a given property. The model is derived from a circuit or program, as informally illustrated above, and the property is supplied by a verification engineer, usually in a property language like temporal logic (though there are other ways of capturing properties, e.g. using in graphical interface).

Model checking was initially applied to hardware and then later to software. Although the general description above doesn't say so, model checking is normally understood to be an automatic method. The inventors of both the concept and the first automatic algorithms for model checking are Edmund M. Clarke, E. Allen Emerson and Joseph Sifakis, who jointly won the 2007 Turing Award. Clarke's student Ken McMillan made a major advance by showing how to represent sets of states 'symbolically' using Binary Decision Diagrams (discussed later). For this he won the 2010 CAV (Computer-Aided Verification) Award, with the citation "for a series of fundamental contributions resulting in significant advances in scalability of model checking tools".

In keeping with the title of this course, we discuss in most detail model checking of properties stated in temporal logic. However, we shall also compare model checking with some other verification methods, including theorem proving (which can verify properties of models that cannot be expressed in standard temporal logics) and simulation (which is the normal verification method used by engineers).

Chapter 2

Temporal logic

The philosopher A. N. Prior originally devised temporal logic – he called it "Tense Logic" – to study "the relationship between tense and modality attributed to the Megarian philosopher Diodorus Cronus (ca. 340-280 BCE)".[1] The use of temporal logic in Computer Science is normally credited to the late Amir Pnueli, who won the 1996 Turing Award "For seminal work introducing temporal logic into computing science and for outstanding contributions to program and systems verification." In fact others, including Vaughan Pratt and Rod Burstall, had used ideas related to temporal logic in computer science earlier, but it was Pnueli who really launched the area and who developed the principles underlying current applications.

As the name suggests, temporal logic is a kind of logic consisting of a language defining temporal formulas and a deductive system for proving true formulas. In this course, we will mainly concentrate on the language(s) of temporal logic, since rather than prove formulas deductively, we will check their truth in models. However, the deductive systems of temporal logic have important applications in Computer Science – indeed such applications were what Pnueli originally pioneered [2].

There is not just one temporal logic: both philosophers and computer scientists have devised, and continue to devise, new temporal logics. However, we will concentrate on the two most widely used in computer science: *linear temporal logic* (LTL) and *computation true logic* (CTL). Model checking was originally developed for CTL, but LTL (which is what Pnueli mainly worked with) is now perhaps more widely used. Model checking algorithms from CTL are simpler and more efficient than those for LTL, however LTL provides a more natural property language form many applications. Both LTL and CTL are still widely used and understanding the tradeoffs between them is one topic in this course [3].

[1] `http://plato.stanford.edu/entries/logic-temporal/`

Bibliography

[1] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, October 2009.

[2] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[3] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 1–22, London, UK, 2001. Springer-Verlag.

Chapter 3

Appendix: slides

Temporal Logic and Model Checking

- Model
  - mathematical structure extracted from hardware or software
- Temporal logic
  - provides a language for specifying functional properties
- Model checking
  - checks whether a given property holds of a model

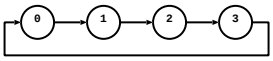
- Model checking is a kind of static verification
  - dynamic verification is simulation (HW) or testing (SW)

Models

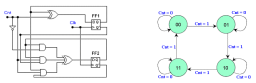
- A model is (for now) specified by a pair (S, R)
  - S is a set of *states*
  - R is a *transition relation*
- Models will get more components later
  - (S, R) could be called a pre-model ... but we won't bother
- $R\ s\ s'$ means s' can be reached from s in one step
  - here $R : S \to (S \to \mathbb{B})$ (where $\mathbb{B} = \{true, \mathsf{F}\}$)
  - more conventional to have $R \subseteq S \times S$, which is equivalent

A simple example model

- A simple model: $(\underbrace{\{0,1,2,3\}}_{S}, \underbrace{\lambda n\ n'.\ n' = n{+}1 (mod\ 4)}_{R})$
  - where "$\lambda x.\ \cdots x \cdots$" is the function mapping x to $\cdots x \cdots$
  - so $R\ n\ n' = (n' = n{+}1 (mod\ 4))$
  - e.g. $R\ 0\ 1 \wedge R\ 1\ 2 \wedge R\ 2\ 3 \wedge R\ 3\ 0$



- Might be extracted from:



[Acknowledgement: http://eelab.usyd.edu.au/digital_tutorial/part3/t-diag.html]

DIV: a software example

- Perhaps a familiar program:

```
0:  R:=X;
1:  Q:=0;
2:  WHILE Y≤R DO
3:  (R:=R-Y;
4:   Q:=Q+1)
5:
```

- State (pc, x, y, r, q)
  - $pc \in \{0,1,2,3,4,5\}$ program counter
  - $x, y, r, q \in \mathbb{Z}$ are the values of X, Y, R, Q
- Model $(S_{\mathtt{DIV}}, R_{\mathtt{DIV}})$ where:

$S_{\mathtt{DIV}} = [0..5] \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$

$R_{\mathtt{DIV}}\ (pc, x, y, r, q)\ (pc', x', y', r', q') =$

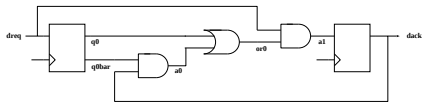
$(pc = 0) \Rightarrow ((pc', x', y', r', q') = (1, x, y, x, q)) \wedge$
$(pc = 1) \Rightarrow ((pc', x', y', r', q') = (2, x, y, r, 0)) \wedge$
$(pc = 2) \Rightarrow ((pc', x', y', r', q') =$
$\quad if\ y \leq r\ then\ (3, x, y, r, q)\ else\ (5, x, y, r, q)) \wedge$
$(pc = 3) \Rightarrow ((pc', x', y', r', q') = (4, x, y, (r - y), q)) \wedge$
$(pc = 4) \Rightarrow ((pc', x', y', r', q') = (3, x, y, r, (q{+}1))$

Deriving a transition relation from a state machine

- State machine transition function : $\delta : I \times S \to I$
  - I is a set of inputs
- State transition relation : $R\ (i, s)\ (i', s') = (s' = \delta(s, i))$
  - i' arbitrary: determined by environment not machine
- Deterministic machine, non-deterministic transition relation
  - inputs unspecified (determined by environment)
  - so called "input non-determinism"

RCV: a hardware model

- Part of a handshake circuit:



- State represented by a triple of Booleans $(dreq, q0, dack)$
- Relationships between Boolean values on wires:

$$\begin{array}{lcl} q0bar & = & \neg q0 \\ a0 & = & q0bar \wedge dack \\ or0 & = & q0 \vee a0 \\ a1 & = & dreq \vee or0 \end{array}$$

- A model of RCV is $(S_{\mathtt{RCV}}, R_{\mathtt{RCV}})$ where:

$$S_{\mathtt{RCV}} = \mathbb{B} \times \mathbb{B} \times \mathbb{B}$$

$$R_{\mathtt{RCV}}\ (dreq, q0, dack)\ (dreq', q0', dack') = (q0' = dreq) \wedge (dack' = (dreq \wedge (q0 \vee (\neg q0 \wedge dack))))$$

Some comments

- $R_{\mathtt{RCV}}$ is non-deterministic and total
  - $R_{\mathtt{RCV}}\ (1,1,1)\ (0,1,1)$ and $R_{\mathtt{RCV}}\ (1,1,1)\ (1,1,1)$ (where $1 = true$ and $0 = \mathsf{F}$)
  - $R_{\mathtt{RCV}}\ (dreq, q0, dack)\ (dreq', dreq, (dreq \wedge (q0 \vee dack)))$
- $R_{\mathtt{DIV}}$ is deterministic and partial
  - at most one successor state
  - no successor when $pc = 5$
- Non-deterministic models are very common, e.g. from:
  - asynchronous hardware
  - parallel software (more than one thread)
- Can extend any transition relation R to be total:

$$R_{total}\ s\ s' = R\ s\ s' \wedge (\neg(\exists s''.\ R\ s\ s'') \Rightarrow (s' = s))$$

  - sometimes totality required (e.g. in the book *Model Checking* by Clarke et. al)

JM1: a non-deterministic software example

- From Jhala and Majumdar's tutorial:

```
Thread 1                              Thread 2
0:  IF LOCK=0 THEN LOCK:=1; 0:  IF LOCK=0 THEN LOCK:=1;
1:  X:=1;                   1:  X:=2;
2:  IF LOCK=1 THEN LOCK:=0; 2:  IF LOCK=1 THEN LOCK:=0;
3:                          3:
```

- Two program counters, state: $(pc_1, pc_2, lock, x)$

$$S_{\mathtt{JM1}} = [0..3] \times [0..3] \times \mathbb{Z} \times \mathbb{Z}$$

$$\begin{array}{ll} R_{\mathtt{JM1}}\ (0, pc_2, 0, x) & (1, pc_2, 1, x) \\ R_{\mathtt{JM1}}\ (1, pc_2, lock, x) & (2, pc_2, lock, 1) \\ R_{\mathtt{JM1}}\ (2, pc_2, 1, x) & (3, pc_2, 0, x) \\ R_{\mathtt{JM1}}\ (pc_1, 0, 0, x) & (pc_1, 1, 1, x) \\ R_{\mathtt{JM1}}\ (pc_1, 1, lock, x) & (pc_1, 2, lock, 2) \\ R_{\mathtt{JM1}}\ (pc_1, 2, 1, x) & (pc_1, 3, 0, x) \end{array}$$

- Not-deterministic:

$$R_{\mathtt{JM1}}(0,0,0,x)(1,0,1,x)$$
$$R_{\mathtt{JM1}}(0,0,0,x)(0,1,1,x)$$

- Not so obvious that $R_{\mathtt{JM1}}$ is a correct model

Atomic properties (properties of states)

- Atomic properties are true or false of individual states
  - an atomic property P is a function $\mathtt{P} : S \to \mathbb{B}$
  - can also be regarded as a subset of state: $\mathtt{P} \subseteq S$
- Example atomic properties of RCV (where $1 = true$ and $0 = \mathsf{F}$)

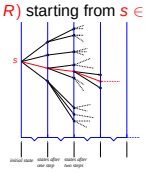
$$\begin{array}{lcl} \mathtt{Dreq}(dreq, q0, dack) & = & (dreq = 1) \\ \mathtt{NotQ0}(dreq, q0, dack) & = & (q0 = 0) \\ \mathtt{Dack}(dreq, q0, dack) & = & (dack = 1) \\ \mathtt{NotDreqAndQ0}(dreq, q0, dack) & = & (dreq{=}0) \wedge (q0{=}1) \end{array}$$

- Example atomic properties of DIV

$$\begin{array}{lcl} \mathtt{AtStart}\ (pc, x, y, r, q) & = & (pc = 0) \\ \mathtt{AtEnd}\ (pc, x, y, r, q) & = & (pc = 5) \\ \mathtt{InLoop}\ (pc, x, y, r, q) & = & (pc \in \{3, 4\}) \\ \mathtt{YleqR}\ (pc, x, y, r, q) & = & (y \leq r) \\ \mathtt{Invariant}\ (pc, x, y, r, q) & = & (x = r + (y \times q)) \end{array}$$

Model behaviour viewed as a computation tree

- Atomic properties are true or false of individual states
- General properties are true or false of whole behaviour
- Behaviour of (S, R) starting from $s \in S$ as a tree:



- A path is shown in red
- Properties may look at all paths, or just a single path
  - CTL: Computation Tree Logic (all paths from a state)
  - LTL: Linear Temporal Logic (a single path)

Paths

- A path of (S, R) is represented by a function $\pi : \mathbb{N} \to S$
  - $\pi(i)$ is the i th element of π (first element is $\pi(0)$)
  - might sometimes write $\pi\ i$ instead of $\pi(i)$
  - $\pi|i$ is the i-th tail of π so $\pi|i(n) = \pi(i+n)$
  - successive states in a path must be related by R
- Path $R\ s\ \pi$ is true if and only if π is a path starting at s:

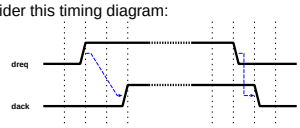
  $\text{Path}\ R\ s\ \pi\ =\ (\pi(0) = s)\ \wedge\ \forall i.\ R\ (\pi(i))\ (\pi(i+1))$

  where:

  $\text{Path} : \underbrace{(S \to S \to \mathbb{B})}_{\text{transition relation}} \to \underbrace{S}_{\text{initial state}} \to \underbrace{(\mathbb{N} \to S)}_{\text{path}} \to \mathbb{B}$

RCV: example hardware properties

- Consider this timing diagram:



- Two handshake properties representing the diagram:
  - following a rising edge on `dreq`, the value of `dreq` remains 1 (i.e. *true*) until it is acknowledged by a rising edge on `dack`
  - following a falling edge on `dreq`, the value on `dreq` remains 0 (i.e. F) until the value of `dack` is 0
- A property language is used to formalise such properties

DIV: example program properties

```
0:  R:=X;
1:  Q:=0;
2:  WHILE Y≤R DO
3:  (R:=R-Y;
4:   Q:=Q+1)
5:
```

$\texttt{AtStart}\ (pc, x, y, r, q) \quad = (pc = 0)$
$\texttt{AtEnd}\ (pc, x, y, r, q) \quad = (pc = 5)$
$\texttt{InLoop}\ (pc, x, y, r, q) \quad = (pc \in \{3, 4\})$
$\texttt{YleqR}\ (pc, x, y, r, q) \quad = (y \leq r)$
$\texttt{Invariant}\ (pc, x, y, r, q) \quad = (x = r + (y \times q))$

