Approaches to verification of (concurrent) systems

Dijkstra noted: *Program testing can best show the presence of errors but never their absence*.

In concurrent systems, the nondeterminism arising from concurrency means that testing does not even show the absence of errors for the given test inputs.

This has lead to the search for more trustworthy methods of verification

Proof based systems:

- *Manual construction of proofs* — e.g. the proofs using Hoare Logic and Proof Outline Logic..
  - Pro: this is simply a formalization of the informal reasoning process that any human must go through in order to come up with a correct algorithm.
  - Pro: proofs are easily read and thus reviewed by other humans.
  - Con: can be difficult to come up with as the complexity of the algorithm increases.
  - Con: lack of education limits the number of available

reviewers

- *Automated theorem proving.*
  - AI programs to do this have had limited success.
    - Often they need considerable hints.
    - E.g. “first prove this lemma”, “use that lemma to prove this theorem”
    - Failure of the system to find a proof often yields little information about
      - Whether there is a bug or whether the AI simply failed to find a proof.
      - If there is a bug, its nature
  - Example systems: The Boyer-Moore prover. PVS. Isabelle. HOL. Eves.
- *Semiautomated methods*
  - Human comes up with a proof or a proof outline
  - Machine coompletes and reviews the proof
  - Example systems: Spec#, ESC-java.

State space exploration:

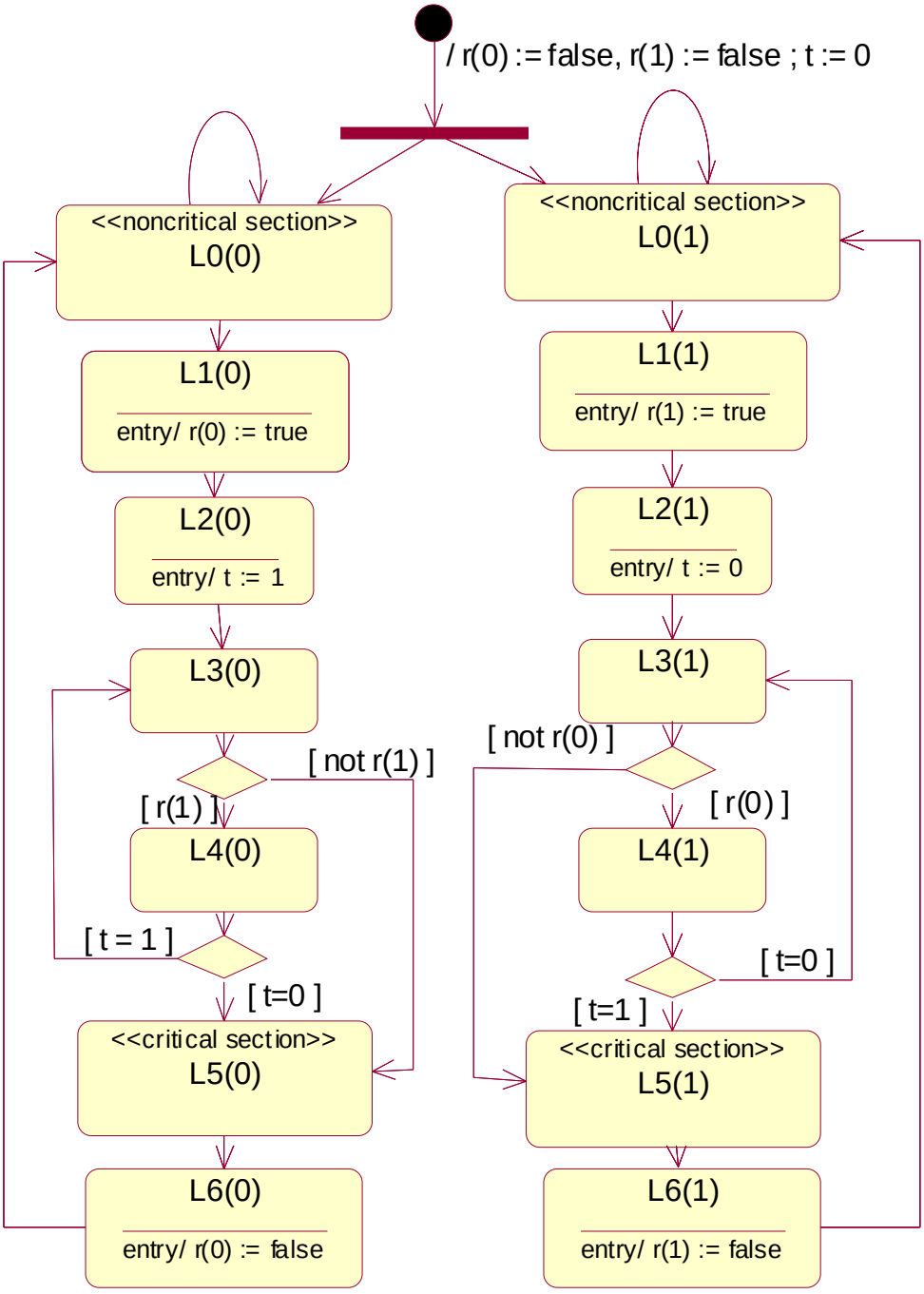
- *Manual enumeration and checking of states*
  - Con: time-consuming to do and review.
  - Con: only works in finite state cases.
- *Automated search of state space*
  - Has proved very effective for finite state systems: Hardware, communication protocols, simple software systems.
  - Compact representations of sets of states has lead to searches of very large and reasonably complex state spaces.
  - Example systems: SPIN, Murphi, TLC, SMV.

