Logic-Based Models & Tools

Temporal Logic

- Temporal logic has a certain appeal for real-time (and concurrency).
- The standard "temporal logics" might more properly be called "sequence logics", since they are based on sequences of events or discrete time steps.
- *Metric* temporal logic treats time as a continuous-valued quantity.

Temporal Logic Operators

- $\Box$ means "henceforth":
  If P is a predicate on state, then
  $\Box$P means that P will be true from now on.
- $\Diamond$ means "eventually":
  $\Diamond$P means that P will be true in *some* "future" state (starting with now).

Now

- In the absence of temporal operators, a predicate is understood to be true "now".
- Since "now" is arbitrary, this is like putting a $\Box$ in front of the predicate (similar to the way in which unquantified variables are implicitly $\forall$-quantified).

TL Examples

- ($\Box$ switch on) $\Rightarrow$ ($\Diamond$ motor running)
- motor running forward $\Rightarrow$ ($\Diamond$switch off)
- $\Box$ $\neg$(p in critical section $\wedge$ q in critical section)
- p at entrance to critical section $\Rightarrow$
  $\Diamond$p in critical section
- p in critical section $\Rightarrow$ x = 1
- req = 1 $\Rightarrow$ $\Diamond$p is bus master

Safety & Liveness

- These are two complementary kinds of properties:
  - **Safety**: $\Box$ $\neg$B means that "nothing bad will happen"
  - **Liveness**: $\Diamond$G means that "something good will happen"

Connection between $\Box$ and $\Diamond$

- $\Box P \equiv \neg\Diamond\neg P$
- $\Diamond P \equiv \neg \Box \neg P$
- sort of like the relationships between $\forall$ and $\exists$ (deMorgan's law)

Idioms and Reductions

- $\Box\Diamond P \equiv$ P is true infinitely often
- $\Diamond\Box P \equiv$ P will eventually be true forever
- $\Diamond\Diamond P \equiv \Diamond P$
- $\Box\Box P \equiv \Box P$

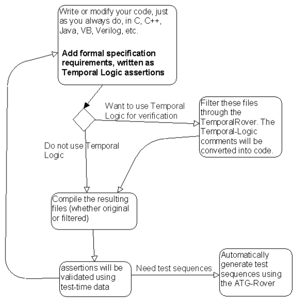
Other Operators

- $\bigcirc P \equiv$ P is true in the "next" state
- $\Box P \equiv P \wedge \bigcirc\Box P$
- $\Diamond P \equiv P \vee \bigcirc\Diamond P$
- $P\ U\ Q$ (P until Q) $\equiv (\Diamond Q) \wedge \neg Q \Rightarrow P$
- $P\ S\ Q$ (P since Q) $\equiv Q \Rightarrow \Box P$

Time Rover

- Time Rover, aka Temporal Rover is a commercial specification product based on temporal logic.
- The user formulates temporal logic assertions and Time Rover checks them in the context of live data.
- Time Rover will also generate test sequences for the system.

Time Rover Strategy



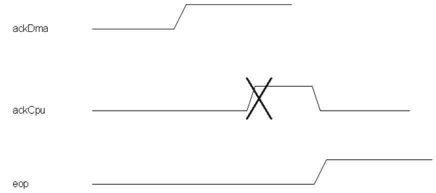
Examples from Time Rover

(http://www.time-rover.com/timing1.html)

"Always, if bus is granted to DMA, it should not be granted to CPU until DMA releases bus."

[ ] {ackDma} -> (! {ackCpu} U {eop}) ← old style

Always {ackDma} Implies (Not {ackCpu} Until {eop}) ← new style



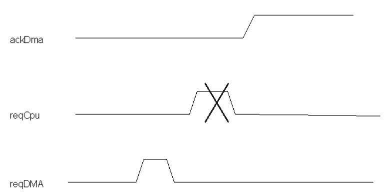
Examples from Time Rover

(http://www.time-rover.com/timing2.html)

"Always, when bus is granted to DMA, there should be an earlier time with a DMA bus request that was not followed by a CPU request."