- Example properties of the program `DIV`.
  - on every execution if `AtEnd` is true then `Invariant` is true and `YleqR` is not true
  - on every execution there is a state where `AtEnd` it true
  - on any execution if there exists a state where `YleqR` is true then there is also a state where `InLoop` is true
- Compare these with what is expressible in Hoare logic
  - execution: a path starting from a state satisfying `AtStart`

JM1: a non-deterministic program example

```
Thread 1                                Thread 2
0:  IF LOCK=0 THEN LOCK:=1; 0:  IF LOCK=0 THEN LOCK:=1;
1:  X:=1; 1:  X:=2;
2:  IF LOCK=1 THEN LOCK:=0; 2:  IF LOCK=1 THEN LOCK:=0;
3:  3:
```

$R_{\texttt{JM1}}\ (0, pc_2, 0, x) \quad (1, pc_2, 1, x)$
$R_{\texttt{JM1}}\ (1, pc_2, lock, x) \quad (2, pc_2, lock, 1)$
$R_{\texttt{JM1}}\ (2, pc_2, 1, x) \quad (3, pc_2, 0, x)$
$R_{\texttt{JM1}}\ (pc_1, 0, 0, x) \quad (pc_1, 1, 1, x)$
$R_{\texttt{JM1}}\ (pc_1, 1, lock, x) \quad (pc_1, 2, lock, 2)$
$R_{\texttt{JM1}}\ (pc_1, 2, 1, x) \quad (pc_1, 3, 0, x)$

- An atomic property:
  - $\texttt{NotAt11}(pc_1, pc_2, lock, x) = \neg((pc_1 = 1) \wedge (pc_2 = 1))$
- A non-atomic property:
  - all states reachable from $(0, 0, 0, 0)$ satisfy `NotAt11`
  - this is an example of a reachability property

Reachability

- $R\ s\ s'$ means s' reachable from s in one step
- $R^n\ s\ s'$ means s' reachable from s in n steps

  $R^0\ s\ s' \quad = (s = s')$
  $R^{n+1}\ s\ s' \quad = \exists s''.\ R\ s\ s'' \wedge R^n\ s''\ s'$
- $R^*\ s\ s'$ means s' reachable from s in finite steps

  $R^*\ s\ s' = \exists n.\ R^n\ s\ s'$
- Note: $R^*\ s\ s' \Leftrightarrow \exists \pi\ n.\ \text{Path}\ R\ s\ \pi \wedge (s' = \pi(n))$
- The set of states reachable from s is $\{s' \mid R^*\ s\ s'\}$
- Verification problem: all states reachable from s satisfy p
  - verify truth of $\forall s'.\ R^*\ s\ s' \Rightarrow p(s')$
  - e.g. all states reachable from $(0, 0, 0, 0)$ satisfy `NotAt11`
  - i.e. $\forall s'.\ R^*_{\texttt{JM1}}\ (0, 0, 0, 0)\ s' \Rightarrow \texttt{NotAt11}(s')$

Model checking reachability properties

- Assume a model (S, R)
- Assume also a set $S_0 \subseteq S$ of *initial states*
- Assume also a set AP of atomic properties
  - if $p \in AP$ then $p : S \to \mathbb{B}$
  - $\mathsf{T}, \mathsf{F} \in AP$ where $\forall s \in S.\mathsf{T}(s)$=*true* and $\forall s \in S.\mathsf{F}(s)$=*false*
- A *Kripke structure* is a tuple (S, S_0, R, AP)
  - often the term "model" is used for a Kripke structure
  - i.e. a model is (S, S_0, R, AP) rather than just (S, R)
  - sometimes AP omitted: one says "Kripke structure over AP"
- Model checking computes whether $(S, S_0, R, AP) \models \phi$
  - ϕ is a property expressed in a property language
  - informally $M \models \phi$ means "wff ϕ is true in model M"

Aside on models and Kripke structures

- Definition of "model" and "Kripke structure" varies
- Initially we defined a model to be (S, R)
- On previous slide a model was (S, R, S_0, AP)
- (S, R) or (S, R, S_0) sometimes called *transition systems*
- We called (S, R, S_0, AP) a Kripke structure
- Clarke et al. define a Kripke structure as (S, S_0, R, L)
  - AP a given set of "atomic propositions" interpreted by L
  - $L : S \to \mathcal{P}(AP)$
  - $AP_{\text{(this course)}} = \{(\lambda s.\ p \in L(s)) \mid p \in AP_{\text{(Clarke et al.)}}\}$

Minimal property language: ϕ is **GA**p where $p \in AP$

- Assume $M = (S, S_0, R, AP)$
- Reachable states of M are $\{s' \mid \exists s \in S_0.\ R^*\ s\ s'\}$
  - i.e. the set of states reachable from an initial state
  - define Reachable $M = \{s' \mid \exists s \in S_0.\ R^*\ s\ s'\}$
- Consider properties ϕ of form **GA**p where $p \in AP$
  - "**GA**" stands for "Globally Always"
- $M \models$ **GA**p means p true of all reachable states of M
- If $M = (S, S_0, R, AP)$ then $M \models \phi$ formally defined by:

$$M \models \mathbf{GA}p \Leftrightarrow \forall s'.\ s' \in \text{Reachable } M \Rightarrow p(s')$$

Model checking $M \models$ **GA**p

- $M \models \mathbf{GA}p \Leftrightarrow \forall s'.\ s' \in \text{Reachable } M \Rightarrow p(s')$
  $\Leftrightarrow \text{Reachable } M \subseteq \{s' \mid p(s')\}$

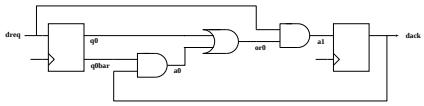
  so:
  - compute Reachable M i.e. compute $\{s' \mid \exists s \in S_0.\ R^*\ s\ s'\}$
  - check p true of all its members
- Let $\mathcal{S} = \{s' \mid \exists s \in S_0.\ R^*\ s\ s'\}$
- Compute $\mathcal{S}$ iteratively: $\mathcal{S} = \mathcal{S}_0 \cup \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_n \cup \cdots$
  - i.e. $\mathcal{S} = \bigcup_{n=0}^{\infty} \mathcal{S}_n$
  - where: $\mathcal{S}_0 = S_0$ (set of initial states)
  - and inductively: $\mathcal{S}_{n+1} = \mathcal{S}_n \cup \{s' \mid \exists s \in \mathcal{S}_n \wedge R\ s\ s'\}$
- Clearly $\mathcal{S}_0 \subseteq \mathcal{S}_1 \subseteq \cdots \subseteq \mathcal{S}_n \subseteq \cdots$
- Hence if $\mathcal{S}_m = \mathcal{S}_{m+1}$ then $\mathcal{S} = \mathcal{S}_m$
- Algorithm: compute $\mathcal{S}_0, \mathcal{S}_1, \ldots,$ until no change;
  check p holds of all members of computed set

compute $\mathcal{S}_0, \mathcal{S}_1, \ldots,$ until no change;
check p holds of all members of computed set

- Does the algorithm terminate?
  - yes, if set of states is finite, because then no infinite chains:
    $\mathcal{S}_0 \subset \mathcal{S}_1 \subset \cdots \subset \mathcal{S}_n \subset \cdots$
- How to represent $\mathcal{S}_0, \mathcal{S}_1, \ldots$?
  - explicitly (e.g. lists or something more clever)
  - symbolic expression
- Huge literature on calculating set of reachable states

Example: RCV

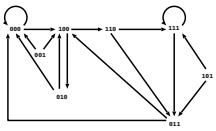
- Recall the handshake circuit:



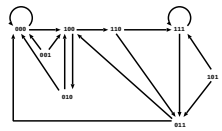
- State represented by a triple of Booleans $(dreq, q0, dack)$
- A model of RCV is M_{RCV} where:
  $M = (S_{\text{RCV}}, \{(1,1,1)\}, R_{\text{RCV}}, AP)$
  and
  $R_{\text{RCV}}\ (dreq, q0, dack)\ (dreq', q0', dack') =$
  $(q0' = dreq) \wedge (dack' = (dreq \wedge (q0 \vee dack)))$

RCV state transition diagram

- Possible states for RCV:
  $\{000, 001, 010, 011, 100, 101, 110, 111\}$
  where $b_2 b_1 b_0$ denotes state
  $\texttt{dreq} = b_2 \wedge \texttt{q0} = b_1 \wedge \texttt{dack} = b_0$
- Graph of the transition relation:



Computing Reachable M_{RCV}

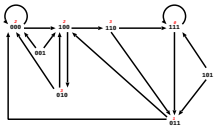


- Define:

$\mathcal{S}_0 = \{b_2b_1b_0 \mid b_2b_1b_0 \in \{111\}\}$

$\mathcal{S}_{i+1} = \mathcal{S}_i \cup \{s' \mid \exists s \in \mathcal{S}_i.\ R_{RCV}\ s\ s'\}$
$= \mathcal{S}_i \cup \{b'_2b'_1b'_0 \mid \exists b_2b_1b_0 \in \mathcal{S}_i.\ (b'_1 = b_2) \wedge (b'_0 = b_2 \wedge (b_1 \vee b_0))\}$

Computing Reachable M_{RCV} (continued)



- Compute:

$\mathcal{S}_0 = \{111\}$
$\mathcal{S}_1 = \{111\} \cup \{011\}$
$= \{111, 011\}$
$\mathcal{S}_2 = \{111, 011\} \cup \{000, 100\}$
$= \{111, 011, 000, 100\}$
$\mathcal{S}_3 = \{111, 011, 000, 100\} \cup \{010, 110\}$
$= \{111, 011, 000, 100, 010, 110\}$
$\mathcal{S}_i = \mathcal{S}_3 \quad (i > 3)$

- Hence Reachable $M_{RCV} = \{111, 011, 000, 100, 010, 110\}$

Model checking $M_{RCV} \models \mathbf{GA}p$

- $M = (S_{RCV}, \{111\}, R_{RCV}, AP)$
  - if $p \in AP$ then $p : S_{RCV} \to \mathbb{B}$
- To check $M_{RCV} \models \mathbf{GA}p$
  - compute Reachable $M_{RCV} = \{111, 011, 000, 100, 010, 110\}$
  - check Reachable $M_{RCV} \subseteq \{s \mid p(s)\}$, i.e. check:

$p(111) = true$
$p(011) = true$
$p(000) = true$
$p(100) = true$
$p(010) = true$
$p(110) = true$

Symbolic Boolean model checking of reachability

- Assume states are n-tuples of Booleans $(b_1, \ldots, b_n)$
  - $b_i \in \mathbb{B} = \{true, false\}$
  - $S = \mathbb{B}^n$, so S is finite: 2^n states
- Assume n distinct Boolean variables: v_1,...,v_n
  - e.g. if $n = 3$ then could have $v_1 = \mathtt{x}$, $v_2 = \mathtt{y}$, $v_3 = \mathtt{z}$
- Boolean formula $f(v_1, \ldots, v_n)$ represents a subset of S
  - $f(v_1, \ldots, v_n)$ only contains variables v_1,...,v_n
  - $f(b_1, \ldots, b_n)$ denotes result of substituting b_i for v_i
  - $f(v_1, \ldots, v_n)$ determines $\{(b_1, \ldots, b_n) | f(b_1, \ldots, b_n) \Leftrightarrow true\}$
- Example $\neg(\mathtt{x} = \mathtt{y})$ represents $\{(true, false), (false, true)\}$
- Transition relations also represented by Boolean formulae
  - e.g. R_{RCV} represented by:
    $(q0' = dreq) \wedge (dack' = (dreq \wedge (q0 \vee (\neg q0 \wedge dack))))$

Symbolically represent Boolean formulae as BDDs

- Key features of Binary Decision Diagrams (BDDs):
  - canonical (given a variable ordering)
  - efficient to manipulate
- Variables:

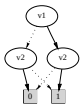
```
v   =  if v then 1 else 0
¬v  =  if v then 0 else 1
```

- Example: BDDs of variable v and ¬v

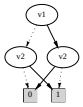
- Example: BDDs of v1 ∧ v2 and v1 ∨ v2

More BDD examples

- BDD of v1 = v2



- BDD of v1 ≠ v2

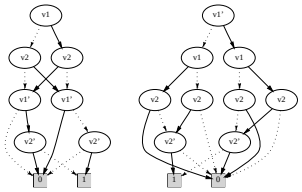


BDD of a transition relation

- BDDs of

  $(v1' = (v1 = v2)) \wedge (v2' = (v1 \neq v2))$

  with two different variable orderings



- **Exercise:** draw BDD of R_{RCV}

Standard BDD operations

- If formulae f_1, f_2 represents sets S_1, S_2, respectively
  then $f_1 \wedge f_2$, $f_1 \vee f_2$ represent $S_1 \cap S_2$, $S_1 \cup S_2$ respectively
- Standard algorithms compute boolean operation on BDDs
- Abbreviate $(v_1, \ldots, v_n)$ to $\vec{v}$
- If $f(\vec{v})$ represents S
  and $g(\vec{v}, \vec{v'})$ represents $\{(\vec{v}, \vec{v'}) \mid R\ \vec{v}\ \vec{v'})\}$
  then $\exists \vec{u}.\ f(\vec{u}) \wedge g(\vec{u}, \vec{v})$ represents $\{\vec{v} \mid \exists \vec{u}.\ \vec{u} \in S \wedge R\ \vec{u}\ \vec{v}\}$
- Can compute BDD of $\exists \vec{u}.\ h(\vec{u}, \vec{v})$ from BDD of $h(\vec{u}, \vec{v})$
  - e.g. BDD of $\exists v_1.\ h(v_1, v_2)$ is BDD of $h(\mathsf{T}, v_2) \vee h(\mathsf{F}, v_2)$
- From BDD of formula $f(v_1, \ldots, v_n)$ can compute $b_1, \ldots, b_n$ such that if $v_1 = b_1, \ldots, v_n = b_n$ then $f(b_1, \ldots, b_n) \Leftrightarrow \mathit{true}$
  - $b_1, \ldots, b_n$ is a satisfying assignment (SAT problem)
  - used for counterexample generation (see later)

