<table>
<thead>
<tr>
<th><strong>Title:</strong></th>
<th><em>Temporal Logic and Model Checking</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lecturer:</strong></td>
<td>Mike Gordon</td>
</tr>
<tr>
<td><strong>Class:</strong></td>
<td>Computer Science Tripos, Part II</td>
</tr>
<tr>
<td><strong>Term:</strong></td>
<td>Easter Term 2011</td>
</tr>
<tr>
<td><strong>First Lecture:</strong></td>
<td>12:00 on Thursday, 28 April, 2011</td>
</tr>
<tr>
<td><strong>Location:</strong></td>
<td>Lecture Theatre 2, WGB</td>
</tr>
<tr>
<td><strong>Duration:</strong></td>
<td>Eight lectures</td>
</tr>
</tbody>
</table>
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 \textit{states} and a binary relation \(R\) on \(S\), called the \textit{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](image)

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: \(R : S \rightarrow (S \rightarrow \mathbb{B})\), where \(\mathbb{B} = \{\text{true}, \text{false}\}\). When we write \(R s s'\) we mean that this equals \textit{true}, i.e. \(R s s' = \text{true}\). 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
$q\overline{0}, a0, or0$ and $a1$ are determined by the values of $q0$ and $dack$ (this should also
be obvious from the circuit diagram).

\[
\begin{align*}
q\overline{0} &= \neg q0 \\
a0 &= q\overline{0} \land dack \\
or0 &= q0 \lor a0 \\
a1 &= dreq \lor or0
\end{align*}
\]

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

\[
S_{RCV} = \mathbb{B} \times \mathbb{B} \times \mathbb{B}
\]

and the transition relation by:

\[
R_{RCV} (dreq, q0, dack) (dreq', q0', dack') =
(q0' = dreq) \land (dack' = (dreq \land (q0 \lor (\neg q0 \land dack))))
\]

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 $R_{RCV}$ 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.

```plaintext
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 $(S_{DIV}, R_{DIV})$ 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:

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

and the transition relation \( R_{\text{DIV}} \) is characterised by:

\[
R_{\text{DIV}} (0, x, y, r, q) (1, x, y, x, q)
\]
\[
R_{\text{DIV}} (1, x, y, r, q) (2, x, y, r, 0)
\]
\[
R_{\text{DIV}} (2, x, y, r, q) ((\text{if } y \leq r \text{ then } 3 \text{ else } 5), x, y, r, q)
\]
\[
R_{\text{DIV}} (3, x, y, r, q) (4, x, y, (r - y), q)
\]
\[
R_{\text{DIV}} (4, x, y, r, q) (3, x, y, r, (q + 1))
\]

which is just a compact way of writing:

\[
(\forall x \ y \ r \ q. R_{\text{DIV}} (0, x, y, r, q) (1, x, y, x, q))
\]
\[
\land
\]
\[
(\forall x \ y \ r \ q. R_{\text{DIV}} (1, x, y, r, q) (2, x, y, r, 0))
\]
\[
\land
\]
\[
(\forall x \ y \ r \ q. R_{\text{DIV}} (2, x, y, r, q) ((\text{if } y \leq r \text{ then } 3 \text{ else } 5), x, y, r, q))
\]
\[
\land
\]
\[
(\forall x \ y \ r \ q. R_{\text{DIV}} (3, x, y, r, q) (4, x, y, (r - y), q))
\]
\[
\land
\]
\[
(\forall x \ y \ r \ q. R_{\text{DIV}} (4, x, y, r, q) (3, x, y, r, (q + 1)))
\]

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:

\[
\forall pc' \ x' \ y' \ r' \ q'. \neg (R_{\text{DIV}} (5, x, y, r, q) (pc', x', y', r', q'))
\]

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

\[
\forall x \ y \ r \ q. R_{\text{DIV}} (5, x, y, r, q) (5, x, y, r, q)
\]

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 \( R_{\text{DIV}} \) is deterministic in that for any state \( s \) there is at most one state \( s' \) such that \( R_{\text{DIV}} s \rightarrow s' \). 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:
```

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:

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

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

\[
\begin{align*}
R_{JM1} (0, pc_2, 0, x) & (1, pc_2, 1, x) & R_{JM1} (pc_1, 0, 0, x) (pc_1, 1, 1, x) \\
R_{JM1} (1, pc_2, lock, x) & (2, pc_2, lock, 1) & R_{JM1} (pc_1, 1, lock, x) (pc_1, 2, lock, 2) \\
R_{JM1} (2, pc_2, 1, x) & (3, pc_2, 0, x) & R_{JM1} (pc_1, 2, 1, x) (pc_1, 3, 0, x)
\end{align*}
\]

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 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 \(P : S \rightarrow \mathbb{B}\). Atomic properties do not depend on the transition relation.

For example, for the model RCV we can define atomic properties \(\mathsf{Dreq}, \mathsf{Q0}\) and \(\mathsf{Dack}\) 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:

\[
\begin{align*}
\text{Dreq}(dreq, q0, dack) &= (dreq = 1) \\
\text{Q0}(dreq, q0, dack) &= (q0 = 1) \\
\text{Dack}(dreq, q0, dack) &= (dack = 1)
\end{align*}
\]

Examples of atomic properties for the model DIV are:

\[
\begin{align*}
\text{AtEnd} \,(pc, x, y, r, q) &= (pc = 5) \\
\text{InLoop} \,(pc, x, y, r, q) &= (pc \in \{3, 4\}) \\
\text{YleqR} \,(pc, x, y, r, q) &= (y \leq r) \\
\text{Invariant} \,(pc, x, y, r, q) &= (x = r + (y \times q))
\end{align*}
\]

\text{AtEnd} \text{ 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 \text{ 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 \textit{computation tree}. If the model is deterministic, then the tree will be linear – i.e. will just be a \textit{path}, where a path is defined to be a sequence of states \(s_0s_1s_2\cdots\) such that for all \(i\): \(r \, s_i \, s_{i+1}\). Paths are also called \textit{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 \( \pi \) 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 \( \pi : \mathbb{N} \rightarrow S \), and hence we write \( \pi(i) \) for the \( i \)th element of \( \pi \).

We define a predicate \( \text{Path} \) so that \( \text{Path} \ R \ s \ \pi \) is true if and only if \( \pi \) is a path starting at \( s \):

\[
\text{Path} \ R \ s \ \pi = (\pi(0) = s) \land \forall i. \ R(\pi(i)) (\pi(i+1))
\]

Mathematically \( \text{Path} \) is a function:

\[
\text{Path} : (S \rightarrow S \rightarrow \mathbb{B}) \rightarrow S \rightarrow (\mathbb{N} \rightarrow S) \rightarrow \mathbb{B}
\]

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 \( \text{dreq} \) rises, then it continues high, until it is acknowledged by a rise on \( \text{dack} \).
- If \( \text{dreq} \) falls, then it will continue low until \( \text{dack} \) 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 \( \text{AtEnd} \) is true then \( \text{Invariant} \) is true and \( \text{YleqR} \) is not true.
CHAPTER 1. INTRODUCTION AND OVERVIEW

- On every path there is a state where \( \text{AtEnd} \) is true.