[] {ackDma} -> (! {reqCpu} S {reqDMA})

Always {ackDma} Implies (Not {ackCpu} Since {reqDma})



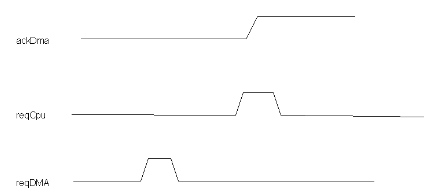
Examples from Time Rover

(http://www.time-rover.com/timing3.html)

"Always, when bus is granted to DMA, there should be an earlier time with a DMA bus request that was followed by a CPU request (cpu request can be before or after bus being granted to DMA)."

[] {ackDma} -> (<> {reqCpu} S {reqDMA})

Always {ackDma} Implies (Eventually {reqCpu} Since {reqDMA})



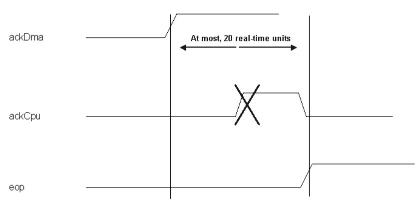
Real Time Constraint Spec.

(http://www.time-rover.com/timing1rt.html)

"Always, if bus is granted to DMA, it should not be granted to CPU until eop, and eop must occur within 20 time units."

[] {ackDma} -> (! {ackCpu} U < 20 {eop})

Always {ackDma} Implies (Not {ackCpu} Until<20 {eop})



Priority Inversion Spec., re Mars Pathfinder

(http://www.time-rover.com/Priority.html)

- Let
  - **HIGHPriorityTaskBlocked()** represent a situation where the information bus thread is blocked by the low priority meteorological data gathering task.
  - **HIGHPriorityTaskInMutex()** represent a situation where the information bus thread is in Mutex.
  - **LOWPriorityTaskInMutex()** represent a situation where the meteorological thread is in Mutex.
  - **MEDPriorityTaskRunning()** represent a situation where the communications task is running.
- Want
  - Not Eventually ({HIGHPriorityTaskBlocked()} And {MEDPriorityTaskRunning()})
  - Always({LOWPriorityTaskInMutex()} Implies Not {MEDPriorityTaskRunning()} Until {HIGHPriorityTaskInMutex()})

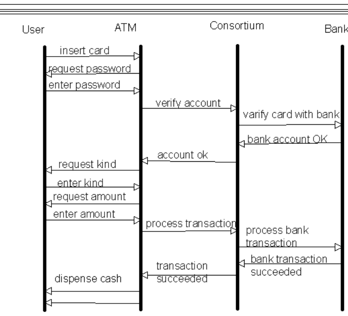
From Time Rover Web Page

(http://www.time-rover.com/Priority.html)

"Using such assertions (written as comments in the Pathfinder code), the Temporal Rover would generate code that announces success and/or failure of any assertion during testing.

Interestingly enough, the JPL engineers actually created a priority inversion situation during testing, but did not manage to *analyze* their recorded data well enough so to conclude that priority inversion is indeed a bug in their system. In other words, their test runs were sufficient, but their analysis tools were not."

ATM Example: Sequence Diagram



ATM Example: Time Rover Specs

Always ({insertCard} Implies
Eventually({ATMcashDispensed} Or {ATMErrorDisplay}))

Always ({ConsortiumSuccess} Implies (Not{insertCard}
Until {ATMcashDispensed}))
Always ({ConsortiumSuccess} Implies Eventually
{ATMcashDispensed})

Always ({ConsortiumSuccess} Implies Eventually C1<30
{ATMcashDispensed})

Other Generic Time Rover Specs

(http://www.time-rover.com/using.html)