Reachable States via BDDs

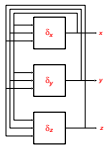
- Assume $M = (S, S_0, R, AP)$ and $S = \mathbb{B}^n$
- Represent R by Boolean formulae $g(\vec{v}, \vec{v'})$
- Iteratively define formula $f_n(\vec{v})$ representing $\mathcal{S}_n$

  $f_0(\vec{v})$ = formula representing S_0

  $f_{n+1}(\vec{v}) = f_n(\vec{v}) \vee (\exists \vec{u}.\ f_n(\vec{u}) \wedge g(\vec{u}, \vec{v}))$
- Let $\mathcal{B}_0$, $\mathcal{B}_R$ be BDDs representing $f_0(\vec{v})$, $g(\vec{v}, \vec{v'})$
- Iteratively compute BDDs $\mathcal{B}_n$ representing f_n

  $\mathcal{B}_{n+1} = \mathcal{B}_n \vee (\exists \vec{u}.\ \mathcal{B}_n[\vec{u}/\vec{v}] \wedge \mathcal{B}_R)[\vec{u}, \vec{v}/\vec{v}, \vec{v'}]$
  - efficient using (blue underlined) standard BDD algorithms (renaming, conjuction, disjunction, existential quantification)
  - BDD $\mathcal{B}_n$ only contains variables $\vec{v}$: represents $\mathcal{S}_n \subseteq S$
- At each iteration check $\mathcal{B}_{n+1} = \mathcal{B}_n$ efficient using BDDs
  - when $\mathcal{B}_{n+1} = \mathcal{B}_n$ can conclude $\mathcal{B}_n$ represents Reachable M
  - we call this BDD $\mathcal{B}_M$ in a later slide (i.e. $\mathcal{B}_M = \mathcal{B}_n$)

Example BDD optimisation: disjunctive partitioning



Three state machines in parallel

$\delta_x, \delta_y, \delta_z : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

- Transition relation (asynchronous interleaving semantics):

$$\begin{array}{l} R\ (x,y,z)\ (x',y',z') = \\ \quad (x' = \delta_x(x,y,z)\ \wedge\ y' = y\ \wedge\ z' = z) \vee \\ \quad (x' = x\ \wedge\ y' = \delta_y(x,y,z)\ \wedge\ z' = z) \vee \\ \quad (x' = x\ \wedge\ y' = y\ \wedge\ z' = \delta_z(x,y,z)) \end{array}$$

Avoiding building big BDDs

- Transition relation for three machines in parallel

$$\begin{array}{l} R(x,y,z)\ (x',y',z') = \\ \quad (x' = \delta_x(x,y,z)\ \wedge\ y' = y\ \wedge\ z' = z) \vee \\ \quad (x' = x\ \wedge\ y' = \delta_y(x,y,z)\ \wedge\ z' = z) \vee \\ \quad (x' = x\ \wedge\ y' = y\ \wedge\ z' = \delta_z(x,y,z)) \end{array}$$

- Recall symbolic iteration:

  $f_{n+1}(\vec{v}) = f_n(\vec{v}) \vee (\exists \vec{u}.\ f_n(\vec{u}) \wedge g(\vec{u}, \vec{v}))$
- For the 3-machine example this is (see next slide):

$$\begin{array}{l} f_{n+1}(x,y,z) \\ \quad = f_n(x,y,z) \vee (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},\overline{y},\overline{z}) \wedge R\ (\overline{x},\overline{y},\overline{z})\ (x,y,z)) \\ \quad = f_n(x,y,z) \quad \vee \\ \qquad (\exists \overline{x}.\ f_n(\overline{x},y,z)\ \wedge\ x = \delta_x(\overline{x},y,z)) \vee \\ \qquad (\exists \overline{y}.\ f_n(x,\overline{y},z)\ \wedge\ y = \delta_y(x,\overline{y},z)) \vee \\ \qquad (\exists \overline{z}.\ f_n(x,y,\overline{z})\ \wedge\ z = \delta_z(x,y,\overline{z})) \end{array}$$

- Don't need to calculate BDD of R!

Disjunctive partitioning

$$\begin{array}{ll} & \exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},\overline{y},\overline{z}) \wedge R\ (\overline{x},\overline{y},\overline{z})\ (x,y,z) \\ = & \exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},\overline{y},\overline{z}) \wedge ((x = \delta_x(\overline{x},\overline{y},\overline{z}) \wedge y = \overline{y} \wedge z = \overline{z}) \vee \\ & \qquad\qquad (x = \overline{x} \wedge y = \delta_y(\overline{x},\overline{y},\overline{z}) \wedge z = \overline{z}) \vee \\ & \qquad\qquad (x = \overline{x} \wedge y = \overline{y} \wedge z = \delta_z(\overline{x},\overline{y},\overline{z}))) \\ = & (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},\overline{y},\overline{z}) \wedge x = \delta_x(\overline{x},\overline{y},\overline{z}) \wedge y = \overline{y} \wedge z = \overline{z}) \vee \\ & (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},\overline{y},\overline{z}) \wedge x = \overline{x} \wedge y = \delta_y(\overline{x},\overline{y},\overline{z}) \wedge z = \overline{z}) \vee \\ & (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},\overline{y},\overline{z}) \wedge x = \overline{x} \wedge y = \overline{y} \wedge z = \delta_z(\overline{x},\overline{y},\overline{z})) \\ = & (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},y,z) \wedge x = \delta_x(\overline{x},y,z) \wedge y = \overline{y} \wedge z = \overline{z}) \vee \\ & (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(x,\overline{y},z) \wedge x = \overline{x} \wedge y = \delta_y(x,\overline{y},z) \wedge z = \overline{z}) \vee \\ & (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(x,y,\overline{z}) \wedge x = \overline{x} \wedge y = \overline{y} \wedge z = \delta_z(x,y,\overline{z})) \\ = & ((\exists \overline{x}.\ f_n(\overline{x},y,z) \wedge x = \delta_x(\overline{x},y,z)) \wedge (\exists \overline{y}.\ y = \overline{y}) \wedge (\exists \overline{z}.\ z = \overline{z})) \vee \\ & ((\exists \overline{x}.\ x = \overline{x}) \wedge (\exists \overline{y}.\ f_n(x,\overline{y},z) \wedge y = \delta_y(x,\overline{y},z)) \wedge (\exists \overline{z}.\ z = \overline{z})) \vee \\ & ((\exists \overline{x}.\ x = \overline{x}) \wedge (\exists \overline{y}.\ y = \overline{y}) \wedge (\exists \overline{z}.\ f_n(x,y,\overline{z}) \wedge z = \delta_z(x,y,\overline{z}))) \\ = & (\exists \overline{x}.\ f_n(\overline{x},y,z) \wedge x = \delta_x(\overline{x},y,z)) \vee \\ & (\exists \overline{y}.\ f_n(x,\overline{y},z) \wedge y = \delta_y(x,\overline{y},z)) \vee \\ & (\exists \overline{z}.\ f_n(x,y,\overline{z}) \wedge z = \delta_z(x,y,\overline{z})) \end{array}$$

Verification and counterexamples

- Typical safety question:
  - is property p true in all reachable states?
  - i.e. check $M \models \mathbf{GA}p$
  - i.e. is $\forall s.\ s \in \text{Reachable } M \Rightarrow p\ s$
- Check using BDDs
  - compute BDD $\mathcal{B}_M$ of Reachable M
  - compute BDD $\mathcal{B}_p$ of $p(\vec{v})$
  - check if BDD of $\mathcal{B}_M \Rightarrow \mathcal{B}_p$ is the single node $\boxed{1}$
- Valid because *true* represented by a unique BDD (canonical property)
- If BDD is not $\boxed{1}$ can get counterexample

Generating counterexamples

BDD algorithms can find satisfying assignments (SAT)

- $M = (S, S_0, R, AP)$ and $\mathcal{B}_0$, $\mathcal{B}_1$, ..., $\mathcal{B}_M$, $\mathcal{B}_R$, $\mathcal{B}_p$ as earlier
- Suppose $\mathcal{B}_M \Rightarrow \mathcal{B}_p$ is not $\boxed{1}$
- Must exist a state $s \in \text{Reachable } M$ such that $\neg(p\ s)$
- Let $\mathcal{B}_{\neg p}$ be the BDD representing $\neg(p\ \vec{v})$
- Iterate to find first n such that $\mathcal{B}_n \wedge \mathcal{B}_{\neg p}$
- Using SAT find $\vec{b}_n$ such that $(\mathcal{B}_n \wedge \mathcal{B}_{\neg p})[\vec{b}_n/\vec{v}]$
- Use SAT to find $\vec{b}_{n-1}$ such that $(\mathcal{B}_{n-1} \wedge \mathcal{B}_R[\vec{b}_n/\vec{v}'])[\vec{b}_{n-1}/\vec{v}]$
- For $0 \leq i \leq n$ find $\vec{b}_i$ such that $(\mathcal{B}_{i-1} \wedge \mathcal{B}_R[\vec{b}_i/\vec{v}'])[\vec{b}_{i-1}/\vec{v}]$
- $\vec{b}_0,\ldots,\vec{b}_i,\ldots,\vec{b}_n$ is a counterexample trace
- Sometimes can use partitioning to avoid constructing $\mathcal{B}_R$

Example (from an exam)

Consider a 3x3 array of 9 switches

Suppose each switch 1,2,...,9 can either be on or off, and that toggling any switch will automatically toggle all its immediate neighbours. For example, toggling switch 5 will also toggle switches 2, 4, 6 and 8, and toggling switch 6 will also toggle switches 3, 5 and 9.

(a) Devise a state space [4 marks] and transition relation [6 marks] to represent the behavior of the array of switches

You are given the problem of getting from an initial state in which even numbered switches are on and odd numbered switches are off, to a final state in which all the switches are off.

(b) Write down predicates on your state space that characterises the initial [2 marks] and final [2 marks] states.

(c) Explain how you might use a model checker to find a sequences of switches to toggle to get from the initial to final state. [6 marks]

You are not expected to actually solve the problem, but only to explain how to represent it in terms of model checking.

Solution

A state is a vector `(v0,v1,v2,v3,v4,v5,v6,v7,v8)`, where `vi` $\in \mathbb{B}$ (`vi` true iff switch number `i+1` is on)

A transition relation `Trans` is then defined by:

```
Trans(v0,v1,v2,v3,v4,v5,v6,v7,v8)(v0',v1',v2',v3',v4',v5',v6',v7',v8')
 = ((v0'=¬v0)∧(v1'=¬v1)∧(v2'=v2)∧(v3'=¬v3)∧(v4'=v4)∧
    (v5'=v5)∧(v6'=v6)∧(v7'=v7)∧(v8'=v8))      (toggle switch 1)
 ∨ ((v0'=¬v0)∧(v1'=¬v1)∧(v2'=¬v2)∧(v3'=v3)∧(v4'=¬v4)∧
    (v5'=v5)∧(v6'=v6)∧(v7'=v7)∧(v8'=v8))      (toggle switch 2)
 ∨ ((v0'=v0)∧(v1'=¬v1)∧(v2'=¬v2)∧(v3'=v3)∧(v4'=v4)∧
    (v5'=¬v5)∧(v6'=v6)∧(v7'=v7)∧(v8'=v8))     (toggle switch 3)
 ∨ ((v0'=¬v0)∧(v1'=v1)∧(v2'=v2)∧(v3'=¬v3)∧(v4'=¬v4)∧
    (v5'=v5)∧(v6'=¬v6)∧(v7'=v7)∧(v8'=v8))     (toggle switch 4)
 ∨ ((v0'=v0)∧(v1'=¬v1)∧(v2'=v2)∧(v3'=¬v3)∧(v4'=¬v4)∧
    (v5'=¬v5)∧(v6'=v6)∧(v7'=¬v7)∧(v8'=v8))    (toggle switch 5)
 ∨ ((v0'=v0)∧(v1'=v1)∧(v2'=¬v2)∧(v3'=v3)∧(v4'=¬v4)∧
    (v5'=¬v5)∧(v6'=v6)∧(v7'=v7)∧(v8'=¬v8))    (toggle switch 6)
 ∨ ((v0'=v0)∧(v1'=v1)∧(v2'=v2)∧(v3'=¬v3)∧(v4'=v4)∧
    (v5'=v5)∧(v6'=¬v6)∧(v7'=¬v7)∧(v8'=v8))    (toggle switch 7)
 ∨ ((v0'=v0)∧(v1'=v1)∧(v2'=v2)∧(v3'=v3)∧(v4'=¬v4)∧
    (v5'=v5)∧(v6'=¬v6)∧(v7'=¬v7)∧(v8'=¬v8))   (toggle switch 8)
 ∨ ((v0'=v0)∧(v1'=v1)∧(v2'=v2)∧(v3'=v3)∧(v4'=v4)∧
    (v5'=¬v5)∧(v6'=v6)∧(v7'=¬v7)∧(v8'=¬v8))   (toggle switch 9)
```