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

- On any path if there exists a state where \( \text{YleqR} \) is true then there is also a state where \( \text{InLoop} \) is true.

An example property of the \( \text{JM1} \) example is:

- If initially \( \text{LOCK} = 0 \) 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


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 ...
- \(Rs\) means \(s'\) can be reached from \(s\) in one step
  - here \(R:S \rightarrow (S - B)\) (where \(B = \{true, r\}\))
  - more conventional to have \(R \subseteq S \times S\), which is equivalent

A simple example model

- A simple model: \(\{(0, 1, 2, 3), x.R, R = n+1(mod 4)\}\)
  - where \("x.R \ldots x.\"\) is the function mapping \(x\) to \(\ldots x\)
- so \(R(n, n) = (n, n+1(mod 4))\)
- e.g. \(R(0, 1) = R(1, 2) = R(2, 3) = R(3, 0)\)

DIV: a software example

- Perhaps a familiar program:
  
<table>
<thead>
<tr>
<th>pc</th>
<th>x</th>
<th>y</th>
<th>r</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q &lt; 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>WHILE Q &lt; 4 DO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S := S + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Q = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- State \((pc, x, y, r, q)\)
  - \(pc \in \{0, 1, 2, 3, 4, 5\}\) program counter
  - \(x, y, r, q, S\) are the values of \(x, y, r, q\)

- Model \((S, R_{4,5})\) where:
  
  \(S = \{0, 1, 2, 3, 4, 5\}\) \(\subseteq \mathbb{Z}\)
  
  \(R_{4,5} = \{(pc, x, y, r, q) \ni (pc, x', y', r', q') = \)
  
  \(\quad (pc = 0) \Rightarrow ((pc, x', y', r', q') = (1, x, y, r, q)) \land \)
  
  \(\quad (pc = 1) \Rightarrow ((pc, x', y', r', q') = (2, x, y, r, 0)) \land \)
  
  \(\quad (pc = 2) \Rightarrow ((pc, x', y', r', q') \land \)
  
  \(\quad (pc = 3) \Rightarrow ((pc, x', y', r', q') = (4, x, y, r, 0)) \land \)
  
  \(\quad (pc = 4) \Rightarrow ((pc, x', y', r', q') = (3, x, y, r, q+1)) \land \)

Might be extracted from:

[Appendix: slides]
Deriving a transition relation from a state machine

- State machine: \( R(S, \delta) \) is a function:
  - \( \delta \) is a set of inputs
  - \( S \) is a set of states

- Transition relation: \( \delta(s, I) = (s' = \delta(s, I)) \)
  - \( I \) is arbitrary: determined by environment not machine

- Deterministic machine, non-deterministic transition relation
  - Inputs unspecified (determined by environment)
  - So-called “input non-determinism”

Non-deterministic models are very common, e.g. from:

- Example of a handshake circuit:
  - Non-deterministic:
    - “Input non-determinism”

Atomic properties (properties of states)

- Atomic properties are true or false of individual states
  - An atomic property \( p \) is a function \( p : S \rightarrow \{0, 1\} \)
  - Can also be regarded as a subset of state: \( p \subseteq S \)

- Example atomic properties of RCV
  - \( \text{dreq}(q_0, q_0, dack) \)
  - \( \text{dack}(q_0, q_0, dack) \)

- Example atomic properties of RSV
  - \( \text{Start}(p, c, x, r, q) \)
  - \( \text{End}(p, c, x, r, q) \)
  - \( \text{InLoop}(p, c, x, r, q) \)
  - \( \text{Invariant}(p, c, x, r, q) \)

- Relationships between Boolean values on wires:
  - No successor when \( 0 \)
  - Atomic property
  - Partial

- At most one successor state

Model behaviour viewed as a computation tree

- Atomic properties viewed as 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)
A path of $\phi$ is represented by a function $\psi : N \rightarrow S$
- $\psi(n)$ is the $n$th element of $\phi$. (first element is $\psi(0)$)
- might sometimes write $\psi$ instead of $\psi(n)$
- $\psi(n)$ is the $n$th tail of $\phi$ so $\psi(n) = \phi(n + 1)$
- successive states in a path must be related by $\phi$

Path $R s = s$ if and only if it is a path starting at $s$:
Path $Rs = (\forall i \in N. R s(i)) (\psi(i+1))$
where:
Path $(S \rightarrow S \rightarrow B) \rightarrow S \rightarrow \neg S \rightarrow N \rightarrow S \rightarrow B$

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

Reachability
- $Rs \models s'$ means $s'$ reachable from $s$ in one step.
- $R^n s \models s'$ means $s'$ reachable from $s$ in $n$ steps.
  - $R^0 s = s$
  - $R^n s \models s' \Rightarrow R^{n+1} s R s' = R^n s\models R^n s' = R^n s'$
- $R^1 s \models s'$ means $s'$ reachable from $s$ in finite steps.
- $R s \models s$ in $n$ steps.
- $R^0 s \models s\models\psi(n)$
- The set of states reachable from $s$ is $\{s' \models R^1 R s R s'\}$
- Verification problem: all states reachable from $s$ satisfy $p$.
  - e.g. all states reachable from $0, 0, 0, 0$ satisfy $\neg \textit{NotAt11}$
  - i.e. $\forall s', R^n s(0, 0, 0, 0) \models \neg \textit{NotAt11}(s')$

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 propositions.
  - if $p \in AP$ then $\models S = \emptyset$
  - $\models T \in AP$ where $\forall a \in S \models T(a) = \text{true}$ and $\forall a \in S \models T(a) = \text{false}$
- 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 $(S, S_0, R, AP)$ instead of just $(S, R)$.
- Sometimes $AP$ is omitted; one says "Kripke structure over $AP$".
- Model checking computes whether $(S, S_0, R, AP) \models \phi$
  - $\phi$ is a property expressed in a property language.
  - informally $M \models \phi$ means "$\phi$ 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 \((S, R, s_0)\) or \((S, R, S_0)\) sometimes called transition systems
- We called \((S, R, S_0)\) a Kripke structure
- Clarke et al. define a Kripke structure as \((S, S_0, R, P)\)
- \(S\) a given set of "atomic propositions" interpreted by a \(L\)
- \(R\) is a relation on \(S\)
- \(S_0\) is a set of initial states
- \(P\) is a set of "atomic propositions"
- Clearly \(P\) means \(L\)
- Let \(M\) be a model
- Reachable states of \(M\) is \(\{(s) : s \in S, R^* s\}\)
- Compute \(R^*\) as \(R \cup R \circ R \cup \ldots\)
- State represented by a triple of Booleans \((\text{dreq}, \text{b}, \text{dack})\)
- A model of \(\text{RCV}\) is \(M_{\text{RCV}}\), where:
  \[ M = (S_{\text{RCV}}, (1, 1, 1); \text{RCV}; \phi) \]
  and
  \[ \text{RCV}(\text{dreq}, \text{b}, \text{dack}) = (\text{dreq} \land \text{b} \land \text{dack}) \]
  Graph of the transition relation:
- Does the algorithm terminate?
  - yes, if set of states is finite, because then no infinite chains:
    \(S_0 \subset S_1 \subset \ldots \subset S_n \subset \ldots\)
- How to represent \(S_0, S_1, \ldots\)?
  - explicitly (e.g. lists or something more clever)
  - symbolic expression
- Huge literature on calculating set of reachable states