- ev1 and ev2 happen or do not happen simultaneously: Always ({ ev1 } Iff { ev2 })
- if ev1 then ev2 two cycles later: Always ({ ev1 } Implies (2){ ev2 })
- ev2 not before ev1: Always (Not{ev2} Until {ev1})
- ev2 within n cycles after ev1: Always ({ev1} Implies Eventually <=n{ev2})
- ev2 within n1 and n2 cycles after ev1: Always ({ev1} Implies Eventually [n1,n2]{ev2})
- ev2 any number of cycles after ev1: Always ({ev1} Implies Eventually {ev2})
- ev2 after n cycles of no ev1: Always <=nNot{ev1} Implies Eventually >n{ev2}
- ev2 any number of cycles after ev1, with no ev3 in between:
  Always ({ev1} Implies ((Not{ev3} Until {ev2}) And Eventually {ev2}))

Other Generic Specs

- ev1 after the last ev2 and before ev3:
  Always (Last{ev2} Implies Eventually ({ev1} And AlwaysInThePast Not{ev3}))
- if ev1, then ev2 must not occur for n cycles:
  Always ({ev1} Implies Always <=n Not{ev2})
- if ev2, then ev1 must have occurred 3 cycles earlier:
  ({ev2} Implies Previous Previous Previous {ev1})
- If ev2, then ev1 must have occurred sometime earlier (not including present time): Always ({ev2} Implies Previous SometimeInThePast {ev1})
- evMid occured at least once between evStart and evStop:
  Always ({ evStart } Implies {evMid}Before{evStop})
- val==VAL between evStart and evStop:
  Always ({evStart } Implies {val==VAL} Until {evStop}) ,
  or Always ({val==VAL} Between ({evStart} , {evStop})

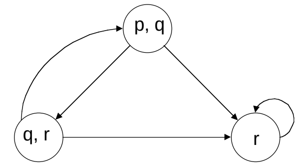
CTL (Computation Tree Logic)

- CTL, a form of temporal logic, is the basis for a formal analysis implementation.
- See Michael Huth and Mark Ryan, Logic in Computer Science: Modelling and reasoning about systems, Cambridge University Press, 2000.
- http://www.cs.bham.ac.uk/research/lics/

Models for CTL

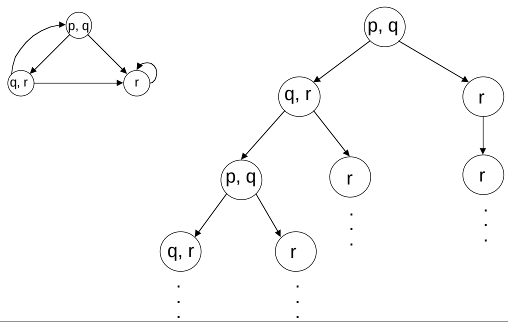
- The intention is that CTL formulas apply to a non-deterministic state-transition model, in which certain propositions evaluate to true or false in every state.

Example Model for CTL



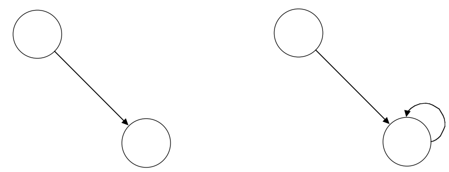
p, q, and r are atomic propositions,
shown in the states for which they are true.

Graphs can be "Unwound" into Trees



Technical Requirement

- Every state has at least one transition out of it (although that transition may immediately return).



not allowed   use instead

CTL Operators

- CTL expresses formulas about the model.
- A formula can make assertions about a single state or about paths from a state.
- The usual propositional operators are included in formulas, as are temporal operators described next.