Solution (continued)

Predicates `Init`, `Final` characterising the initial and final states, respectively, are defined by:

```
Init(v0,v1,v2,v3,v4,v5,v6,v7,v8) =
 ¬v0 ∧ v1 ∧ ¬v2 ∧ v3 ∧ ¬v4 ∧ v5 ∧ ¬v6 ∧ v7 ∧ ¬v8

Final(v0,v1,v2,v3,v4,v5,v6,v7,v8) =
 ¬v0 ∧ ¬v1 ∧ ¬v2 ∧ ¬v3 ∧ ¬v4 ∧ ¬v5 ∧ ¬v6 ∧ ¬v7 ∧ ¬v8
```

Model checkers can find counter-examples to properties, and sequences of transitions from an initial state to a counter-example state. Thus we could use a model checker to find a trace to a counter-example to the property that

```
¬Final(v0,v1,v2,v3,v4,v5,v6,v7,v8)
```


Properties

- $\forall s \in S_0.\ R^*\ s\ \Rightarrow\ p\ s$ means p true in all reachable states
- Might want to verify other properties
  1. `DeviceEnabled` holds infinitely often along every path
  2. From any state it is possible to get to a state where `Restart` holds
  3. After a three or more consecutive occurrences of `Req` there will eventually be an `Ack`
- Temporal logic can express such properties
- There are several temporal logics in use
  - LTL is good for the first example above
  - CTL is good for the second example
  - PSL is good for the third example
- Model checking:
  - Emerson, Clarke & Sifakis: Turing Award 2008
  - widely used in industry: first hardware, later software

Temporal logic (originally called “tense logic”)

A. N. Prior
1914-1969

Originally devised for investigating: “the relationship between tense and modality attributed to the Megarian philosopher Diodorus Cronus (ca. 340-280 BCE)”.

Mary Prior, his wife, recalls “I remember his waking me one night [in 1953], coming and sitting on my bed, ... and saying he thought one could make a formalised tense logic”.

- Temporal logic: deductive system for reasoning about time
  - temporal formulae for expressing temporal statements
  - deductive system for proving theorems
- Temporal logic model checking
  - uses semantics to check truth of temporal formulae in models
- Temporal logic proof systems also important in CS
  - use pioneered by Amir Pnueli (1996 Turing Award)
  - not considered in this course

Recommended: `http://plato.stanford.edu/entries/prior/`

Temporal logic formulae (statements)

- Many different languages of temporal statements
  - linear time (LTL)
  - branching time (CTL)
  - finite intervals (SEREs)
  - industrial languages (PSL, SVA)
- Prior used linear time, Kripke suggested branching time:

  ... we perhaps should not regard time as a linear series ... there are several possibilities for what the next moment may be like - and for each possible next moment, there are several possibilities for the moment after that. Thus the situation takes the form, not of a linear sequence, but of a ‘tree’.
  [Saul Kripke, 1958 (aged 17, still at school)]
- CS issues different from philosophical issues
  - Moshe Vardi: “Branching vs. Linear Time: Final Showdown”

http://www.computer.org/portal/web/awards/Vardi

Moshe Vardi (aged 56, still at school)
www.computer.org

“For fundamental and lasting contributions to the development of logic as a unifying foundational framework and a tool for modeling computational systems”

2011 Harry H. Goode Memorial Award Recipient

Linear Temporal Logic (LTL)

- Grammar of *well formed formulae* (wff) ϕ

ϕ	$::=$	p	(Atomic formula: $p \in AP$)
	$\mid$	$\neg\phi$	(Negation)
	$\mid$	$\phi_1 \vee \phi_2$	(Disjunction)
	$\mid$	$\mathbf{X}\phi$	(successor)
	$\mid$	$\mathbf{F}\phi$	(sometimes)
	$\mid$	$\mathbf{G}\phi$	(always)
	$\mid$	$[\phi_1\ \mathbf{U}\ \phi_2]$	(Until)

- Details differ from Prior’s tense logic – but similar ideas
- Semantics define when ϕ true in model M
  - where $M = (S, R, S_0, AP)$ – a Kripke structure
  - notation: $M \models \phi$ means ϕ true in model M
  - model checking algorithms compute this (when decidable)

$M \models \phi$ means “wff ϕ is true in model M”

- If $M = (S, S_0, R, AP)$ then
  π is an M-path starting from s iff Path R s π
- If $M = (S, S_0, R, AP)$ then we define $M \models \phi$ to mean:
  ϕ is true on all M-paths starting from a member of S_0
- We will define $[\![\phi]\!]_M(\pi)$ to mean
  ϕ is true on the M-path π
- Thus $M \models \phi$ will be formally defined by:
  $M \models \phi \Leftrightarrow \forall \pi\ s.\ s \in S_0 \wedge \text{Path}\ R\ s\ \pi \Rightarrow [\![\phi]\!]_M(\pi)$
- It remains to actually define $[\![\phi]\!]_M$ for all wffs ϕ

Definition of $[\![\phi]\!]_M(\pi)$

- $[\![\phi]\!]_M(\pi)$ is the application of function $[\![\phi]\!]_M$ to path π
  - thus $[\![\phi]\!]_M : (\mathbb{N} \to S) \to \mathbb{B}$
- Let $M = (S, S_0, R, AP)$
  $[\![\phi]\!]_M$ is defined by structural induction on ϕ

$$
\begin{array}{lcl}
[\![p]\!]_M(\pi) & = & p(\pi\ 0) \\
[\![\neg\phi]\!]_M(\pi) & = & \neg([\![\phi]\!]_M(\pi)) \\
[\![\phi_1 \vee \phi_2]\!]_M(\pi) & = & [\![\phi_1]\!]_M(\pi)\ \vee\ [\![\phi_2]\!]_M(\pi) \\
[\![\mathbf{X}\phi]\!]_M(\pi) & = & [\![\phi]\!]_M(\pi\downarrow 1) \\
[\![\mathbf{F}\phi]\!]_M(\pi) & = & \exists i.\ [\![\phi]\!]_M(\pi\downarrow i) \\
[\![\mathbf{G}\phi]\!]_M(\pi) & = & \forall i.\ [\![\phi]\!]_M(\pi\downarrow i) \\
[\![[\phi_1\ \mathbf{U}\ \phi_2]]\!]_M(\pi) & = & \exists i.\ [\![\phi_2]\!]_M(\pi\downarrow i)\ \wedge\ \forall j.\ j<i\ \Rightarrow\ [\![\phi_1]\!]_M(\pi\downarrow j)
\end{array}
$$

- We look at each of these semantic equations in turn

$[\![p]\!]_M(\pi) = p(\pi\ 0)$

- Assume $M = (S, S_0, R, AP)$
- We have: $[\![p]\!]_M(\pi) = p(\pi\ 0)$
  - p is an atomic property, i.e. $p \in AP$
  - $\pi : \mathbb{N} \to S$ so $\pi\ 0 \in S$
  - $\pi\ 0$ is the first state in path π
  - $p(\pi\ 0)$ is *true* iff atomic property p holds of state $\pi\ 0$
- $[\![p]\!]_M(\pi)$ means p holds of the first state in path π
- Assumed $\mathsf{T}, \mathsf{F} \in AP$ with $\mathsf{T}(s) = true$ and $\mathsf{F}(s) = false$
  - $[\![\mathsf{T}]\!]_M(\pi)$ is always true
  - $[\![\mathsf{F}]\!]_M(\pi)$ is always false

$$[\![\neg\phi]\!]_M(\pi) = \neg([\![\phi]\!]_M(\pi))$$
$$[\![\phi_1 \vee \phi_2]\!]_M(\pi) = [\![\phi_1]\!]_M(\pi) \vee [\![\phi_2]\!]_M(\pi)$$

- $[\![\neg\phi]\!]_M(\pi) = \neg([\![\phi]\!]_M(\pi))$
  - $[\![\neg\phi]\!]_M(\pi)$ true iff $[\![\phi]\!]_M(\pi)$ is not true
- $[\![\phi_1 \vee \phi_2]\!]_M(\pi) = [\![\phi_1]\!]_M(\pi) \vee [\![\phi_2]\!]_M(\pi)$
  - $[\![\phi_1 \vee \phi_2]\!]_M(\pi)$ true iff $[\![\phi_1]\!]_M(\pi)$ is true or $[\![\phi_2]\!]_M(\pi)$ is true

$$[\![\mathbf{X}\phi]\!]_M(\pi) = [\![\phi]\!]_M(\pi{\downarrow}1)$$

- $[\![\mathbf{X}\phi]\!]_M(\pi) = [\![\phi]\!]_M(\pi{\downarrow}1)$
  - $\pi{\downarrow}1$ is π with the first state chopped off
    $\pi{\downarrow}1(0) = \pi(1+0) = \pi(1)$
    $\pi{\downarrow}1(1) = \pi(1+1) = \pi(2)$
    $\pi{\downarrow}1(2) = \pi(1+2) = \pi(3)$
    $\vdots$
- $[\![\mathbf{X}\phi]\!]_M(\pi)$ true iff $[\![\phi]\!]_M$ true *starting at the next state of* π

$$[\![\mathbf{F}\phi]\!]_M(\pi) = \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i)$$

- $[\![\mathbf{F}\phi]\!]_M(\pi) = \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i)$
  - $\pi{\downarrow}m$ is π with the first i states chopped off
    $\pi{\downarrow}i(0) = \pi(i+0) = \pi(i)$
    $\pi{\downarrow}i(1) = \pi(i+1)$
    $\pi{\downarrow}i(2) = \pi(i+2)$
    $\vdots$
  - $[\![\phi]\!]_M(\pi{\downarrow}i)$ true iff $[\![\phi]\!]_M$ true *starting i states along* π
- $[\![\mathbf{F}\phi]\!]_M(\pi)$ true iff $[\![\phi]\!]_M$ true *starting somewhere along* π
- "$\mathbf{F}\phi$" is read as "sometimes ϕ"

$$[\![\mathbf{G}\phi]\!]_M(\pi) = \forall i.\ [\![\phi]\!]_M(\pi{\downarrow}i)$$

- $[\![\mathbf{G}\phi]\!]_M(\pi) = \forall i.\ [\![\phi]\!]_M(\pi{\downarrow}i)$
  - $\pi{\downarrow}i$ is π with the first i states chopped off
  - $[\![\phi]\!]_M(\pi{\downarrow}i)$ true iff $[\![\phi]\!]_M$ true *starting i states along* π
- $[\![\mathbf{G}\phi]\!]_M(\pi)$ true iff $[\![\phi]\!]_M$ true *starting anywhere along* π
- "$\mathbf{G}\phi$" is read as "always ϕ" or "globally ϕ"
- $M \models \mathbf{GA}p$ defined earlier: $M \models \mathbf{GA}p \Leftrightarrow M \models \mathbf{G}(p)$
- $\mathbf{G}$ is definable in terms of $\mathbf{F}$ and $\neg$: $\mathbf{G}\phi = \neg(\mathbf{F}(\neg\phi))$
  $[\![\neg(\mathbf{F}(\neg\phi))]\!]_M(\pi) = \neg([\![\mathbf{F}(\neg\phi)]\!]_M(\pi))$
  $= \neg(\exists i.\ [\![\neg\phi]\!]_M(\pi{\downarrow}i))$
  $= \neg(\exists i.\ \neg([\![\phi]\!]_M(\pi{\downarrow}i)))$
  $= \forall i.\ [\![\phi]\!]_M(\pi{\downarrow}i)$
  $= [\![\mathbf{G}\phi]\!]_M(\pi)$

$$[\![[\phi_1\ \mathbf{U}\ \phi_2]]\!]_M(\pi) = \exists i.\ [\![\phi_2]\!]_M(\pi{\downarrow}i) \wedge \forall j.\ j{<}i \Rightarrow [\![\phi_1]\!]_M(\pi{\downarrow}j)$$

- $[\![[\phi_1\ \mathbf{U}\ \phi_2]]\!]_M(\pi) = \exists i.\ [\![\phi_2]\!]_M(\pi{\downarrow}i) \wedge \forall j.\ j{<}i \Rightarrow [\![\phi_1]\!]_M(\pi{\downarrow}j)$
  - $[\![\phi_2]\!]_M(\pi{\downarrow}i)$ true iff $[\![\phi_2]\!]_M$ true *starting i states along* π
  - $[\![\phi_1]\!]_M(\pi{\downarrow}j)$ true iff $[\![\phi_1]\!]_M$ true *starting j states along* π
- $[\![[\phi_1\ \mathbf{U}\ \phi_2]]\!]_M(\pi)$ is true iff
  $[\![\phi_2]\!]_M$ is true somewhere along π and up to then $[\![\phi_1]\!]_M$ is true
- "$[\phi_1\ \mathbf{U}\ \phi_2]\phi$" is read as "$\phi_1$ until ϕ_2"
- $\mathbf{F}$ is definable in terms of $[-\ \mathbf{U}\ -]$: $\mathbf{F}\phi = [\mathsf{T}\ \mathbf{U}\ \phi]$
  $[\![[\mathsf{T}\ \mathbf{U}\ \phi]]\!]_M(\pi)$
  $= \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i) \wedge \forall j.\ j{<}i \Rightarrow [\![\mathsf{T}]\!]_M(\pi{\downarrow}j)$
  $= \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i) \wedge \forall j.\ j{<}i \Rightarrow \mathit{true}$
  $= \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i) \wedge \mathit{true}$
  $= \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i)$
  $= [\![\mathbf{F}\phi]\!]_M(\pi)$