Minimal property language: \(\phi\) is \(\text{GA}\) where \(p \in AP\)

- Assume \(M = (S, S_0, R, AP)\)
- Reachable states of \(M\) are \(\{(s) : s \in S, R^* s\}\)
  - i.e. the set of states reachable from an initial state
  - define Reachable \(M = (\exists s_0 \in S_0, R^* s)\)
- Consider properties \(\phi\) of form \(\text{GA} p\) where \(p \in AP\)
  - \(\text{GA}\) stands for "Globally Always"
- If \(M, s_0 \models \text{GA} p\) means \(p\) true of all reachable states of \(M\)
- If \(M = (S, S_0, R, AP)\) then \(M, s_0 \models \phi\) formally defined by:
  \[ M, s_0 \models \phi \Longleftrightarrow (\forall s \in \text{Reachable}(M) \models \phi) \]
- Possible states for \(\text{RCV}\):
  \([000, 001, 010, 011, 100, 101, 110, 111]\)
  where \(b_0 b_1 b_2\) denotes state
  \[ f = b_0 \land \phi = b_0 \land \text{dack} = \text{dack} \]
- Graph of the transition relation:
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
  - $\neg v$ = if $v$ then 0 else 1

- Example: BDDs of variable $v$ and $\neg v$

- Example: BDDs of $v_1 \land v_2$ and $v_1 \lor v_2$

Computing Reachable $M_{BCV}$

- Define:
  $\delta_0 = \{b_0 b_1 b_2 | b_0 b_1 b_2 \in \{000, 001, 010, 011, 100, 101, 110, 111\} \}$
  $\delta_{i+1} = \delta_i \cup \{ d' d'' | d' \in \delta_i, \nu_{BCV} \land d'' \}$
  $= \delta_i \cup \{ b_0 b_1 b_2 \in \delta_i, (b_0 = b_2) \land (b_1 = b_0 \lor (b_2 = 0)) \}$

Model checking $M_{BCV} \models GAP$

- $M = (S_{BCV}, R_{BCV}, AP)$
- If $p \in AP$ then $p : S_{BCV} \rightarrow \mathbb{B}$

- To check $M_{BCV} \models GAP$
  - compute $\text{Reachable } M_{BCV} = \{111, 011, 000, 100, 010, 110\}$
  - check $\text{Reachable } M_{BCV} \subseteq \{x | p(x)\}$, i.e. check:
    - $\rho(111) = 1$
    - $\rho(011) = 1$
    - $\rho(000) = 1$
    - $\rho(100) = 1$
    - $\rho(101) = 1$
    - $\rho(110) = 1$

Computing Reachable $M_{BCV}$ (continued)

- Compute:
  $\delta_0 = \{111\}$
  $\delta_1 = \{111 \cup \{011\\}\}$
  $\delta_2 = \{111, 011 \cup \{011\\}\}$
  $\delta_3 = \{111, 011, 000, 100, 010, 110\}$
  $\delta_4 = \{111, 011, 000, 100, 010, 110\}$

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

Symbolically Boolean model checking of reachability

- Assume states are $n$-tuples of Booleans $(b_1, \ldots, b_n)$
  - $b_i \in \mathbb{B} = \{\text{true}, \text{false}\}$
  - $S = 2^n$, so $S$ is finite: $2^n$ states

- Assume $n$ distinct Boolean variables: $v_1, \ldots, v_n$
  - e.g. if $n = 3$ then could have $v_1 = x, v_2 = y, v_3 = z$

- Boolean formula $f(v_1, \ldots, v_n)$ represents a subset of $S$
  - $R(v_1, \ldots, v_n)$ only contains variables $v_1, \ldots, v_n$
  - $R(b_1, \ldots, b_n)$ denotes result of substituting $b_i$ for $v_i$
  - $f(v_1, \ldots, v_n)$ determines $(b_1, \ldots, b_n)$ if $(b_1, \ldots, b_n) \in S$

- Example: $\neg(x = y)$ represents $\{\text{true, false} \}$

- Transition relations also represented by Boolean formulae
  - e.g. $R_{BCV}$ represented by:
    - $(p(g') = \neg \text{dreq}) \land \land (\neg \text{dack} = \neg \text{dreq} \land (p(g') \lor \neg p(g) \land \text{dack}))))$
BDD of a transition relation

- BDDs of
  \[ (v' = (v_1 = v_2)) \land (v'' = (v_1 \neq v_2)) \]
  with two different variable orderings
  
  \[ \text{Exercise: draw BDD of } R_{\text{fork}} \]

Standard BDD operations

- If formula \( f \) represents sets \( S_1, S_2 \), respectively
  then \( f_1 \land f_2 \) represents \( S_1 \cup S_2 \), \( f_1 \lor f_2 \) represents \( S_1 \cap S_2 \), respectively
- Standard algorithms compute boolean operation on BDDs
-Abbreviate \((v_1, \ldots, v_k)\) to \( \mathbf{v} \)

- If \( f(\mathbf{v}) \) represents \( S \) and \( g(\mathbf{v}) \) represents \((R(\mathbf{v})) \)
  then \( h(\mathbf{u} \oplus \mathbf{v}) \) represents \((v_1 = \mathbf{u} \land \mathbf{v} \in S \land \mathbf{R} \mathbf{v}) \)
- Can compute BDD of \( h(\mathbf{v}) \) from BDD of \( h(\mathbf{u} \land \mathbf{v}) \)
  - e.g. BDD of \((\mathbf{u} = \mathbf{v_1} \lor \mathbf{v_2}) \land \mathbf{R} (\mathbf{v}) \)
  - From BDD of formula \((v_1, \ldots, v_k)\) can compute \( b_1, \ldots, b_n \) such that if \( v_1 = b_1, \ldots, v_k = b_k \) then \( f(b_1, \ldots, b_k) \) is true
  - \( b_1, \ldots, b_n \) is a satisfying assignment (SAT problem)
- used for counterexample generation (see later)