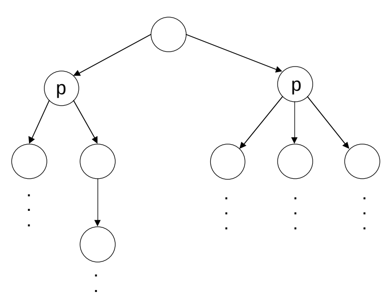
CTL Temporal Operators

- Relative to a given state and the paths leaving it:
  - **AX p**: All Next states satisfy p.
  - **AG p**: Along all paths, all states satisfy p.
  - **AF p**: Along all paths, some state satisfies p.
  - **A[p U q]**: Along all paths, p holds until q.
  - **EX p**: Some Next state satisfies p.
  - **EG p**: Along some path, all states satisfy p.
  - **EF p**: Along some path, some state satisfies p.
  - **E[p U q]**: Along some path, p holds until q.
- [A = All, E = Exists, X = neXt, G = Global, F = Future, U = Until]
- Note that "future" includes the present.

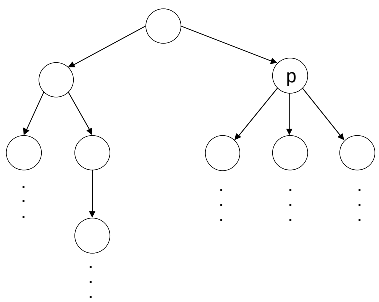
Comparison with Classical Temporal Operators

- **AX p** is like $\bigcirc$p
- **AG p** is like $\Box$p
- **AF p** is like $\Diamond$p
- **A[p U q]** is like p *U* q

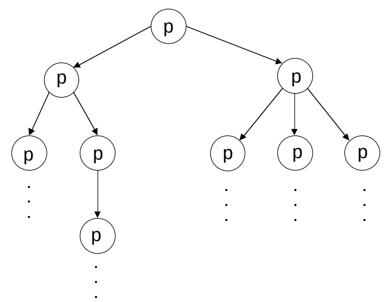
AX p (All neXt)



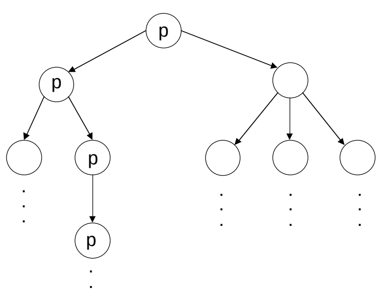
EX p (Some neXt)



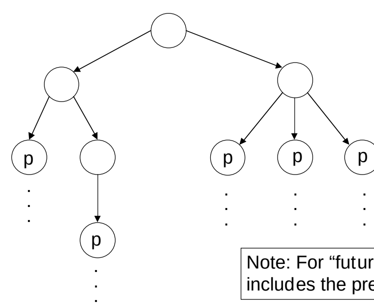
AG p (All Global)



EG p (Some Global)

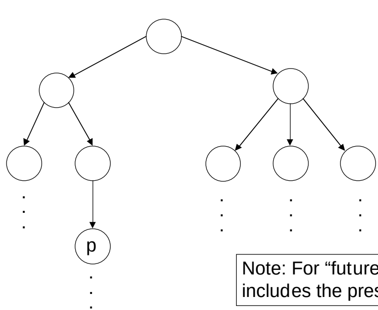


AF p (All Future)



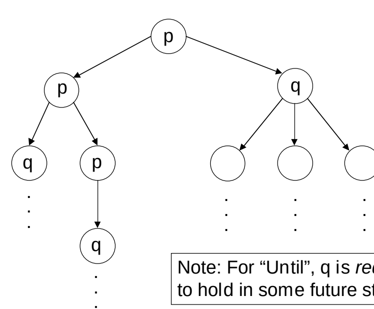
Note: For “future” includes the present.

EF p (Some Future)



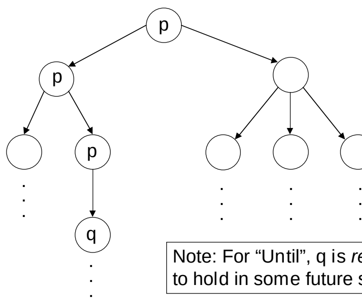
Note: For “future” includes the present.

A[p U q] (All … Until)



Note: For “Until”, q is *required* to hold in some future state.