Computation Tree Logic (CTL)

- Syntax of CTL well-formed formulae:

ϕ	$::=$	p	(Atomic formula $p \in AP$)
	$\mid$	$\neg\phi$	(Negation)
	$\mid$	$\phi_1 \wedge \phi_2$	(Conjunction)
	$\mid$	$\phi_1 \vee \phi_2$	(Disjunction)
	$\mid$	$\phi_1 \Rightarrow \phi_2$	(Implication)
	$\mid$	$\mathbf{AX}\phi$	(All successors)
	$\mid$	$\mathbf{EX}\phi$	(Some successors)
	$\mid$	$\mathbf{A}[\phi_1\ \mathbf{U}\ \phi_2]$	(Until – along all paths)
	$\mid$	$\mathbf{E}[\phi_1\ \mathbf{U}\ \phi_2]$	(Until – along some path)

- Sometimes just write "p" rather than "p"
- LTL formulae ϕ are evaluated on paths – path formulae
- CTL formulae ϕ are evaluated on states – state formulae

Semantics of CTL

- Assume $M = (S, S_0, R, AP)$ and then define:

$$
\begin{array}{lcl}
[\![p]\!]_M(s) & = & p(s) \\
[\![\neg\phi]\!]_M(s) & = & \neg([\![\phi]\!]_M(s)) \\
[\![\phi_1 \wedge \phi_2]\!]_M(s) & = & [\![\phi_1]\!]_M(s) \ \wedge \ [\![\phi_2]\!]_M(s) \\
[\![\phi_1 \vee \phi_2]\!]_M(s) & = & [\![\phi_1]\!]_M(s) \ \vee \ [\![\phi_2]\!]_M(s) \\
[\![\phi_1 \Rightarrow \phi_2]\!]_M(s) & = & [\![\phi_1]\!]_M(s) \ \Rightarrow \ [\![\phi_2]\!]_M(s) \\
[\![\mathbf{AX}\phi]\!]_M(s) & = & \forall s'.\ R\ s\ s' \ \Rightarrow \ [\![\phi]\!]_M(s') \\
[\![\mathbf{EX}\phi]\!]_M(s) & = & \exists s'.\ R\ s\ s' \ \wedge \ [\![\phi]\!]_M(s') \\
[\![\mathbf{A}[\phi_1\ \mathbf{U}\ \phi_2]]\!]_M(s) & = & \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \\
 & & \quad \Rightarrow \exists i.\ [\![\phi_2]\!]_M(\pi(i)) \\
 & & \qquad \wedge \\
 & & \quad \forall j.\ j{<}i \ \Rightarrow \ [\![\phi_1]\!]_M(\pi(j)) \\
[\![\mathbf{E}[\phi_1\ \mathbf{U}\ \phi_2]]\!]_M(s) & = & \exists \pi.\ \mathsf{Path}\ R\ s\ \pi \\
 & & \quad \wedge \ \exists i.\ [\![\phi_2]\!]_M(\pi(i)) \\
 & & \qquad \wedge \\
 & & \quad \forall j.\ j{<}i \ \Rightarrow \ [\![\phi_1]\!]_M(\pi(j))
\end{array}
$$

The defined operator AF

- Define $\mathbf{AF}\phi = \mathbf{A}[\mathsf{T}\ \mathbf{U}\ \phi]$
- $\mathbf{AF}\phi$ true at s iff ϕ true somewhere on every R-path from s

$$
\begin{array}{lcl}
[\![\mathbf{AF}\phi]\!]_M(s) & = & [\![\mathbf{A}[\mathsf{T}\ \mathbf{U}\ \phi]]\!]_M(s) \\
 & = & \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \\
 & & \quad \Rightarrow \\
 & & \quad \exists i.\ [\![\phi]\!]_M(\pi(i)) \ \wedge \ \forall j.\ j < i \ \Rightarrow \ [\![\mathsf{T}]\!]_M(\pi(j)) \\
 & = & \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \\
 & & \quad \Rightarrow \\
 & & \quad \exists i.\ [\![\phi]\!]_M(\pi(i)) \ \wedge \ \forall j.\ j < i \ \Rightarrow \ \mathit{true} \\
 & = & \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \ \Rightarrow \ \exists i.\ [\![\phi]\!]_M(\pi(i))
\end{array}
$$

The defined operator EF

- Define $\mathbf{EF}\phi = \mathbf{E}[\mathsf{T}\ \mathbf{U}\ \phi]$
- $\mathbf{EF}\phi$ true at s iff ϕ true somewhere on some R-path from s

$$
\begin{array}{lcl}
[\![\mathbf{EF}\phi]\!]_M(s) & = & [\![\mathbf{E}[\mathsf{T}\ \mathbf{U}\ \phi]]\!]_M(s) \\
 & = & \exists \pi.\ \mathsf{Path}\ R\ s\ \pi \\
 & & \quad \wedge \\
 & & \quad \exists i.\ [\![\phi]\!]_M(\pi(i)) \ \wedge \ \forall j.\ j < i \ \Rightarrow \ [\![\mathsf{T}]\!]_M(\pi(j)) \\
 & = & \exists \pi.\ \mathsf{Path}\ R\ s\ \pi \\
 & & \quad \wedge \\
 & & \quad \exists i.\ [\![\phi]\!]_M(\pi(i)) \ \wedge \ \forall j.\ j < i \ \Rightarrow \ \mathit{true} \\
 & = & \exists \pi.\ \mathsf{Path}\ R\ s\ \pi \ \wedge \ \exists i.\ [\![\phi]\!]_M(\pi(i))
\end{array}
$$

The defined operator AG

- Define $\mathbf{AG}\phi = \neg\mathbf{EF}(\neg\phi)$
- $\mathbf{AG}\phi$ true at s iff ϕ true everywhere on every R-path from s

$$
\begin{array}{lcl}
[\![\mathbf{AG}\phi]\!]_M(s) & = & [\![\neg\mathbf{EF}(\neg\phi)]\!]_M(s) \\
 & = & \neg([\![\mathbf{EF}(\neg\phi)]\!]_M(s)) \\
 & = & \neg(\exists \pi.\ \mathsf{Path}\ R\ s\ \pi \wedge \exists i.\ [\![\neg\phi]\!]_M(\pi(i))) \\
 & = & \neg(\exists \pi.\ \mathsf{Path}\ R\ s\ \pi \wedge \exists i.\ \neg[\![\phi]\!]_M(\pi(i))) \\
 & = & \forall \pi.\ \neg(\mathsf{Path}\ R\ s\ \pi \wedge \exists i.\ \neg[\![\phi]\!]_M(\pi(i))) \\
 & = & \forall \pi.\ \neg\mathsf{Path}\ R\ s\ \pi \vee \neg(\exists i.\ \neg[\![\phi]\!]_M(\pi(i))) \\
 & = & \forall \pi.\ \neg\mathsf{Path}\ R\ s\ \pi \vee \forall i.\ \neg\neg[\![\phi]\!]_M(\pi(i)) \\
 & = & \forall \pi.\ \neg\mathsf{Path}\ R\ s\ \pi \vee \forall i.\ [\![\phi]\!]_M(\pi(i)) \\
 & = & \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \Rightarrow \forall i.\ [\![\phi]\!]_M(\pi(i))
\end{array}
$$

- $\mathbf{AG}\phi$ means ϕ true at all reachable states
- $[\![\mathbf{AG}(p)]\!]_M(s) \ \equiv \ \forall s'.\ R^*\ s\ s' \ \Rightarrow \ p(s')$

The defined operator EG

- Define $\mathbf{EG}\phi = \neg\mathbf{AF}(\neg\phi)$
- $\mathbf{EG}\phi$ true at s iff ϕ true everywhere on some R-path from s

$$
\begin{array}{lcl}
[\![\mathbf{EG}\phi]\!]_M(s) & = & [\![\neg\mathbf{AF}(\neg\phi)]\!]_M(s) \\
 & = & \neg([\![\mathbf{AF}(\neg\phi)]\!]_M(s)) \\
 & = & \neg(\forall \pi.\ \mathsf{Path}\ R\ s\ \pi \Rightarrow \exists i.\ [\![\neg\phi]\!]_M(\pi(i))) \\
 & = & \neg(\forall \pi.\ \mathsf{Path}\ R\ s\ \pi \Rightarrow \exists i.\ \neg[\![\phi]\!]_M(\pi(i))) \\
 & = & \exists \pi.\ \neg(\mathsf{Path}\ R\ s\ \pi \Rightarrow \exists i.\ \neg[\![\phi]\!]_M(\pi(i))) \\
 & = & \exists \pi.\ \mathsf{Path}\ R\ s\ \pi \wedge \neg(\exists i.\ \neg[\![\phi]\!]_M(\pi(i))) \\
 & = & \exists \pi.\ \mathsf{Path}\ R\ s\ \pi \wedge \forall i.\ \neg\neg[\![\phi]\!]_M(\pi(i)) \\
 & = & \exists \pi.\ \mathsf{Path}\ R\ s\ \pi \wedge \forall i.\ [\![\phi]\!]_M(\pi(i))
\end{array}
$$

The defined operator $\mathbf{A}[\phi_1\mathbf{W}\phi_2]$

- $\mathbf{A}[\phi_1\mathbf{W}\phi_2]$ is a 'partial correctness' version of $\mathbf{A}[\phi_1\mathbf{U}\phi_2]$
- It is true at s if along all R-paths from s:
  - ϕ_1 always holds on the path
  - ϕ_2 holds sometime on the path, and until it does ϕ_1 holds
- Define

$$
\begin{array}{l}
[\![\mathbf{A}[\phi_1\mathbf{W}\phi_2]]\!]_M(s) \\
\quad = [\![\neg\mathbf{E}[(\phi_1{\wedge}\neg\phi_2)\mathbf{U}(\neg\phi_1{\wedge}\neg\phi_2)]]\!]_M(s) \\
\quad = \neg[\![\mathbf{E}[(\phi_1{\wedge}\neg\phi_2)\mathbf{U}(\neg\phi_1{\wedge}\neg\phi_2)]]\!]_M(s) \\
\quad = \neg(\exists \pi.\ \mathsf{Path}\ R\ s\ \pi \\
\qquad\quad \wedge \\
\qquad\quad \exists i.\ [\![\neg\phi_1{\wedge}\neg\phi_2]\!]_M(\pi(i)) \\
\qquad\qquad \wedge \\
\qquad\quad \forall j.\ j{<}i \ \Rightarrow \ [\![\phi_1{\wedge}\neg\phi_2]\!]_M(\pi(j)))
\end{array}
$$

- Exercise: understand the next three slides!

$\mathbf{A}[\phi_1 \mathbf{W} \phi_2]$ continued (1)

- Continuing:

$$\neg(\exists \pi.\ \mathsf{Path}\ R\ s\ \pi \\ \wedge \\ \exists i.\ [\![\neg\phi_1 \wedge \neg\phi_2]\!]_M(\pi(i)) \wedge \forall j.\ j<i \Rightarrow [\![\phi_1 \wedge \neg\phi_2]\!]_M(\pi(j)))$$

$$= \forall \pi.\ \neg(\mathsf{Path}\ R\ s\ \pi \\ \wedge \\ \exists i.\ [\![\neg\phi_1 \wedge \neg\phi_2]\!]_M(\pi(i)) \wedge \forall j.\ j<i \Rightarrow [\![\phi_1 \wedge \neg\phi_2]\!]_M(\pi(j)))$$

$$= \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \\ \Rightarrow \\ \neg(\exists i.\ [\![\neg\phi_1 \wedge \neg\phi_2]\!]_M(\pi(i)) \wedge \forall j.\ j<i \Rightarrow [\![\phi_1 \wedge \neg\phi_2]\!]_M(\pi(j)))$$

$$= \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \\ \Rightarrow \\ \forall i.\ \neg[\![\neg\phi_1 \wedge \neg\phi_2]\!]_M(\pi(i)) \vee \neg(\forall j.\ j<i \Rightarrow [\![\phi_1 \wedge \neg\phi_2]\!]_M(\pi(j)))$$

$\mathbf{A}[\phi_1 \mathbf{W} \phi_2]$ continued (2)

- Continuing:

$$= \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \\ \Rightarrow \\ \forall i.\ \neg[\![\neg\phi_1 \wedge \neg\phi_2]\!]_M(\pi(i)) \vee \neg(\forall j.\ j<i \Rightarrow [\![\phi_1 \wedge \neg\phi_2]\!]_M(\pi(j)))$$

$$= \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \\ \Rightarrow \\ \forall i.\ \neg(\forall j.\ j<i \Rightarrow [\![\phi_1 \wedge \neg\phi_2]\!]_M(\pi(j))) \vee \neg[\![\neg\phi_1 \wedge \neg\phi_2]\!]_M(\pi(i))$$

$$= \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \\ \Rightarrow \\ \forall i.\ (\forall j.\ j<i \Rightarrow [\![\phi_1 \wedge \neg\phi_2]\!]_M(\pi(j))) \Rightarrow [\![\phi_1 \vee \phi_2]\!]_M(\pi(i))$$

- Exercise: explain why this is $[\![\mathbf{A}[\phi_1 \mathbf{W} \phi_2]]\!]_M(s)$?
  - this exercise illustrates the subtlety of writing CTL!