Since we have already looked at manual proof construction we will look state-space exploration methods

State space exploration

We model our algorithm as a (nondeterministic, concurrent) finite state automaton.

For example, Peterson’s algorithm.



Computing the flattened state machine

We can derive a nonconcurrent, nondeterministic state machine that is equivalent to the concurrent machine.

For the Peterson's algorithm example use the following encoding for state names

$$(pc(0), pc(1), r(0), r(1), t)$$

For example

$$(L0, L0, f, f, 0)$$

is the initial state.

There are $7 \times 7 \times 2 \times 2 \times 2 = 392$ states.

(However a local analysis shows that the value of $r(i)$ is determined by the control state of automaton i. Thus it is clear that at least 3/4 of these states are unreachable.)

Interleaving vs. true concurrency

In *interleaving semantics* we assume that in each computation step exactly 1 concurrent automaton advances.

- Thus the successors of $(L0, L0, f, f, 0)$ are
  - $(L1, L0, t, f, 0)$,
  - $(L0, L1, f, t, 0)$, and
  - $(L0, L0, f, f, 0)$ itself.

In *true concurrency semantics*, we also consider simultaneous transitions

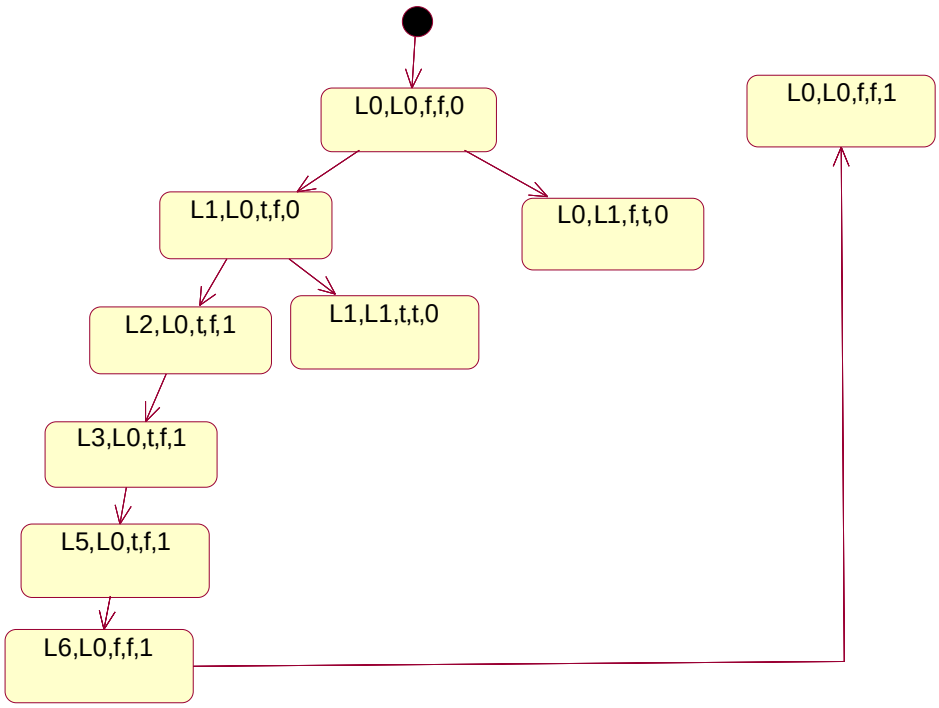
- So $(L1, L1, t, t, 0)$ is also a successor of $(L0, L0, f, f, 0)$

In true concurrency we have to decide how concurrent writes work. When $p0$ and $p1$ both write to t at the same time, what should be the result. One variant is to assume that one or the other write will succeed. Thus $(L1, L1, t, t, 0)$ and $(L1, L1, t, t, 1)$ both have successors

- $(L2, L2, t, t, 0)$ and $(L2, L2, t, t, 1)$ as well as
- $(L1, L2, t, t, 0)$ and $(L2, L1, t, t, 1)$

I'll assume interleaving from here on.