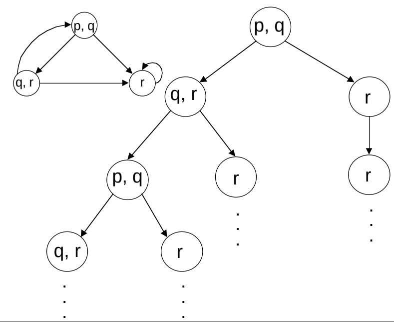
E[p U q] (Some … Until)



Note: For "Until", q is *required* to hold in some future state.

Which formulas hold for this model?

(a non-temporal formula holds iff it holds for the initial state)



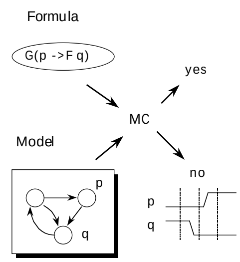
1. p ∧ q
2. ¬p ∨ r
3. EX p ∧ q
4. AX r
5. ¬EX q
6. EX ¬q
7. EF q ∧ r
8. AF r
9. E[q U r]
10. A[q U r]

Typical Formulas of Interest in Concurrent Systems

- AG (requested ⇒ AF granted)
- AG (AF enabled) [enabled infinitely-often]
- AF (AG deadlocked) [will always deadlock]
- A[motor-on U switch-off]

Model Checking (Clarke and Emerson)

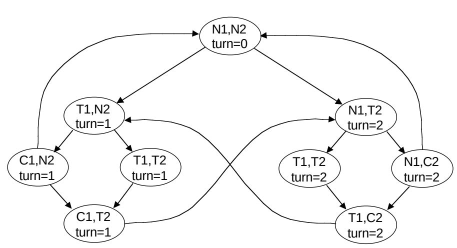
(slide from Ken McMillan, Cadence)



- output
  - yes
  - no + counterexample
- input:
  - temporal logic spec
  - finite-state model

A Mutual Exclusion Model

(Ken McMillan, Cadence)



N = noncritical, T = trying, C = critical

To show: $AG\ (T_1 \Rightarrow AF\ C_1)$

Labeling Algorithm

- The labeling algorithm is used for model-checking.
- Given a finite-state model and a formula, the algorithm will determine:
  - whether the formula is true for the model
  - if not, a state sequence that refutes the formula

Adequacy

- In order to simplify the algorithm, it is desirable to work with as *few* operators as possible.
- A set S of operators is *adequate* if the other operators can be expressed using only operators in S.

Adequate Operators

- Propositional: $\wedge$, $\neg$, $\bot$ ($\bot$ = "false")
- Temporal: AF, EU, EX
- Examples of translation:
  - $\top \equiv \neg\bot$
  - $p \vee q \equiv \neg(\neg p \wedge \neg q)$
  - $\neg AX\phi \equiv EX\neg\phi$
  - $EG\phi \equiv \neg AF\neg\phi$
  - $EF\phi \equiv E[\top \cup \phi]$
  - $A[p \cup q] \equiv \neg(E[\neg q \ U \ (\neg p \wedge \neg q)] \vee EG\neg q)$

Labeling Algorithm

- Input: A finite-state model, consisting of states and transitions, as well as a function
  L: states $\rightarrow$ proposition symbols
  and a CTL formula ϕ.
- Output: The set of states of the model for which the formula ϕ is true.

Labeling Algorithm

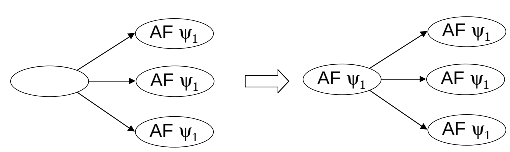
- First the formula ϕ is translated into an equivalent formula ϕ' that uses only operators from the adequate set mentioned.
- Then states are labeled, by building up from sub-formulas of ϕ'. When labeling is complete, the states labeled with ϕ' itself are those that satisfy ϕ.