$\mathbf{A}[\phi \mathbf{W} \mathsf{F}] = \mathbf{AG}\ \phi$

- From last slide:

$$[\![\mathbf{A}[\phi_1 \mathbf{W} \phi_2]]\!]_M(s) \\ = \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \\ \Rightarrow \\ \forall i.\ (\forall j.\ j<i \Rightarrow [\![\phi_1 \wedge \neg\phi_2]\!]_M(\pi(j))) \Rightarrow [\![\phi_1 \vee \phi_2]\!]_M(\pi(i))$$

- Set ϕ_1 to ϕ and ϕ_2 to F:

$$[\![\mathbf{A}[\phi \mathbf{W} \mathsf{F}]]\!]_M(s) \\ = \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \\ \Rightarrow \\ \forall i.\ (\forall j.\ j<i \Rightarrow [\![\phi \wedge \neg\mathsf{F}]\!]_M(\pi(j))) \Rightarrow [\![\phi \vee \mathsf{F}]\!]_M(\pi(i))$$

- Simplify:

$$[\![\mathbf{A}[\phi \mathbf{W} \mathsf{F}]]\!]_M(s) \\ = \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \Rightarrow \forall i.\ (\forall j.\ j<i \Rightarrow [\![\phi]\!]_M(\pi(j))) \Rightarrow [\![\phi]\!]_M(\pi(i))$$

- By induction on i:

$$[\![\mathbf{A}[\phi \mathbf{W} \mathsf{F}]]\!]_M(s) = \forall \pi.\ \mathsf{Path}\ R\ s\ \pi \Rightarrow \forall i.\ [\![\phi]\!]_M(\pi(i))$$

- **Exercise:** describe the property specified by $\mathbf{A}[\mathsf{T}\ \mathbf{W} \phi]$

Summary of CTL operators (primitive + defined)

- CTL formulas:

p	(Atomic formula - p : *states*→*bool*)
$\neg\phi$	(Negation)
$\phi_1 \wedge \phi_2$	(Conjunction)
$\phi_1 \vee \phi_2$	(Disjunction)
$\phi_1 \Rightarrow \phi_2$	(Implication)
$\mathbf{AX}\phi$	(All successors)
$\mathbf{EX}\phi$	(Some successors)
$\mathbf{AF}\phi$	(Somewhere – along all paths)
$\mathbf{EF}\phi$	(Somewhere – along some path)
$\mathbf{AG}\phi$	(Everywhere – along all paths)
$\mathbf{EG}\phi$	(Everywhere – along some path)
$\mathbf{A}[\phi_1\ \mathbf{U}\ \phi_2]$	(Until – along all paths)
$\mathbf{E}[\phi_1\ \mathbf{U}\ \phi_2]$	(Until – along some path)
$\mathbf{A}[\phi_1\ \mathbf{W}\ \phi_2]$	(Unless – along all paths)
$\mathbf{E}[\phi_1\ \mathbf{W}\ \phi_2]$	(Unless – along some path)

Example CTL formulas

- $\mathbf{EF}(\mathit{Started} \wedge \neg \mathit{Ready})$
  It is possible to get to a state where Started holds but Ready does not hold
- $\mathbf{AG}(\mathit{Req} \Rightarrow \mathbf{AF}\mathit{Ack})$
  If a request Req occurs, then it will eventually be acknowledged by Ack
- $\mathbf{AG}(\mathbf{AF}\mathit{DeviceEnabled})$
  DeviceEnabled is always true somewhere along every path starting anywhere: i.e. DeviceEnabled holds infinitely often along every path
- $\mathbf{AG}(\mathbf{EF}\mathit{Restart})$
  From any state it is possible to get to a state for which Restart holds

More CTL examples (1)

- $\mathbf{AG}(\mathit{Req} \Rightarrow \mathbf{A}[\mathit{Req}\ \mathbf{U}\ \mathit{Ack}])$
  If a request Req occurs, then it continues to hold, until it is eventually acknowledged
- $\mathbf{AG}(\mathit{Req} \Rightarrow \mathbf{AX}(\mathbf{A}[\neg \mathit{Req}\ \mathbf{U}\ \mathit{Ack}]))$
  Whenever Req is true either it must become false on the next cycle and remains false until Ack, or Ack must become true on the next cycle
  Exercise: is the **AX** necessary?
- $\mathbf{AG}(\mathit{Req} \Rightarrow (\neg \mathit{Ack} \Rightarrow \mathbf{AX}(\mathbf{A}[\mathit{Req}\ \mathbf{U}\ \mathit{Ack}])))$
  Whenever Req is true and Ack is false then Ack will eventually become true and until it does Req will remain true
  Exercise: is the **AX** necessary?

More CTL examples (2)

- **AG**[*Enabled* ⇒ **AG**[*Start* ⇒ **A**[¬*Waiting* **U** *Ack*]]]
  If Enabled is ever true then if Start is true in any subsequent state then Ack will eventually become true, and until it does Waiting will be false
- **AG**[¬Req_1∧¬Req_2⇒**A**[¬Req_1∧¬Req_2**U**(*Start*∧¬Req_2)]]
  Whenever Req_1 and Req_2 are false, they remain false until Start becomes true with Req_2 still false
- **AG**[*Req* ⇒ **AX**(*Ack* ⇒ **AF** ¬*Req*)]
  If Req is true and Ack becomes true one cycle later, then eventually Req will become false

Some abbreviations

- $\mathbf{AX}_i\ \phi \equiv \underbrace{\mathbf{AX}(\mathbf{AX}(\cdots(\mathbf{AX}\ \phi)\cdots))}_{i \text{ instances of } \mathbf{AX}}$
  ϕ is true on all paths i units of time later
- $\mathbf{ABF}_{i..j}\ \phi \equiv \mathbf{AX}_i(\underbrace{\phi \vee \mathbf{AX}(\phi \vee \cdots \mathbf{AX}(\phi \vee \mathbf{AX}\ \phi)\cdots)}_{j-i \text{ instances of } \mathbf{AX}})$
  ϕ is true on all paths sometime between i units of time later and j units of time later
- **AG**[*Req* ⇒ **AX**[Ack_1 ∧ $\mathbf{ABF}_{1..6}$(Ack_2 ∧ **A**[*Wait* **U** *Reply*])]]
  One cycle after Req, Ack_1 should become true, and then Ack_2 becomes true 1 to 6 cycles later and then eventually Reply becomes true, but until it does Wait holds from the time of Ack_2
- More abbreviations in 'Industry Standard' language PSL

CTL model checking

- For LTL path formulae ϕ recall that $M \models \phi$ is defined by:
  $$M \models \phi \Leftrightarrow \forall \pi\ s.\ s \in S_0 \wedge \text{Path}\ R\ s\ \pi \Rightarrow [\![\phi]\!]_M(\pi)$$
- For CTL state formulae ϕ the definition of $M \models \phi$ is:
  $$M \models \phi \Leftrightarrow \forall s.\ s \in S_0 \Rightarrow [\![\phi]\!]_M(s)$$
- M common; LTL, CTL formulae ϕ and semantics $[\![\ ]\!]_M$ differ
- CTL model checking algorithm:
  - compute $\{s \mid [\![\phi]\!]_M(s) = \mathit{true}\}$ bottom up
  - check $S_0 \subseteq \{s \mid [\![\phi]\!]_M(s) = \mathit{true}\}$
  - symbolic model checking represents these sets as BDDs

CTL model checking: p, **AX**ϕ, **EX**ϕ

- For CTL formula ϕ let $\{\phi\} = \{s \mid [\![\phi]\!]_M(s) = \mathit{true}\}$
- $\{p\} = \{s \mid p(s) = \mathit{true}\}$
  - scan through set of states S marking states that satisfy p
  - $\{p\}$ is set of marked states
- To compute $\{\mathbf{AX}\phi\}$
  - recursively compute $\{\phi\}$
  - marks those states all of whose successors are in $\{\phi\}$
  - $\{\mathbf{AX}\phi\}$ is the set of marked states
- To compute $\{\mathbf{EX}\phi\}$
  - recursively compute $\{\phi\}$
  - marks those states with at least one successor in $\{\phi\}$
  - $\{\mathbf{AX}\phi\}$ is the set of marked states

CTL model checking: $\{[\![\mathbf{E}[\phi_1\ \mathbf{U}\ \phi_2]]\!]\}$, $\{[\![\mathbf{A}[\phi_1\ \mathbf{U}\ \phi_2]]\!]\}$

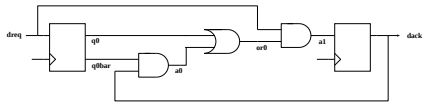
- To compute $\{\mathbf{E}[\phi_1\ \mathbf{U}\ \phi_2]\}$
  - recursively compute $\{\phi_1\}$ and $\{\phi_2\}$
  - mark all states in $\{\phi_2\}$
  - mark all states in $\{\phi_1\}$ with a successor state that is marked
  - repeat previous line until no change
  - $\{\mathbf{E}[\phi_1\ \mathbf{U}\ \phi_2]\}$ is set of marked states
- More formally: $\{\mathbf{E}[\phi_1\ \mathbf{U}\ \phi_2]\} = \bigcup_{n=0}^{\infty}\{\mathbf{E}[\phi_1\ \mathbf{U}\ \phi_2]\}_n$ where:
  $$\begin{array}{lcl}\{\mathbf{E}[\phi_1\ \mathbf{U}\ \phi_2]\}_0 & = & \{\phi_2\}\\ \{\mathbf{E}[\phi_1\ \mathbf{U}\ \phi_2]\}_{n+1} & = & \{\mathbf{E}[\phi_1\ \mathbf{U}\ \phi_2]\}_n\\ & & \cup\\ & & \{s \in \mathcal{S}\phi_1 \mid \exists s' \in \{\mathbf{E}[\phi_1\ \mathbf{U}\ \phi_2]\}_n.\ R\ s\ s'\}\end{array}$$
- $\{\mathbf{A}[\phi_1\ \mathbf{U}\ \phi_2]\}$ similar, but with a more complicated iteration
  - details omitted

Example: checking **EF** p

- $\mathbf{EF}\phi = \mathbf{E}[\mathsf{T}\ \mathbf{U}\ \phi]$
  - holds if ϕ holds along some path
- Let $\mathcal{S}_n = \{\mathbf{E}[\mathsf{T}\ \mathbf{U}\ p]\}_n$:
  $$\begin{array}{lcl}\mathcal{S}_0 & = & \{p\}\\ & = & \{s \mid p(s)\}\\ \mathcal{S}_{n+1} & = & \mathcal{S}_n \cup \{s \mid \exists s'.\ R\ s\ s' \wedge s' \in \mathcal{S}_n\}\end{array}$$
  - mark all the states satisfying p
  - mark all with at least one marked successor
  - repeat until no change
  - $\{\mathbf{EF}\ p\}$ is set of marked states

Example: RCV

- Recall the handshake circuit:



- State represented by a triple of Booleans $(dreq, q0, dack)$
- A model of RCV is M_{RCV} where:

  $M = (S_{\mathrm{RCV}}, \{(1,1,1)\}, R_{\mathrm{RCV}}, AP)$

  and

  $R_{\mathrm{RCV}}\ (dreq, q0, dack)\ (dreq', q0', dack') =$
  $(q0' = dreq) \wedge (dack' = (dreq \wedge (q0 \vee dack)))$

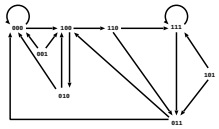
RCV state transition diagram

- Possible states for RCV:

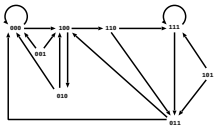
  $\{000, 001, 010, 011, 100, 101, 110, 111\}$

  where $b_2b_1b_0$ denotes state

  $\texttt{dreq} = b_2 \wedge \texttt{q0} = b_1 \wedge \texttt{dack} = b_0$
- Graph of the transition relation:



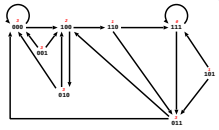
Model checking $M_{\mathrm{RCV}} \models (\lambda b_2b_1b_0.\ b_2 \wedge b_1 \wedge b_0)$



- Define:

$$\begin{array}{lll} \mathcal{S}_0 & = \{b_2b_1b_0 \mid (\lambda b_2b_1b_0.\ b_2 \wedge b_1 \wedge b_0)b_2b_1b_0)\} \\ & = \{b_2b_1b_0 \mid b_2 \wedge b_1 \wedge b_0\} \\ \\ \mathcal{S}_{i+1} & = \mathcal{S}_i \cup \{s \mid \exists s' \in \mathcal{S}_i.\ \mathcal{R}(s,s')\} \\ & = \mathcal{S}_i \cup \{b_2b_1b_0 \mid \\ & \qquad \exists b_2'b_1'b_0' \in \mathcal{S}_i.\ (b_1' = b_2) \wedge (b_0' = b_2 \wedge (b_1 \vee b_0))\} \end{array}$$