Reachable States via BDDs

- Assume \( M = (S, \mathcal{S}, R, \mathcal{A}) \) and \( S = \mathbb{B}^n \)
- Represent \( \mathcal{S} \) by Boolean formula \( g(\mathbf{v}, \mathbf{v'}) \)
- Iteratively define formula \( \delta(\mathbf{v}) \) representing \( S \)
  \[ \delta(\mathbf{v}) = \text{ formula representing } S \]
  \[ \delta_{n+1}(\mathbf{v}) = \delta_n(\mathbf{v}) \lor (\mathcal{U} \delta_n(\mathbf{u}) \land g(\mathbf{u}, \mathbf{v})) \]
- Let \( R_0, R_n \) be BDDs representing \( \delta(\mathbf{v}), g(\mathbf{v}, \mathbf{v'}) \)
- Iteratively compute BDDs \( R_n \) representing \( R_{n+1} \)
  \[ \delta_{n+1} = \delta_n \lor (\mathcal{U} \delta_n (\mathbf{u}) \land g(\mathbf{u}, \mathbf{v})) \]
  - efficient using (blue underlined) standard BDD algorithms
- BDD \( R_n \) only contains variables \( \mathbf{v} \); represents \( S \subseteq S \)
- At each iteration check \( R_{n+1} = R_n \); efficient using BDDs
  - when \( R_{n+1} = R_n \) can conclude \( R_n \) represents \( \text{Reachable } M \)
  - we call this BDD \( R_n \) in a later slide (i.e. \( R_0 = R_n \))

Example BDD optimisation: disjunctive partitioning

Three state machine in parallel

\[ \mathcal{A}_x, \mathcal{A}_y, \mathcal{A}_z : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \]

- Transition relation (asynchronous interleaving semantics):
  \[ R(x, y, z) (x', y', z') = \]
  \[ (x' = x \land y = y' \land z = z') \lor \]
  \[ (x = x' \land y = y' \land z = z') \]

- Avoiding building big BDDs

Transition relation for three machines in parallel

\[ R(x, y, z) (x', y', z') = \]
\[ (x' = x \land y = y' \land z = z') \lor \]
\[ (x = x' \land y = y' \land z = z') \]

- Recall symbolic iteration:
  \[ \delta_{n+1}(\mathbf{v}) = \delta_n(\mathbf{v}) \lor (\mathcal{U} \delta_n(\mathbf{u}) \land g(\mathbf{u}, \mathbf{v})) \]

- For the 3 machine example this is (see next slide):
  \[ \delta_{n+1}(x, y, z) = \]
  \[ g(x, y, z) \lor (\mathcal{U} \delta_n(\mathbf{u}) \land g(\mathbf{u}, \mathbf{v})) \]

- Don't need to calculate BDD of \( R \)

Disjunctive partitioning

- \[ g(x, y, z) \land R(x, y, z) (x', y', z') \]
  \[ = g(x, y, z) \land (x = x \land y = y' \land z = z') \lor \]
  \[ (x = x' \land y = y' \land z = z') \]

- \[ g(x, y, z) \land R(x, y, z) (x', y', z') \]
  \[ = g(x, y, z) \land (x = x \land y = y' \land z = z') \lor \]
  \[ (x = x' \land y = y' \land z = z') \]

- \[ g(x, y, z) \land R(x, y, z) (x', y', z') \]
  \[ = g(x, y, z) \land (x = x \land y = y' \land z = z') \lor \]
  \[ (x = x' \land y = y' \land z = z') \]

- \[ g(x, y, z) \land R(x, y, z) (x', y', z') \]
  \[ = g(x, y, z) \land (x = x \land y = y' \land z = z') \lor \]
  \[ (x = x' \land y = y' \land z = z') \]
Verification and counterexamples

- Typical safety question:
  - Is property true in all reachable states?
  - I.e. check if \( \neg p \) \( \Rightarrow \) CST

- Check using BDDs
  - Compute BDD \( B_0 \) of Reachable M
  - Compute BDD \( B_n \) of \( p(v) \)
- Check if BDD of \( R \) \( \Rightarrow \) B is the single node

- Valid because \( \neg p \) can be represented by a unique BDD
- If BDD is not \( \neg p \) can get counterexample

Generating counterexamples

- BDD algorithms can find satisfying assignments (SAT)
  - \( M = (S, S_0, R, \neg p) \) and \( S_0, S_1, \ldots, S_n, \neg p, S_0 \) as earlier
  - Suppose \( R_{\neg p} \neq S_0 \) is not \( \neg p \)
  - Must exist a state \( s \in \text{Reachable } M \) such that \( \neg (p s) \)
  - Let \( B_0 \) be the BDD representing \( p s \)
  - Iterate to find first n such that \( R_0 \Rightarrow B_n \)
  - Use SAT find \( B_0 \) such that \( (R_0 \Rightarrow B_n) \Rightarrow B_{n+1} \)
  - For \( 0 \leq i \leq n \) find \( B_i \) such that \( (R_i \Rightarrow B_0) \Rightarrow (b_{i+1} \Rightarrow) \)
  - \( b_0 \ldots b_m \ldots b_n \) is a counterexample trace
  - Sometimes can use partitioning to avoid constructing \( B_{n+1} \)

Example (from an exam)

Consider a 3x3 array of switches

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

(a) Check if 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 on.

(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 sequence of switches to toggle to get from the initial to final state. [5 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 \((v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)\), where \( v_i \in \{\text{ON, OFF}\} \)

A transition relation is then defined by:

- \( \text{Trans}(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8) \)
- \( \text{Trans}(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8) \)
- \( \text{Trans}(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8) \)
- \( \text{Trans}(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8) \)
- \( \text{Trans}(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8) \)
- \( \text{Trans}(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8) \)
- \( \text{Trans}(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8) \)
- \( \text{Trans}(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8) \)

(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 sequence of switches to toggle to get from the initial to final state. [5 marks]

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

Solution (continued)

Predicates \( \text{Init}, \text{Final} \) characterising the initial and final states, respectively, are defined by:

- \( \text{Init}(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8) \) \( \Rightarrow \) \( v_0 \land v_1 \land v_2 \land v_3 \land v_4 \land v_5 \land v_6 \land v_7 \land v_8 \)
- \( \text{Final}(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8) \) \( \Rightarrow \) \( v_0 \land v_1 \land v_2 \land v_3 \land v_4 \land v_5 \land v_6 \land v_7 \land v_8 \)

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:

- \( \forall i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8\} \)

Properties

- \( \forall s \in S_0, R \Rightarrow p s \) means \( p \) true in all reachable states
- Might want to verify other properties
  1. \( (R_{\neg p} \Rightarrow \neg p) \text{ holds infinitely often} \)
  2. From any state it is possible to get to a state where \( \neg \text{Ripart} \) holds
  3. After a three or more consecutive occurrences of \( \neg \text{Ripart} \) there will eventually be an \( \text{Ripart} \)
- Temporal logic can express such properties
- There are several temporal logics in use
  - \( \text{LTL} \) is good for the first example above
  - \( \text{CTL} \) is good for the second example
  - \( \text{PSL} \) 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”)

- Originally devised for investigating the relationship between tense and modality attributed to the Megarian philosopher Diaclonius Corax (ca. 345-260 BCE).
- A. N. Prior, his wife, recalls “I remember his waking me one night in 1959, coming and sitting in 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

Many different languages of temporal statements
- Linear time (LTL)
- Branching time (CTL)
- Infinite intervals (SEREs)
- Industrial languages (PSL, SVA)

If $\pi$ is a path starting from a member of $M$, then we define $[\pi]_M$ (successor)

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 in 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”.