Labeling Algorithm (1), by Sub-Formulas of ϕ'

- $\bot$: label no states with this formula.
- p (propositional variable): label with p those states s that have $p \in L(s)$.
- $\psi_1 \wedge \psi_2$: label with $\psi_1 \wedge \psi_2$ those states that are labeled with both ψ_1 and ψ_2.
- $\neg\psi_1$: label with $\neg\psi_1$ those states that are *not* labeled with ψ_1.
- EX ψ_1: label with EX ψ_1 those states having at least one successor labeled ψ_1.

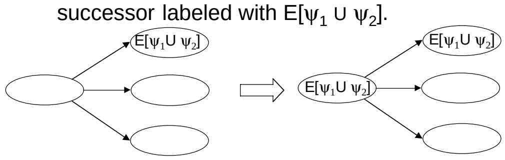
Labeling Algorithm (2), by Sub-Formulas of ϕ'

- AF ψ_1: label with AF ψ_1 (recursively):
  - all states labeled with ψ_1
  - all states the successors of which are labeled with AF ψ_1.

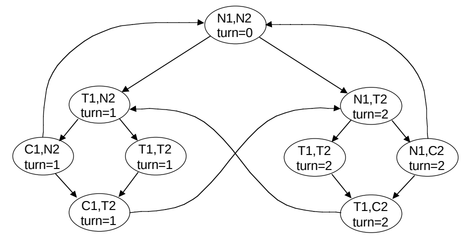


Labeling Algorithm (2), by Sub-Formulas of ϕ'

- $E[\psi_1 \cup \psi_2]$: label with $E[\psi_1 \cup \psi_2]$ (recursively):
  - all states labeled with ψ_2
  - any state labeled with ψ_2 and having *a* successor labeled with $E[\psi_1 \cup \psi_2]$.



Example: *Show: $AG(T_1 \Rightarrow AF\ C_1)$*

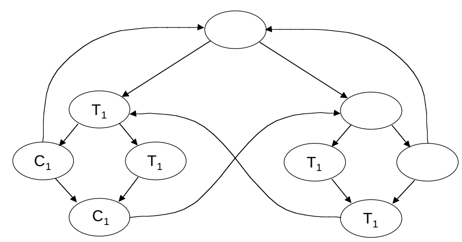


N = noncritical, T = trying, C = critical

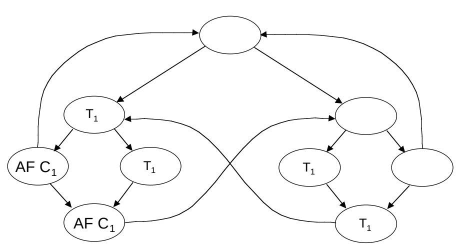
Transformation Steps

- *To be shown: $AG(T_1 \Rightarrow AF\ C_1)$*
- AG ψ is true for a state iff ψ is true for all states reachable from that state.
- For simplicity, we work with: $T_1 \Rightarrow AF\ C_1$
- Transform to: $\neg(T_1 \wedge \neg AF\ C_1)$
- We'll start labeling with: T_1 and C_1