A portion of the resulting automaton:



A state is *reachable* if there exists a path from the initial state to it.

The complete automaton has at least 30 reachable states.

Proving properties:

Safety

The **safety property** that we need to show is **mutual exclusion**:

- *It is always the case that one process is not in* $L5$.

In terms of the flattened automaton this becomes:

- *No reachable state is of the form* $(L5, L5, ?, ?, ?)$.

This can easily be shown by examining the full flattened automaton.

Liveness

The **liveness property** we need to show is **nonstarvation**:

- *If a process ever reaches state L1 then eventually it will reach state L5.*

We might show this for P0 by showing that there are no cycles in the graph that both

- Include a state $(L1, ?, ?, ?, ?)$
- but not a state $(L5, ?, ?, ?, ?)$.

But this is not true. One such cycle is

$$(L1, L0, t, t, 0) \rightarrow (L1, L0, t, t, 0)$$

In fact the property is not true.

- We should make a **fairness assumption**:
  - ∗ *No process waits forever without making a transition.*

In terms of the flattened automaton we can observe

- All infinite paths that
  - ∗ start at the initial state, and
  - ∗ includes a state $L1, ?, ?, ?, ?$, but
  - ∗ not a subsequent state $L5, ?, ?, ?, ?$,
  - ∗ violate the fairness assumption

Temporal Logic

Originally temporal logic was invented to account for linguistic constructs such as:

- "Eventually my prince will come."
- "We'll always have Paris."

First we'll look at models in non-temporal propositional logic

Propositional models and validity

In propositional logic, a *model* is a set of primitive propositions.

A propositional formula is *valid* in the model if it is true, assuming exactly the primitive propositions in the model are true.

For example the sentence $p \Rightarrow q \wedge r$ is valid in models

$$\{p, q, r\}, \{q\}, \{r\}, \{q, r\}, \text{ and } \{\}$$

It is not valid in the models

$$\{p, q\}, \{p, r\}, \text{ and } \{p\}$$

A two-time temporal logic

Consider the statement

- *If it is nice today, it will be nice and cold tomorrow.*

This involves propositions

- *it is nice* — n
- *it is cold* — c

And times *today* and *tomorrow* .

We use the notation $\mathbf{X}$ to mean the next day.

From the point of view of today, we can write the sentence

$$n \Rightarrow \mathbf{X}(n \wedge c)$$

A model in our two time world is a sequence of two sets of propositions..

Using 0 for today and 1 for tomorrow, the following is a model

$$\langle \{n, c\}, \{n, c\} \rangle$$

Our example sentence $n \Rightarrow \mathbf{X}(n \wedge c)$ happens to be valid in this model.

Here is another model

$$\langle \{n, c\}, \{n\} \rangle$$

our example sentence is invalid in this model.

Infinite time

We can extend this idea to a one-way infinite, discrete model of time. I.e. times will be natural numbers, and models will be infinite sequences of sets of propositions.

We use

- $\mathbf{X}(s)$ to mean s is true in the next time period ($\mathbf{X}$ = neXt)
- $\mathbf{G}(s)$ to mean s is true now and at all future times ($\mathbf{G}$ = Global)
- $\mathbf{F}(s)$ to mean there exists a time now or in the future when s is true ($\mathbf{F}$ = Future)
- $s\mathbf{U}t$ to mean $\mathbf{F}(t)$ and s is true for all times from now (inclusive) until the next time t is true (exclusive). ($\mathbf{U}$ = Until)

$$\langle \{s\}, \{s\}, \{t\}, \{\}, \{t\}, \{s,t\} \rangle$$

This is LTL (Linear Temporal Logic).

Note (Demorgan)

$$\mathbf{F}(p) = \neg\mathbf{G}(\neg p)$$

Applying temporal logic

We can state safety properties, liveness properties, and fairness assumptions in temporal logic.