CS issues different from philosophical issues
- Moshe Vardi: “Branching vs. Linear Time: Final Showdown”

Temporal logic formulae (statements)

- Many different languages of temporal statements
  - Linear time (LTL)
  - Branching time (CTL)
  - Infinite 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 in 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”.
- CS issues different from philosophical issues
- Moshe Vardi: “Branching vs. Linear Time: Final Showdown”

Grammar of well formed formulae (wffs) $\phi$

- $\phi ::= p$ (Atomic formulae $p \in AP$)
- $\phi \lor \phi_2$ (Disjunction)
- $X\phi$ (successor)
- $F\phi$ (sometime thereafter)
- $G\phi$ (always)
- $[\phi]$ (Until)

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

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

- $[\phi]_M(\pi)$ is the application of function $[\phi]_M$ to path $\pi$
  - thus $[\phi]_M(\pi) : (\pi, \pi_0) \rightarrow \mathbb{B}$
- Let $M = (S, R, S_0, AP)$
  - $[\phi]_M(\pi)$ is defined by structural induction on $\phi$
    - $[\phi]_M(\pi) = p(\pi)$
    - $[\phi\lor\phi]_M(\pi) = [\phi]_M(\pi) \lor [\phi]_M(\pi)$
    - $[X\phi]_M(\pi) = [\phi]_M(\pi)\pi R\pi_0$
    - $[F\phi]_M(\pi) = \exists \eta [\phi]_M(\eta)$
    - $[\phi \Rightarrow \psi]_M(\pi) = \forall \eta [\phi]_M(\eta) \Rightarrow [\psi]_M(\eta)$
    - $[\phi \land \psi]_M(\pi) = [\phi]_M(\pi) \land [\psi]_M(\pi)$
- We look at each of these semantic equations in turn

Temporal logic proof systems also important in CS
- use pioneered by Amir Pnueli (1996 Turing Award)
  - Details differ from Prior’s tense logic – but similar ideas
- snapshot showing he thought one could make a formalised tense logic”.

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

Mike Gordon 43 / 118

Linear Temporal Logic (LTL)

- Grammar of well formed formulae (wff) $\phi$

  $\phi ::= p$ (Atomic formulae $p \in AP$)

- $\phi \lor \phi_2$ (Disjunction)
- $X\phi$ (successor)
- $F\phi$ (sometime thereafter)
- $G\phi$ (always)
- $[\phi]$ (Until)

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

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

- If $M = (S, R, S_0, AP)$ then
  - $M \models \phi$ if there is an $M$-path starting from a $S_0$-Path $\pi \in S$
  - If $M = (S, R, AP)$, then we define $M \models \phi$ to mean:
    - $\phi$ is true on all $M$-paths starting from a member of $S_0$
  - We will define $[\phi]_M(\pi)$ to mean:
    - $\phi$ is true on the $\pi$-path
  - Thus $M \models \phi$ will be formally defined by:
    - $M \models \phi$ if $\forall \pi \in S_0 \exists \pi_0 \in S \text{ Path } R \pi \pi_0 \models [\phi]_M(\pi_0)$
  - It remains to actually define $[\phi]_M$ for all wffs $\phi$

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

- $[\phi]_M(\pi)$ is the application of function $[\phi]_M$ to path $\pi$
  - thus $[\phi]_M(\pi) : (\pi, \pi_0) \rightarrow \mathbb{B}$
- Let $M = (S, R, S_0, AP)$
  - $[\phi]_M(\pi)$ is defined by structural induction on $\phi$
  - $[\phi]_M(\pi) = p(\pi)$
  - $[\phi\lor\phi]_M(\pi) = [\phi]_M(\pi) \lor [\phi]_M(\pi)$
  - $[X\phi]_M(\pi) = [\phi]_M(\pi)\pi R\pi_0$
  - $[F\phi]_M(\pi) = \exists \eta [\phi]_M(\eta)$
  - $[\phi \Rightarrow \psi]_M(\pi) = \forall \eta [\phi]_M(\eta) \Rightarrow [\psi]_M(\eta)$
  - $[\phi \land \psi]_M(\pi) = [\phi]_M(\pi) \land [\psi]_M(\pi)$
- We look at each of these semantic equations in turn

$M \models \phi(\pi) = p(\pi \in S_0)$

- Assume $M = (S, R, S_0, AP)$
- We have:
  - $[p]_M(\pi) = p(\pi \in S_0)$
  - $p$ is an atomic property, i.e. $p \in AP$
  - $\pi : 0 \in S_0 \land \exists \pi_0 \pi R \pi_0$
  - $\phi$ is true if atomic property $p$ holds of state $\pi$ $\neq 0$
  - $[p]_M(\pi)$ means $p$ holds of the first state in path $\pi$

Assumed $\tau \in AP$ with $\tau(\pi) \rightarrow \text{true}$ and $\tau(\pi) \rightarrow \text{false}$

- $[\tau]_M(\pi)$ is always true
- $[\tau]_M(\pi)$ is always false
\[\neg \phi(u) = \neg \{u \ni \phi(u)\}\]
\[\phi \lor \phi_2(u) = \{u \ni \phi(u) \lor \phi_2(u)\}\]

- \(\neg \phi(u)\) true if \(\{u \ni \phi(u)\}\) is not true
- \(\phi \lor \phi_2(u)\) true if \(\{u \ni \phi(u) \lor \phi_2(u)\}\) is true or \(\{u \ni \phi(u)\}\) is true

\[X\phi(u) = \{u \ni \phi(u)\}\]

- \(X\phi(u)\) true iff \(\{u \ni \phi(u)\}\) true starting at the next state of \(u\)

\[\text{Computation Tree Logic (CTL)}\]

**Syntax of CTL well-formed formulae:**
- Sometimes just write \(\nu \phi\) rather than \(\nu (\phi\nu)\)
- LTL formulae \(\phi\) are evaluated on paths - path formulae
- CTL formulae \(\phi\) are evaluated on states - state formulae
### Semantics of CTL
- Assume $M = (S, s_0, R, AP)$ and then define:
  - $[\phi](s) = \rho(s)$
  - $[-\phi](s) = -[-\phi(s)]$
  - $[\pi \land \phi](s) = [\pi](s) \land [\phi](s)$
  - $[\pi \lor \phi](s) = [\pi](s) \lor [\phi](s)$
  - $[\exists R \phi](s) = \exists s' \in R s \cdot [\phi](s)$
  - $[\forall R \phi](s) = \forall s' \in R s \cdot [\phi](s)$
- $[\phi^R](s) = (\forall i < |R|) \exists j < i \cdot [\phi](s_{ij})$