Model checking $M_{\mathrm{RCV}} \models (\lambda b_2b_1b_0.\ b_2 \wedge b_1 \wedge b_0)$ (continued)



- Compute:

$$\begin{array}{ll} \mathcal{S}_0 & = \{111\} \\ \mathcal{S}_1 & = \{111\} \cup \{101, 110\} \\ & = \{111, 101, 110\} \\ \mathcal{S}_2 & = \{111, 101, 110\} \cup \{100\} \\ & = \{111, 101, 110, 100\} \\ \mathcal{S}_3 & = \{111, 101, 110, 100\} \cup \{000, 001, 010, 011\} \\ & = \{111, 101, 110, 100, 000, 001, 010, 011\} \\ \mathcal{S}_i & = \mathcal{S}_3 \quad (i > 3) \end{array}$$

- $\forall s.\ [\![\mathbf{EF}\ (\lambda(\texttt{dreq}, \texttt{q0}, \texttt{dack}).\ \texttt{dreq} \wedge \texttt{q0} \wedge \texttt{dack})]\!]_M(s)$
- $M_{\mathrm{RCV}} \models \mathbf{EF}\ (\lambda(\texttt{dreq}, \texttt{q0}, \texttt{dack}).\ \texttt{dreq} \wedge \texttt{q0} \wedge \texttt{dack})$

Symbolic model checking

- Represent sets of states with BDDs
- Represent Transition relation with a BDD
- If BDDs of $\{\phi\}$, $\{\phi_1\}$, $\{\phi_2\}$ are known, then:
  - BDDs of $\{\neg\phi\}$, $\{\phi_1 \wedge \phi_2\}$, $\{\phi_1 \vee \phi_2\}$, $\{\phi_1 \Rightarrow \phi_2\}$ computed using standard BDD algorithms
  - BDDs of $\{\mathbf{AX}\phi\}$, $\{\mathbf{EX}\phi\}$, $\{\mathbf{A}[\phi_1\ \mathbf{U}\ \phi_2]\}$, $\{\mathbf{E}[P\ \mathbf{U}\ Q]\}$ computed using straightforward algorithms (see textbooks)
- Model checking CTL generalises reachable states Iteration

History of Model checking

- CTL model checking due to Emerson, Clarke & Sifakis
- Symbolic model checking due to several people:
  - Clarke & McMillan (idea usually credited to McMillan's PhD)
  - Coudert, Berthet & Madre
  - Pixley
- SMV (McMillan) is a popular symbolic model checker:

`http://www.cs.cmu.edu/~modelcheck/smv.html`	(original)
`http://www.kenmcmil.com/smv.html`	(Cadence extension by McMillan)
`http://nusmv.irst.itc.it/`	(new implementation)

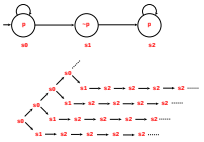
- Other temporal logics
  - CTL*: combines CTL and LTL
  - Engineer friendly industrial languages: PSL, SVA

Expressibility of CTL

- Consider the property

  "on every path there is a point after which p is always true on that path"

- Consider



- Property true, but cannot be expressed in CTL
  - would need something like **AF**ϕ
  - where ϕ is something like "property p *true from now on*"
  - but in CTL ϕ must start with a path quantifier **A** or **E**
  - cannot talk about current path, only about all or some paths
  - **AF**(**AG** p) is false (consider path s0 s0 s0 $\cdots$)

LTL can express things CTL can't

- Recall:

$$\begin{array}{lcl}[\![\mathbf{F}\phi]\!]_M(\pi) & = & \exists i.\ [\![\phi]\!]_M(\pi\downarrow i)\\ [\![\mathbf{G}\phi]\!]_M(\pi) & = & \forall i.\ [\![\phi]\!]_M(\pi\downarrow i)\end{array}$$

- **FG**ϕ is true if there is a point after which ϕ is always true

$$\begin{array}{lcl}[\![\mathbf{FG}\phi]\!]_M(\pi) & = & [\![\mathbf{F}(\mathbf{G}(\phi))]\!]_M(\pi)\\ & = & \exists m_1.\ [\![\mathbf{G}(\phi)]\!]_M(\pi\downarrow m_1)\\ & = & \exists m_1.\ \forall m_2.\ [\![\phi]\!]_M((\pi\downarrow m_1)\downarrow m_2)\\ & = & \exists m_1.\ \forall m_2.\ [\![\phi]\!]_M(\pi\downarrow(m_1{+}m_2))\end{array}$$

- LTL can express things that CTL can't express

CTL can express things that LTL can't express

- **AG**(**EF** ϕ) says:

  "from every state it is possible to get to a state for which ϕ holds"

- Can't say this in LTL (proof omitted)
- Consider disjunction:

  "along every path there is a state from which ϕ will hold forever
  or
  from every state it is possible to get to a state for which ϕ holds"

- Can't say this in either CTL or LTL! (proof omitted)
- CTL* combines CTL and LTL and can express this property

CTL*

- Both state formulas (ψ) and path formulas (ϕ)
  - state formulas are true of a state s like CTL
  - path formulas are true of a path π like LTL
- Defined mutually recursively

$$\begin{array}{lcll}\psi & ::= & p & \text{(Atomic formula)}\\ & | & \neg\psi & \text{(Negation)}\\ & | & \psi_1 \vee \psi_2 & \text{(Disjunction)}\\ & | & \mathbf{A}\phi & \text{(All paths)}\\ & | & \mathbf{E}\phi & \text{(Some paths)}\\ \phi & ::= & \psi & \text{(Every state formula is a path formula)}\\ & | & \neg\phi & \text{(Negation)}\\ & | & \phi_1 \vee \phi_2 & \text{(Disjunction)}\\ & | & \mathbf{X}\phi & \text{(Successor)}\\ & | & \mathbf{F}\phi & \text{(Sometimes)}\\ & | & \mathbf{G}\phi & \text{(Always)}\\ & | & [\phi_1\ \mathbf{U}\ \phi_2] & \text{(Until)}\end{array}$$

- CTL is CTL* with **X**, **F**, **G**, [−**U**−] preceded by **A** or **E**
- LTL consists of CTL* formulas of form **A**ϕ, where the only state formulas in ϕ are atomic

CTL* semantics

- Combines CTL state semantics with LTL path semantics:

$$\begin{array}{lcl}[\![p]\!]_M(s) & = & p(s)\\ [\![\neg\psi]\!]_M(s) & = & \neg([\![\psi]\!]_M(s))\\ [\![\psi_1 \vee \psi_2]\!]_M(s) & = & [\![\psi_1]\!]_M(s) \vee [\![\psi_2]\!]_M(s)\\ [\![\mathbf{A}\phi]\!]_M(s) & = & \forall\pi.\ \text{Path}\ R\ s\ \pi \Rightarrow \phi(\pi)\\ [\![\mathbf{E}\phi]\!]_M(s) & = & \exists\pi.\ \text{Path}\ R\ s\ \pi \wedge [\![\phi]\!]_M(\pi)\end{array}$$

$$\begin{array}{lcl}[\![\psi]\!]_M(\pi) & = & [\![\psi]\!]_M(\pi(0))\\ [\![\neg\phi]\!]_M(\pi) & = & \neg([\![\phi]\!]_M(\pi))\\ [\![\phi_1 \vee \phi_2]\!]_M(\pi) & = & [\![\phi_1]\!]_M(\pi) \vee [\![\phi_2]\!]_M(\pi)\\ [\![\mathbf{X}\phi]\!]_M(\pi) & = & [\![\phi]\!]_M(\pi\downarrow 1)\\ [\![\mathbf{F}\phi]\!]_M(\pi) & = & \exists m.\ [\![\phi]\!]_M(\pi\downarrow m)\\ [\![\mathbf{G}\phi]\!]_M(\pi) & = & \forall m.\ [\![\phi]\!]_M(\pi\downarrow m)\\ [\![[\phi_1\ \mathbf{U}\ \phi_2]]\!]_M(\pi) & = & \exists i.\ [\![\phi_2]\!]_M(\pi\downarrow i) \wedge \forall j.\ j{<}i \Rightarrow [\![\phi_1]\!]_M(\pi\downarrow j)\end{array}$$

- Note $[\![\psi]\!]_M : S{\to}\mathbb{B}$ and $[\![\phi]\!]_M : (\mathbb{N}{\to}S){\to}\mathbb{B}$

LTL and CTL as CTL*

- As usual: $M = (S, S_0, R, AP)$
- If ψ is a CTL* state formula: $M \models \psi \Leftrightarrow \forall s \in S_0.\ [\![\psi]\!]_M(s)$
- If ϕ is an LTL path formula then: $M \models_{\text{LTL}} \phi \Leftrightarrow M \models_{\text{CTL*}} \mathbf{A}\phi$
- If R is total ($\forall s.\ \exists s'.\ R\ s\ s'$) then (exercise):
  $\forall s\ s'.\ R\ s\ s' \Leftrightarrow \exists\pi.\ \text{Path}\ R\ s\ \pi \wedge (\pi(1) = s')$
- The meanings of CTL formulae are the same in CTL*

$$\begin{array}{lcll}[\![\mathbf{A}(\mathbf{X}\psi)]\!]_M(s) & & & \\ = \forall\pi.\ \text{Path}\ R\ s\ \pi \Rightarrow [\![\mathbf{X}\psi]\!]_M(\pi) & & & \\ = \forall\pi.\ \text{Path}\ R\ s\ \pi \Rightarrow [\![\psi]\!]_M(\pi\downarrow 1) & & & (\psi \text{ as path formula})\\ = \forall\pi.\ \text{Path}\ R\ s\ \pi \Rightarrow [\![\psi]\!]_M((\pi\downarrow 1)(0)) & & & (\psi \text{ as state formula})\\ = \forall\pi.\ \text{Path}\ R\ s\ \pi \Rightarrow [\![\psi]\!]_M(\pi(1)) & & & \end{array}$$

$$\begin{array}{l}[\![\mathbf{AX}\psi]\!]_M(s)\\ = \forall s'.\ R\ s\ s' \Rightarrow [\![\psi]\!]_M(s')\\ = \forall s'.\ (\exists\pi.\ \text{Path}\ R\ s\ \pi \wedge (\pi(1) = s')) \Rightarrow [\![\psi]\!]_M(s')\\ = \forall s'.\ \forall\pi.\ \text{Path}\ R\ s\ \pi \wedge (\pi(1) = s') \Rightarrow [\![\psi]\!]_M(s')\\ = \forall\pi.\ \text{Path}\ R\ s\ \pi \Rightarrow [\![\psi]\!]_M(\pi(1))\end{array}$$

Exercise: do similar proofs for other CTL formulae

Fairness

- May want to assume system or environment is 'fair'
- Example 1: fair arbiter
  the arbiter doesn't ignore one of its requests forever
  - not every request need be granted
  - want to exclude infinite number of requests and no grant
- Example 2: reliable channel
  no message continuously transmitted but never received
  - not every message need be received
  - want to exclude an infinite number of sends and no receive

Handling fairness in CTL and LTL

- Consider:
  P holds infinitely often along a path then so does Q
- In LTL is expressible as $\mathbf{G}(\mathbf{F}\,P) \Rightarrow \mathbf{G}(\mathbf{F}\,Q)$
- Can't say this in CTL
  - why not – what's wrong with $\mathbf{AG}(\mathbf{AF}\,P) \Rightarrow \mathbf{AG}(\mathbf{AF}\,Q)$?
  - in CTL* expressible as $\mathbf{A}(\mathbf{G}(\mathbf{F}\,P) \Rightarrow \mathbf{G}(\mathbf{F}\,Q))$
  - fair CTL model checking implemented in checking algorithm
  - fair LTL just a fairness assumption like $\mathbf{G}(\mathbf{F}\,P) \Rightarrow \cdots$
- Fairness is a tricky and subtle subjec
  - many kinds of fairness:
    'weak fairness', 'strong fairness' etc
  - exist whole books on fairness

Propositional modal μ-calculus

- You may learn this in *Topics in Concurrency*
- μ-calculus is an even more powerful property language
  - has fixed-point operators
  - both maximal and minimal fixed points
  - model checking consists of calculating fixed points
  - many logics (e.g. CTL*) can be translated into μ-calculus
- Strictly stronger than CTL*
  - expressibility strictly increases as allowed nesting increases
  - need fixed point operators nested 2 deep for CTL*
- The μ-calculus is very non-intuitive to use!
  - intermediate code rather than a practical property language
  - nice meta-theory and algorithms, but terrible usability!