- **Safety properties.**
  - Invariants are easily stated in the form $\mathbf{G}(P)$
- **Liveness properties**
  - $\mathbf{G}(\mathbf{F}(Q))$ or $\mathbf{G}(\neg\mathbf{G}(\neg Q))$
    - Q will be true infinitely often. (It is never the case that Q will never be true again.)
  - $\mathbf{G}(P \Rightarrow \mathbf{F}(Q))$
    - Any time p is true, q will be true at the same time or at some future time
- **Fairness assumptions** are similar to liveness properties.
  - To show that property P is true under fairness assumption Q we show

$$Q \Rightarrow P$$

Peterson's algorithm again.

Mutual exclusion

The mutual exclusion property is an invariant:

$$\mathbf{G}(\neg in(L5_0) \vee \neg in(L5_1))$$

Nonstarvation

This is a liveness property.

$$\mathbf{G}(in(L1_0) \;\Rightarrow\; \mathbf{F}(in(L5_0)))$$
$$\mathbf{G}(in(L1_1) \;\Rightarrow\; \mathbf{F}(in(L5_1)))$$

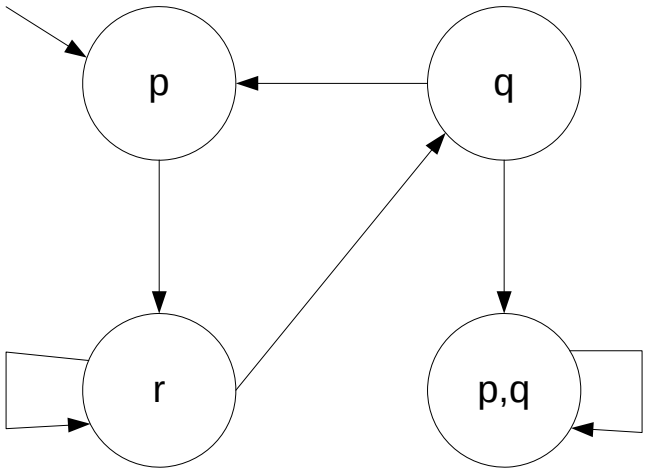
No infinite delay

This is a fairness assumption. Eventually, any thread not in state 0 will make a transition.

$$\begin{aligned}
&\mathbf{G}\,(in(L1_0) \Rightarrow \mathbf{F}(\neg in(L1_0))\\
&\wedge \mathbf{G}\,(in(L2_0) \Rightarrow \mathbf{F}(\neg in(L2_0))\\
&\wedge \cdots\\
&\mathbf{G}\,(in(L1_1) \Rightarrow \mathbf{F}(\neg in(L1_1))\\
&\wedge \mathbf{G}\,(in(L2_1) \Rightarrow \mathbf{F}(\neg in(L2_1))\\
&\wedge \cdots
\end{aligned}$$

Automated analysis using temporal logic

We can assign each state of a nondeterministic finite state automaton a set of propositions that are true in that state.



Then each path in the automaton gives rise to a model. E.g. (using — to mean infinite repetition).

$$\left\langle \{p\}, \{r\}, \overline{\{r\}} \right\rangle$$
$$\left\langle \{p\}, \{r\}, \{r\}, \{q\}, \{p,q\}, \overline{\{p,q\}} \right\rangle$$
$$\left\langle \{p\}, \{r\}, \{q\}, \overline{\{p\}, \{r\}, \{q\}} \right\rangle$$

We can check a temporal formula against each of these models. For example

$$P : \mathbf{G}(p \Rightarrow \mathbf{F}(q))$$

is valid in some models and not in others. But if we add a fairness assumption, say

$$Q : \mathbf{G}(r \Rightarrow \mathbf{F}(\neg r))$$

then we can show

$$Q \Rightarrow P$$

is valid in every model that this automaton describes.

Method: For each subformula, compute the set of states where it is true.

Luckily there are tools that automate this process.

Next we look at one such tool: SMV.

SMV

SMV developed at CMU and Berkeley by Ken McMillan

- Checks temporal logic formulas against models defined by automata
- Gives counter-examples when properties fail.
- Mostly used for clocked hardware but applicable to protocols and software.
- Exists commercial version (FormalCheck) from Cadence.
- Accepts Verliog syntax as well as its native syntax.
- Uses BDD (Binary Decision Diagrams) to efficiently represent large sets of states: 10^{20} or more!