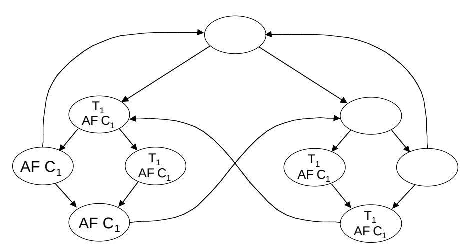
First-Level Labelings



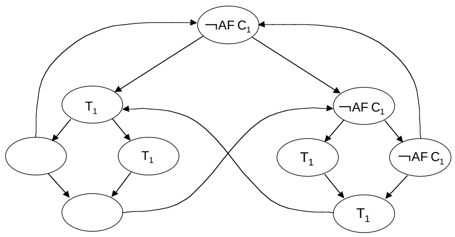
Second-Level Labeling-Basis



Second-Level Labeling-Recursion

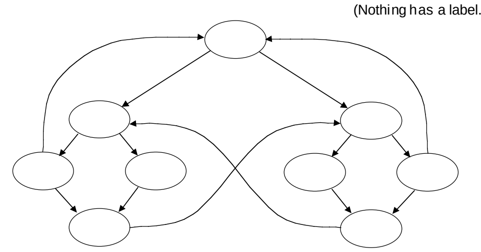


Third-Level Labelings



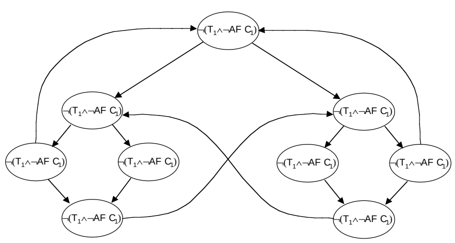
Fourth-Level Labeling ($T_1 \wedge \neg AF\ C_1$)

(Nothing has a label.)



Conclusion: $AG(T_1 \Rightarrow AF\ C_1)$

Fifth-Level Labeling ($\neg(T_1 \wedge \neg AF\ C_1)$)



(Everything has a label.)

Conclusion: $AG(T_1 \Rightarrow AF\ C_1)$

SMV (Symbolic Model Verifier)

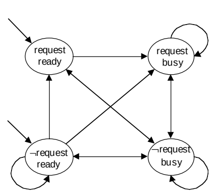
- SMV is a “modeling language” for systems.
- It resembles a programming language, but includes non-determinism.
- It includes a model checker for CTL formulas.
- Original was developed at CMU
- Cadence version is available for Windows on the web:
  http://www-cad.eecs.berkeley.edu/~kenmcmil/

SMV Example

SMV spec:

```
MODULE main
    VAR
        request : boolean;
        status : {ready,busy};
    ASSIGN
        init(status) := ready;
        next(status) :=
            case
                request : busy;
                1 : {ready,busy};
            esac;
    SPEC
        AG(request -> AF status = busy)
```

Finite-state model:



busy is shorthand for:
status = busy
and ready is for:
status = ready

SMV Spec for a Mutex Problem

```
MODULE main
    VAR
        pr1 : process prc(pr2.st, turn, 0);
        pr2 : process prc(pr1.st, turn, 1);
        turn : boolean;
    ASSIGN
        init(turn) := 0;
    --safety
    SPEC AG!((pr1.st = c) & (pr2.st = c))
    --liveness
    SPEC AG((pr1.st = t) -> AF (pr1.st = c))
    SPEC AG((pr2.st = t) -> AF (pr2.st = c))
    --no strict sequencing
    SPEC EF(pr1.st = c
            & E[pr1.st = c U (!pr1.st = c
            & E[! pr2.st = c U pr1.st = c ])])
```

```
MODULE prc(other-st, turn, myturn)
    VAR
        st : {n, t, c};
    ASSIGN
        init(st) := n;
        next(st) :=
            case
                (st = n)                                    :{t, n};
                (st = t) & (other-st = n)                   : c;
                (st = t) & (other-st = t) & (turn = myturn) : c;
                (st = c)                                    : {c, n};
                1                                           : st;
            esac;
        next(turn) :=
            case
                turn = myturn & st = c  : !turn;
                1                       : turn;
            esac;
    FAIRNESS running
    FAIRNESS !(st = c)
```

http://www.cs.bham.ac.uk/research/lics/ancillary/smv/index.html
also gives code for the alternating-bit protocol.

SMV on turing

- For examples, see: /cs/cs156/smv/doc/smv/examples
- To set up:

```
setenv SMV_DIR /cs/cs156/smv
setenv PATH $SMV_DIR/bin:$PATH
setenv MANPATH $SMV_DIR/man:$MANPATH
setenv LD_LIBRARY_PATH $SMV_DIR/lib:$LD_LIBRARY_PATH
```