### The defined operator $AF$
- Define $AFo = A[\cdot U s]$
  - $AFo \true \text{ at } s$ if $\true$ is $\true$ somewhere on every $R$-path from $s$
    - $[AFo \phi](s) = A[\phi U s](s)$
      - $\forall s \cdot \text{Path } R s \implies (\exists i \cdot [\phi](s_{ij}))$
    - $\forall s' \cdot \text{Path } R s' \implies (\exists i \cdot [\phi](s_{ij}))$

### The defined operator $EF$
- Define $EFo = E[\cdot U s]$
  - $EFo \true \text{ at } s$ if $\true$ is $\true$ somewhere on some $R$-path from $s$
    - $[EFo \phi](s) = E[\phi U s](s)$
      - $\exists s \cdot \text{Path } R s \implies (\exists i \cdot [\phi](s_{ij}))$

### The defined operator $EG$
- Define $EGo = \neg AF(\neg \phi)$
  - $EGo \true \text{ at } s$ if $\true$ is $\true$ everywhere on some $R$-path from $s$
    - $[EGo \phi](s) = \neg [AF(\neg \phi)](s)$
      - $\neg (\forall i < |R|) \exists j < i \cdot [\phi](s_{ij})$

### The defined operator $AG$
- Define $AGo = \neg EF(\neg \phi)$
  - $AGo \true \text{ at } s$ if $\true$ is $\true$ everywhere on every $R$-path from $s$
    - $[AGo \phi](s) = \neg [EF(\neg \phi)](s)$
      - $\neg (\forall i < |R|) \exists j < i \cdot [\phi](s_{ij})$

### The defined operator $A[\phi W_2]$
- $A[\phi W_2]$ is a ‘partial correctness’ version of $A[\phi U_2]$
  - It is $\true$ at $s$ if all $R$-paths from $s$:
    - $s$, always holds on the path
    - $s'$, holds sometime on the path, and until it does $\phi$ holds

- Define
  - $[A[\phi W_2]](s) = \neg [E[\neg \phi U_2] W_2](s)$
    - $\neg (\exists i < |R|) \exists j < i \cdot [\phi](s_{ij})$
A[φ1,Wφ2] continued (1)

- Continuing:
  - ¬(∃t. Path s t φ)
    - ∨ (Path R s t φ) ∧ ∀j (j < t) → [(φ1 ∧ φ2) u(t)]]
  - ∀t. Path R s t φ
    - (Path R s t φ) ∧ ∀j (j < t) → [(φ1 ∧ φ2) u(t)]]
  - ∀t. Path R s t φ
    - (¬φ1 ∧ ¬φ2 u(t)]] ∧ ∀j (j < t) → [(φ1 ∧ φ2) u(t)]]
  - ∀t. Path R s t φ
    - (∃i. ¬φ1 ∧ ¬φ2 u(t)]] ∧ ∀j (j < t) → [(φ1 ∧ φ2) u(t)]]

A[φ1,Wφ2] continued (2)

- Continuing:
  - (∀φ. Path R s t φ)
    - ∀t. Path R s t φ
      - (¬φ1 ∧ ¬φ2 u(t)]] ∨ ∀j (j < t) → [(φ1 ∧ φ2) u(t)]]
    - ∀t. Path R s t φ
      - (∃i. ¬φ1 ∧ ¬φ2 u(t)]] ∨ ∀j (j < t) → [(φ1 ∧ φ2) u(t)]]
    - ∀t. Path R s t φ
      - (∃i. ¬φ1 ∧ ¬φ2 u(t)]] ∨ ∀j (j < t) → [(φ1 ∧ φ2) u(t)]]

Exercise: explain why this is [A[φ1,Wφ2]](s)?
  - this exercise illustrates the subtlety of writing CTL!

Summary of CTL operators (primitive + defined)

- CTL formulas:
  - p
    - (Atomic formula - p: states→bool)
  - φ1 ∨ φ2
    - (Conjunction)
  - φ1 ∨ φ2
    - (Disjunction)
  - φ1 ⇒ φ2
    - (Implication)
  - AXφ
    - (All successors)
  - EXφ
    - (Some successors)
  - AFφ
    - (Somewhere-→ along all paths)
  - EFφ
    - (Somewhere-→ along some path)
  - AGφ
    - (Everywhere-→ along all paths)
  - EGφ
    - (Everywhere-→ along some path)
  - A[φ1,Wφ2]
    - (Until-→ along all paths)
  - A[φ1,Wφ2]
    - (Until-→ along some path)
  - A[φ1,Wφ2]
    - (Unless-→ along all paths)
  - A[φ1,Wφ2]
    - (Unless-→ along some path)

Example CTL formulas

- EF(Started ∧ ¬Ready)
  - It is possible to get to a state where Started holds but Ready does not hold
- AG(Req ⇒ AF(Ack))
  - If a request Req occurs, then it continues to hold, until it is eventually acknowledged
- AG(AF(DeviceEnabled))
  - DeviceEnabled is always true somewhere along every path starting anywhere: i.e. DeviceEnabled holds infinitely often along every path
- AG(EF(Restart))
  - From any state it is possible to get to a state for which Restart holds

More CTL examples (1)

- AG(Req ⇒ A[(Req U 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?
- AG(Req ⇒ ¬(Ack ⇒ AX(A[(Req U 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)

- $\text{AG}([\text{Enabled}] \rightarrow \text{AG}([\text{Start} \rightarrow \text{A}([\text{Waiting U Ack}]))])$
  - 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
- $\text{AG}([\text{Req}] \land \neg \text{Req} \rightarrow \text{A}([\text{Req} \land \neg \text{Req} \cup \text{U}([\text{Start} \land \neg \text{Req}]]))$  
  - Whenever Req and Req are false, they remain false until Start becomes true with Req, still false
- $\text{AG}([\text{Req}] \rightarrow \text{A}([\text{Ack} \land \neg \text{Req}]))$
  - If Req is true and Ack becomes true one cycle later, then eventually Req will become false

Some abbreviations

- $\text{AX} \varphi = \text{AX}(\text{AX} \cdots (\text{AX} \varphi) \cdots))$
- $\text{A} \varphi = \text{AX}(\varphi \lor \text{A} \varphi \lor \cdots)$
  - $f$ instances of $\text{AX}$
  - $\varphi$ is true on all paths $i$ units of time later
- $\text{ABF} \varphi = \text{AX}(\varphi \lor \text{A} \text{B} \text{F} \varphi)$
  - One cycle after $\text{A} \varphi$, should become true, and then $\text{A} \varphi_i$ becomes true 1 to 6 cycles later and then eventually $\text{Reply}$ becomes true, but until it does Wait holds from the time of $\text{A} \varphi_i$