The automaton is described using a *set* of (directed) equations that must be true in each clock cycle. E.g.

```
ack1 := req1;
ack2 := req2 & ~req1;
```

---

means

$$\mathbf{G}\left(\begin{array}{ll} & \mathsf{ack1} = \mathsf{req1} \\ \wedge & \mathsf{ack2} = (\mathsf{req2} \wedge \neg \mathsf{req1}) \end{array}\right)$$

E.g. a 2-way arbiter:

```
module main(req1,req2,ack1,ack2)
{
        input req1,req2 : boolean;
        output ack1,ack2 : boolean;

        ack1 := req1;
        ack2 := req2 & ~req1;
        ...
}
```

The states of this automaton are $(req1, req2, ack1, ack2)$. The states are

$$\{(F,F,F,F), (T,F,T,F), (T,T,T,F), (F,T,F,T)\}$$

All reachable states can transition to all others.

Properties are given as a set of assertions

```
mutex : assert G( ~(ack1 & ack2) );
serve : assert G( (req1 | req2) -> (ack1 | ack2) );
waste1 : assert G( ack1 -> req1) ;
waste2 : assert G( ack2 -> req2 );
```

SMV will verify each property or discover a model in which it is invalid.

States

Local variables can also declared and assignments can determine their initial and next states.

```
module main(req1,req2,ack1,ack2)
{
        input req1,req2 : boolean;
        output ack1,ack2 : boolean;

        prev : 1..2 ;

        if( prev=2 ) {
                ack1 := req1 ;
                ack2 := req2 & ~req1 ; }
        else {
                ack2 := req2 ;
                ack1 := req1 & ~req2 ; }
```

```
        init( prev ) := 1 ;
        if( ack1 ) next( prev ) := 1
        else if( ack2 ) next( prev ) := 2
        else next(prev) := prev
        ...
}
```

Aside: The last assignment could be written as

```
        if( ack1 ) next( prev ) := 1
        else if( ack2 ) next( prev ) := 2
```

End of Aside.

In terms of states, there is a transition, for example,

$$(T, F, T, F, 2) \rightarrow (T, T, F, T, 1)$$

We might prove the following timeliness requirements

```
fast1 : assert G( req1 -> ack1 | X(req1 -> ack1) ) ;
fast2 : assert G( req2 -> ack2 | X(req2 -> ack2) ) ;
```

Back to Peterson's

Data. We use the following types

```
typedef INDEX 0..1 ;
typedef PC {L0, L1, L2, L3, L4, L5, L6};
```

---

and variables

```
pc: array INDEX of PC; /* A pair of program counters */
r: array INDEX of boolean;
t: INDEX;
```

We also use two variables that are never assigned to.

Thus they may take on arbitrary values at each time point.

This models nondeterminism

```
act: INDEX ; /* The active process. */
arb : boolean ;
```

Initialization. We initialize the variables

```
init(t) := 0 ;
forall (i in INDEX) {
        init(r[i]) := 0;
        init(pc[i]) := L0 ; }
```

The next state. We make one transition in process 0 or process 1 arbitrarily. Recall act is arbitrarily either 0 or 1:

```
switch( pc[act] ) {
L0: { if( arb ) {
            next(pc[act]) := L0 ; }
       else {
            next(pc[act] ) := L1 ;
            next( r[act] ) := 1 ; } }
L1: { next(pc[act] ) := L2 ; next(t) := 1-act ; }
L2: { next(pc[act] ) := L3 ; }
L3: { if( ~r[1-act] ) next(pc[act]) := L5 ;
       else next(pc[act]) := L4 ; }
L4: { if( t=act ) next(pc[act]) := L5 ;
       else next(pc[act]) := L3 ; }
L5: { next(pc[act]) := L6 ; next(r[act]) := 0 ;}
L6: { next(pc[act]) := L0 ; } }
```

Safety. Mutual exclusion is

```
mutex : assert G ( pc[0] ~= L5 | pc[1] ~= L5 ) ;
```

Liveness. Nonstarvation properties

```
forall (i in INDEX) {
      nonstarve[i] : assert G( pc[i]=L1 -> F( pc[i]=L5 ) ) ;
    }
```

Fairness. To prove nonstarvation we must make a fairness assumption.

We can say that act will never exclude one value forever:

```
forall (i in INDEX) { fair[i]: assert G F( act = i ); }
```

We must tell the checker not to check the fairness assumptions

---

```
forall (i in INDEX) { assume fair[i]; }
```

We must tell the checker to use these assumptions in proving liveness

---

```
forall (i in INDEX) {
        using fair[i], fair[1-i] prove nonstarve[i] ; }
```