SEREs: Sequential Extended Regular Expressions

- SEREs are from the industrial PSL (more on PSL later)
- Syntax :

r ::=	p	(Atomic formula $p \in AP$)
	$!p$	(Negated atomic formula $p \in AP$)
	$r_1 \mid r_2$	(Disjunction)
	$r_1 ; r_2$	(Concatenation)
	$r_1 : r_2$	(Fusion)
	$r_1 \,\&\&\, r_2$	(Length matching conjunction)
	$r[*]$	(Repeat)

- Semantics:
  (w ranges over finite lists of states s; $|w|$ is length of w;
  $w_1.w_2$ is concatenation of w_1 and w_2; $\langle\rangle$ is empty word)

$$
\begin{array}{ll}
[\![p]\!](w) & = p(\mathbf{head}\ w) \wedge |w| = 1 \\
[\![!p]\!](w) & = \neg(p(\mathbf{head}\ w)) \wedge |w| = 1 \\
[\![r_1 \mid r_2]\!](w) & = [\![r_1]\!](w) \ \vee\ [\![r_2]\!](w) \\
[\![r_1 ; r_2]\!](w) & = \exists w_1\ w_2.\ w = w_1.w_2 \ \wedge\ [\![r_1]\!](w_1) \ \wedge\ [\![r_2]\!](w_2) \\
[\![r_1 : r_2]\!](w) & = \exists w_1\ s\ w_2.\ w = w_1.s.w_2 \ \wedge\ [\![r_1]\!](w_1.s) \wedge\ [\![r_2]\!](s.w_2) \\
[\![r_1 \&\& r_2]\!](w) & = [\![r_1]\!](w) \ \wedge\ [\![r_2]\!](w) \\
[\![r[*]]\!](w) & = w{=}\langle\rangle \vee \exists w_1 \cdots w_l.\ w{=}w_1.\cdots.w_l \wedge [\![r]\!](w_1) \wedge \cdots \wedge [\![r]\!](w_l)
\end{array}
$$

Example SERE

- Example
  A sequence in which `req` *is asserted, followed four cycles later by an assertion of* `grant`*, followed by a cycle in which* `abortin` *is not asserted.*
- Can this represent by the SERE:

```
req;[*3];grant;!abortin
```


Assertion-based verification (ABV)

- Claimed that assertion based verification:

  "*is likely to be the next revolution in hardware design verification*"
- Basic idea:
  - document designs with formal properties
  - use simulation (dynamic) and model checking (static)
- Problem: too many languages
  - academic logics: LTL, CTL
  - tool-specific industrial versions:
    Intel, Cadence, Motorola, IBM, Synopsys
- What to do? Solution: a competition!
  - run by Accellera organisation
  - results standardised by IEEE
  - lots of politics

IBM's *Sugar* and Accellera's PSL

- *Sugar 1*: property language of IBM RuleBase checker
  - CTL plus *Sugar Extended Regular Expressions* (SEREs)
- Competition finalists: IBM's *Sugar 2* and Motorola's *CBV*
  - Intel/Synopsys ForSpec eliminated earlier (apparently industry politics involved)
- *Sugar 2* is based on LTL rather than CTL
  - has CTL constructs: "Optional Branching Extension" (OBE)
  - has clocking constructs for temporal abstraction
- Accellera purged "Sugar" from it property language
  - the word "Sugar" was too associated with IBM
  - language renamed to PSL
  - SEREs now *Sequential Extended Regular Expressions*
- Lobbying to make PSL more like ForSpec (align with SVA)

PSL Foundation Language (FL)

- Syntax:

f ::=	p	(Atomic formula)
	`!`f	(Negation)
	f_1 `or` f_2	(Disjunction)
	`next` f	(successor)
	$\{r\}(f)$	(Suffix implication: r a SERE)
	$\{r_1\}$ `|->` $\{r_2\}$	(Suffix next implication: r_1, r_2 SEREs)
	$[f_1$ `until` $f_2]$	(Until)

- Semantics (omits clocking, weak/strong distinction)

$$\begin{array}{ll}
[\![p]\!]_M(\pi) & = p(\pi(0)) \\
[\![!f]\!]_M(\pi) & = \neg([\![f]\!]_M(\pi)) \\
[\![f_1 \text{ or } f_2]\!]_M(\pi) & = [\![f_1]\!]_M(\pi) \vee [\![f_2]\!]_M(\pi) \\
[\![\text{next } f]\!]_M(\pi) & = [\![f]\!]_M(\pi\downarrow 1) \\
[\![\{r\}(f)]\!]_M(\pi) & = \exists w\ \pi'.\ \pi = w.\pi' \wedge [\![r]\!]_M(w) \wedge [\![f]\!]_M(\pi') \\
[\![\{r_1\}\,|\text{->}\{r_2\}]\!]_M(\pi) & = \exists w_1\ \pi'.\ \pi = w_1.\pi' \wedge \{r_1\}(w_1) \\
& \quad \Rightarrow \exists w_2 \pi''.\ \pi' = w_2.\pi'' \wedge \{r_2\}(w_2) \\
[\![[f_1 \text{ until } f_2]]\!]_M(\pi) & = \exists i.\ [\![f_2]\!]_M(\pi\downarrow i) \wedge \forall j.\ j<i \Rightarrow [\![f_1]\!]_M(\pi\downarrow j)
\end{array}$$

- There is also an Optional Branching Extension (OBE)
  - completely standard CTL: **EX**, **E[–U–]**, **EG** etc.

Combining SEREs with LTL formulas

- Formula $\{r\}f$ means LTL formula f true after SERE r
- Example

  After a sequence in which `req` *is asserted, followed four cycles later by an assertion of* `grant`*, followed by a cycle in which* `abortin` *is not asserted, we expect to see an assertion of* `ack` *some time in the future.*

- Can represent by

```
always {req;[*3];grant;!abortin}(eventually ack)
```

- where `eventually` is LTL future operator, so:

```
eventually f  =  [true until f]
```

- N.B. Ignoring strong/weak distinction
  - strong/weak distinction important for dynamic checking
  - semantics when simulator halts before expected event
  - strictly should write `until!`, `eventually!`

SERE examples

- How can we modify

```
always reqin;ackout;!abortin |-> ackin;ackin
```

  so that the two cycles of `ackin` start the cycle after `!abortin`

- Two ways of doing this

```
always{reqin;ackout;!abortin}|->{true;ackin;ackin}
always{reqin;ackout;!abortin}|=>{ackin;ackin}
```

- `|=>` is a defined operator

```
{r1}|=>{r2}  =  {r1}|->{true;r2}
```

- Note: `true` and `T` are synonyms

Examples of defined notations: consecutive repetition

- Define

```
r[+]      = r;r[*]
r[*i]     = |  false[*]  if i=0
            |  r;...;r otherwise (i repetitions)
r[*i..j] = r[*i] | r[*(i+1)] | ... | r[*j]
[+]       = true[+]
[*]       = true[*]
```

- Example

  Whenever we have a sequence of `req` *followed by* `ack`*, we should see a full transaction starting the following cycle. A full transaction starts with an assertion of the signal* `start_trans`*, followed by one to eight consecutive data transfers, followed by the assertion of signal* `end_trans`*. A data transfer is indicated by the assertion of signal* `data`

```
always{req;ack}|=>{start_trans;data[*1..8];end_trans}
```


Fixed number of non-consecutive repetitions

- Example

  Whenever we have a sequence of `req` *followed by* `ack`*, we should see a full transaction starting the following cycle. A full transaction starts with an assertion of the signal* `start_trans`*, followed by eight not necessarily consecutive data transfers, followed by the assertion of signal* `end_trans`*. A data transfer is indicated by the assertion of signal* `data`

- Can represent by

```
always
 {req;ack} |=>
 {start_trans;
  {{!data[*];data}[*8];!data[*]};
  end_trans}
```

- Define: `b[= i] = {!b[*];b}[*i];!b[*]`
- Then have a nicer representation

```
always{req;ack}|=>{start_trans;data[= 8];end_trans}
```

Variable number of non-consecutive repetitions

- Example

  Whenever we have a sequence of `req` *followed by* `ack`*, we should see a full transaction starting the following cycle. A full transaction starts with an assertion of the signal* `start_trans`*, followed by* one to eight *not necessarily consecutive data transfers, followed by the assertion of signal* `end_trans`*. A data transfer is indicated by the assertion of signal* `data`

- Define

```
b[= i..j] = {b[= i]} | {b[= (i+1)]} | ... | {b[= j]}
```

- Then

```
always {req;ack} |=>
       {start_trans;data[= 1..8];end_trans}
```

- These examples are meant to illustrate how PSL/Sugar is much more readable than raw CTL or LTL

Clocking

- Basic idea: `b@clk` samples `b` on rising edges of `clk`
- Can clock SEREs (`r@clk`) and formulas (`f@clk`)
- Can have several clocks
- Official semantics messy due to clocking
- Can 'translate away' clocks by pushing `@clk` inwards
  - rules given in PSL manual
  - roughly: `b@clk` ⇝ `{!clk[*];clk & b}`

Model checking PSL (outline)

- SEREs checked by generating a finite automaton
  - recognise regular expressions
  - these automata are called "satellites"
- FL checked using standard LTL methods
- OBE checked by standard CTL methods
- Can also check formula for runs of a simulator
  - this is dynamic verification
  - semantics handles possibility of finite paths – messy!
- Commercial checkers only handle a subset of PSL

PSL layer structure

- Boolean layer has atomic predicates
- Temporal layer has LTL (FL) and CTL (OBE) properties
- Verification layer has commands for how to use properties
  - e.g. `assert`, `assume`

```
assert always (!en1 & en2))
   |     |          |
   |     |          |
   |     |          |--- Boolean layer
   |     |
   |     |--------------- temporal layer
   |
   |--------------------- verification layer
```

- Modelling layer has HDL constructs for specifying inputs and auxiliary hardware

PSL/Sugar summary

- Combines together LTL, ITL and CTL
- Regular expressions – SEREs
- LTL – Foundation Language formulas
- CTL – Optional Branching Extension
- Relatively simple set of primitives + definitional extension
- Boolean, temporal, verification, modelling layers
- Semantics for static and dynamic verification (needs strong/weak distinction)

Simulation or Event semantics

- HDLs use *discrete event simulation*
  - changes to variables ⇒ threads enabled
  - enabled threads executed non-deterministically
  - execution of threads ⇒ more events
- Combinational thread:

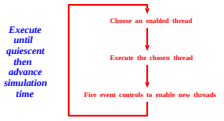
  `always @(`v_1 `or` ⋯ `or` v_n`) v:=E`
  - enabled by any change to $v_1, \ldots, v_n$
- Positive edge triggered sequential threads:

  `always @(posedge clk) v:=E`
  - enabled by *clk* changing to `T`
- Negative edge triggered sequential threads:

  `always @(negedge clk) v:=E`
  - enabled by *clk* changing to `F`

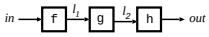
Simulation

- Given
  - a set of threads
  - initial values for variables read or written by threads
  - a sequence of input values
    (inputs are variables not in LHS of assignments)
- *simulation algorithm* ⇒ a sequence of states



- Simulation is non-deterministic

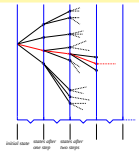
Combinational threads in series



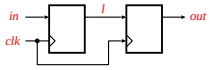
- HDL-like specification:
  `always @(in) l1 := f(in)` ..... thread T1
  `always @(l1) l2 := g(l1)` ..... thread T2
  `always @(l2) out := h(l2)` ..... thread T3
- Suppose *in* changes to *v* at simulation time *t*
  - T1 will become enabled and assign f(*v*) to l_1
  - if l_1's value changes then T2 will become enabled
    (still simulation time *t*)
  - T2 will assign g(f(*v*)) to l_2
  - if l_2's value changes then T will become enabled
    (still simulation time *t*)
  - T3 will assign h(g(f(*v*))) to *out*
  - simulation quiesces
    (still simulation time *t*)
- Steps at same simulation time happen in δ-*time*
  (VHDL jargon)

Semantic gap

- Designers use HDLs and verify via simulation
  - event semantics
- Formal verifiers use logic and verify via proof
  - trace semantics
- Problem: do trace and simulation semantics agree?
- Would like:
  traces = sequences of quiescent simulation states

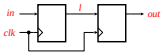


Sequential threads – event semantics



- Consider two Dtypes in series:
  `always @(posedge clk) l := in`
  `always @(posedge clk) out := l`
- If `posedge clk`:
  - both threads become enabled
  - race condition
- Right thread executed first:
  - *out* gets previous value of *l*
  - then left thread executed
  - so *l* gets value input at *in*
- Left thread executed first:
  - *l* gets input value at *in*
  - then right thread executed
  - so *out* gets input value at *in*

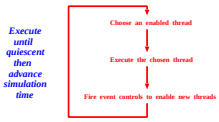
Sequential threads – trace semantics



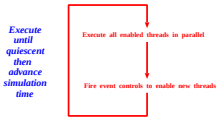
- Trace semantics:
  $(\forall t.\ l(t{+}1) = (\text{Rise } clk\ t \rightarrow in\ t \mid l\ t)) \wedge$
  $(\forall t.\ out(t{+}1) = (\text{Rise } clk\ t \rightarrow l\ t \mid out\ t))$
- Corresponds to right thread executed first
- How to ensure event and trace semantics agree?
- Method 1: use non-blocking assignments:
  `always @(posedge clk) l <= in;`
  `always @(posedge clk) out <= l;`
  - non-blocking assignments (<=) in Verilog
  - RHS of all non-blocking assignments first computed
  - assignments done at end of simulation cycle
- Method 2: make simulation cycle VHDL-like

Verilog versus VHDL simulation cycles

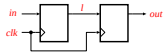
- Verilog-like simulation cycle:



- VHDL-like simulation cycle:



VHDL event semantics



- Recall HDL:

```
always @(posedge clk) l := in
always @(posedge clk) out := l
```

- If `posedge` *clk*:
  - both threads become enabled
- VHDL semantics:
  - both threads executed in parallel
  - *out* gets previous value of *l*
  - in parallel *l* gets value input at *in*
- Now no race
- Event semantics matches trace semantics
- What about combinational threads?