More abbreviations in ‘Industry Standard’ language PSL

### CTL model checking:

- For LTL path formula $\varphi$ recall that $M \models \varphi$ is defined by:
  $$M \models \varphi = \forall s \in S_0 \land \text{Path } R \models [\varphi]_u[s]$$

- For CTL state formula $\varphi$ the definition of $M \models \varphi$ is:
  $$M \models \varphi = \forall s \in S_0 \land M = [\varphi]_\pi[s]$$

- $M$ common; LTL, CTL formulae $\varphi$ and semantics $[\varphi]_u$ differ

### CTL model checking:

- **CTL model checking algorithm:**
  - compute $[\varphi]_u(s) = \text{true}$ bottom up
  - check $S_0 \subseteq \{ s | [\varphi]_u(s) = \text{true} \}$
  - symbolic model checking represents these sets as BDDs

- **Example:** Checking $\text{EF} p$
  - $\text{EF} p = [\varphi]_u p$
  - holds if $p$ holds along some path

- Let $S_0 = [\varphi]_u p$.
  - $S_0 = \{ s | [\varphi]_u(s) \}$
  - $S_{n+1} = S_n \cup \{ s | \exists s' : R s s' \land s' \in S_n \}$
  - mark all the states satisfying $p$
  - mark all with at least one marked successor
  - repeat until no change
  - $[\varphi]_u p$ is set of marked states
Recall the handshake circuit:

- Other temporal logics
- Represent sets of states with BDDs
- CTL model checking due to Emerson, Clarke & Sifakis
- Possible states for RCV: \(b_0 b_1 b_2\) denotes state \(dreq = b_0 \land qF = b_1 \land dack = b_2\)
- Graph of the transition relation:

Example: RCV

- State represented by a triple of Booleans \((dreq, qF, dack)\)
- A model of RCV is \(M_{RCV}\), where:
  
  \[
  M = (S_{RCV}, \{0, 1, 2\}, \{dreq, qF, dack\}, \delta_{RCV}, (0, 1, 1))
  \]

Model checking \(M_{RCV} \models (d_0 b_1 b_0 b_2)\)

- Define:
  
  \[
  \begin{align*}
  \mathcal{A}_0 &= \{d_0 b_1 b_0 b_2, b_0 b_2 b_1, b_0 b_2 b_2, b_0 b_1 b_0, b_1 b_1 b_0, b_0 b_0 b_0\} \\
  \mathcal{A}_{t+1} &= \mathcal{A}_t \cup \{s \in \mathcal{A}_t \mid dreq \in \mathcal{A}_t \}
  \end{align*}
  \]

Model checking \(M_{RCV} \models (d_0 b_1 b_0 b_2)\) (continued)

- Compute:
  
  \[
  \begin{align*}
  S_0 &= \{111\} \\
  S_1 &= \{111\} \cup \{101, 110\} \\
  S_2 &= \{111, 101, 110\} \cup \{100\} \\
  S_3 &= \{111, 101, 110, 100\} \cup \{000, 001, 010, 011\} \\
  S_4 &= \{111, 101, 110, 100, 000, 001, 010, 011\} \\
  S_5 &= S_4 \land \exists (\mathbf{d}_{t-3})
  \end{align*}
  \]

- Use [EF (][dreq ∧ qF ∧ dack)] \(\mathbf{d}_{t-3}\)
- Use [EF (][dreq ∧ qF ∧ dack)] \(\mathbf{d}_{t-3}\)
- Use [EF (][dreq ∧ qF ∧ dack)] \(\mathbf{d}_{t-3}\)
- 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, Barth & Murata
- SMV (McMillan) is a popular symbolic model checker:
  - [http://www.cs.unm.edu/~mcmillan/smv.html](http://www.cs.unm.edu/~mcmillan/smv.html)
- Other temporal logics
  - CTL*: combines CTL and LTL
  - Engineer friendly industrial languages: PSL, SVA
Consider the property

\[ A \]

is total (\( s_2 X \))

\( \text{AG}(\text{Successor}) \)

\( \text{Negation} \)

\( \text{Defined mutually recursively} \)

\[ F \]

\( \text{for} \)

\( \text{Negation} \)

\( \phi \)

\( \text{but in CTL} \)

\( \text{Consider} \)

\( \text{path formulas} \)

\( \psi \)

\( \phi \)

\( \text{is false} \)

\( \text{consider path} \)

\( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots \)

\( \text{AF} \)

\( \text{Recall:} \)

\( \text{CTL} \) is \( \text{CTL}^* \) with

\( \text{atomic formula} \)

\( \phi \)

\( \text{preceded by} \)

\( A \) or \( E \)

\( \text{LTL} \) can express things that \( \text{CTL} \) can't express

\( \text{A cannot say this in either CTL or LTL!} \)

(\( \text{proof omitted} \))

\( \text{and} \)

\( \text{if} \)

\( \text{a state} \)

\( \text{formulas are true of a state} \)

\( \text{where} \)

\( \text{if} \)

\( \text{path formulas are true of a path} \)

\( \text{AF} \)

\( \text{Recall:} \)

\( \text{CTL} \) is \( \text{CTL}^* \) with

\( \text{atomic formula} \)

\( \phi \)

\( \text{preceded by} \)

\( A \) or \( E \)

\( \text{LTL} \) can express things that \( \text{CTL} \) can't express

\( \text{CTL}^* \) combines \( \text{CTL} \) and \( \text{LTL} \) and can express this property

\( \text{CTL}^* \) semantics

\( \text{Combines \( \text{CTL} \) state semantics with \( \text{LTL} \) path semantics:} \)

\( \text{F} \)

\( \text{negation} \)

\( \phi \)

\( \text{or} \)

\( \text{for} \)

\( \text{CTL}^* \) and \( \text{CTL} \) as \( \text{CTL}^* \)

\( \text{as usual:} \)

\( \text{M} = (S, \text{R}, \text{AP}) \)

\( \text{if} \)

\( \text{is a \( \text{CTL}^* \) state formula:} \)

\( \text{M} \models \phi \iff \exists s \in S \ [s]_M \phi \)

\( \text{if} \)

\( \text{is an \( \text{LTL} \) path formula then:} \)

\( \text{M} \models \phi \iff \exists s \in S \ [s]_M \phi \)

\( \text{if} \)

\( \text{M} \models \phi \iff \exists s \in S \ [s]_M \phi \)

\( \text{if} \)

\( \text{M} \models \phi \iff \exists s \in S \ [s]_M \phi \)

\( \text{The meanings of \( \text{CTL} \) formulas are the same in \( \text{CTL}^* \):} \)

\( \text{M} \models \phi \iff \exists s \in S \ [s]_M \phi \)

\( \text{M} \models \phi \iff \exists s \in S \ [s]_M \phi \)

Exercise: do similar proofs for other \( \text{CTL} \) formulas
Fairness

- May want to assume system or environment is fair
  - Example 1: fair arbiter
    - the arbiter ignores 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 $G(F(P)) \Rightarrow G(F(Q))$
- Can’t say this in CTL
  - why not – what’s wrong with $AG(AF(P)) = AF(AF(Q))$?
  - In CTL* expressible as $AG(F(P)) = G(F(Q))$
  - fair CTL model checking implemented in checking algorithm
  - fair LTL just a fairness assumption like $G(F(P))$
- Fairness is a tricky and subtle subject
  - many kinds of fairness: ‘weak fairness’, ‘strong fairness’ etc
  - exist whole books on fairness

Propositional modal $\mu$-calculus

- You may learn this in Topics in Concurrency
  - $\mu$-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

- Strictly stronger than CTL*
  - expressibility strictly increases as allowed nesting increases
  - need fixed-point operators nested 2 deep for CTL*
  - The $\mu$-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:
    - $r = \rho$ (Atomic formula $\rho \in AP$)
    - $\neg r$ (Negated atomic formula $\rho \in AP$)
    - $r_1 \land r_2$ (Conjunction)
    - $r_1 \lor r_2$ (Disjunction)
    - $\rho_1 \Rightarrow \rho_2$ (Implication)
    - $\rho_1 \Rightarrow \rho_2$ (Length matching conjunction)
  - Semantics:
    - $|w|$ ranges over finite lists of states
    - $\exists r_1$ (length of $w$)
    - $w_1 w_2$ is concatenation of $w_1$ and $w_2$; $\epsilon$ is empty word
    - $[\rho][w] = [\rho][\text{head } w] \lor |w| = 1$
    - $[\rho][w] = [\rho][\text{head } w] \land |w| = 1$
    - $[\rho][w] = [\rho][w_1] \lor [\rho][w_2]$
    - $[\rho][w_1] = [\rho][w_2] = [\rho][w_3] = [\rho][w_4] = [\rho][w_5]$
    - $[\rho][w_1] = [\rho][w_2] = [\rho][w_3] = [\rho][w_4] = [\rho][w_5]$
    - $[\rho][w] = [\rho][w_1] \lor [\rho][w_2] = [\rho][w_3] \lor [\rho][w_4] = [\rho][w_5]$

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;[+]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 organization
  - 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 its 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:
  - Syntax:
    - $F := μ \mid ν \mid \overline{F} \mid F_1 \
      \mid F_2 \mid \langle\ldots\rangle$ (Atomic formula)
    - $\Phi$ is LTL future operator, so:
      - $F_1 \Rightarrow F_2 (\text{successor})$
      - $\langle F \rangle (\text{disjunction})$
      - $\langle F_1 \rangle \Rightarrow \langle F \rangle_2 (\text{suffix implication / a SERE})$
      - $\langle F \rangle_1 \Rightarrow \langle F \rangle_2 (\text{suffix implication / i d SEREs})$
      - $\langle F \rangle (\text{unit})$
  - Semantics (omits clocking, weak/strong distinction)
    - $\langle F \rangle = [ F ]_w (F)$
    - $[ F ]_w = [ F (w)]_w$ (strong)
    - $[ F ]_a (F) = [ F (a)]_a$ (weak)
    - $[ F ]_w (F) = \exists (\exists ! F (w)) (F)$ (strong)
    - $[ F ]_a (F) = \exists (\exists ! F (a)) (F)$ (weak)
  - There is also an Optional Branching Extension (OBE)
    - completely standard CTL: $E_1 \cup \cdots \cup E_n$

Combining SEREs with LTL formulas

- Formula $(/r)$ means LTL formula $r$ 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
  - $\text{always} \langle \text{req,}|1;\text{grant,}!\text{abortin} \rangle \langle \text{eventually \text{ack}} \rangle$
  - $\langle \text{eventually} r \langle /r \rangle \Rightarrow \langle \text{true until } f \rangle \rangle$
- N.B. Ignoring strong/weak distinction
  - strong/weak distinction important for dynamic checking
  - semantics when simulator halts before expected event
  - strictly should write $\text{until} 1, \text{eventually} 1 ?$

Examples of defined notations: consecutive repetition

- Define
  - $r[1] = r[1]$ \n  - $r[1+i] = r[r[i]] \ldots r[r[i]]$ if $i > 0$
  - $r[1..i] = r[1] | r[1+i] | \ldots | r[i]$
  - $r[1] = \text{true}[r][r]$
  - $r[0] = \text{false}[r][r]$
- 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
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) \Rightarrow (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 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) \Rightarrow (start_trans;data[1..8];end_trans)
  - Define: $b[1] = (b[1] | b[1+i] | b[i])$
  - Then have a nice representation
    always(req,ack) \Rightarrow (start_trans;data[1..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 $b[i..j] = \{b[i]\} \cup \{b[i+1]\} \cup \ldots \cup \{b[j]\}$
- Then always (req & ack) $\implies$ (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 ($@clk$) and formulas ($@$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: $b@clk \equiv ((clk[*]) & clk \& b)$

Model checking PSL (outline)

- SEREs checked by generating a finite automaton
  - recognize 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)
    - assert always (!en1 & en2)
- 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 $\Rightarrow$ threads enabled
  - enabled threads executed non-deterministically
  - execution of threads $\Rightarrow$ more events
- Combinational thread:
  - always @($v_1$ or $\ldots$ or $v_k$) $v_i$ = $E$
  - enabled by any change to $v_1, \ldots, v_k$
- Positive edge triggered sequential threads:
  - always @($posedge clk$) $v_i$ = $E$
  - enabled by $clk$ changing to $T$
- Negative edge triggered sequential threads:
  - always @($negedge clk$) $v_i$ = $E$
  - enabled by $clk$ changing to $F$
CHAPTER 3. APPENDIX: SLIDES

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

◮ Execution

◮ Simulation is non-deterministic

Combinational threads in series

◮ HDL-like specification:
  
  always @ (posedge clk) out <= f(l);
  always @ (posedge clk) out <= g(l);
  always @ (posedge clk) out <= h(l);

◮ Suppose in changes to v at simulation time t
  ◦ T1 will become enabled and assign f(v) to l
  ◦ if l's value changes then T2 will become enabled
  ◦ T2 will assign g(f(v)) to l
  ◦ if l's value changes then T3 will become enabled
  ◦ T3 will assign h(g(f(v))) to out

◮ Simulation quiesces (still simulation time t)

◮ Steps at same simulation time happen in \( \delta \)-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
  ◦ Left thread executed first:
    ◦ l gets input value at in

Sequential threads – trace semantics

◮ Trace semantics:
  
  \( (\forall t) \land (l(t+1) = (\text{Rise clk} t \rightarrow l(t)) \land (\forall t) \land (\text{out}(t+1) = (\text{Rise clk} t \rightarrow \text{out}(t))) \)

◮ 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:
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 / gets value input at i

Now no race
- Event semantics matches trace semantics
- What about combinational threads?