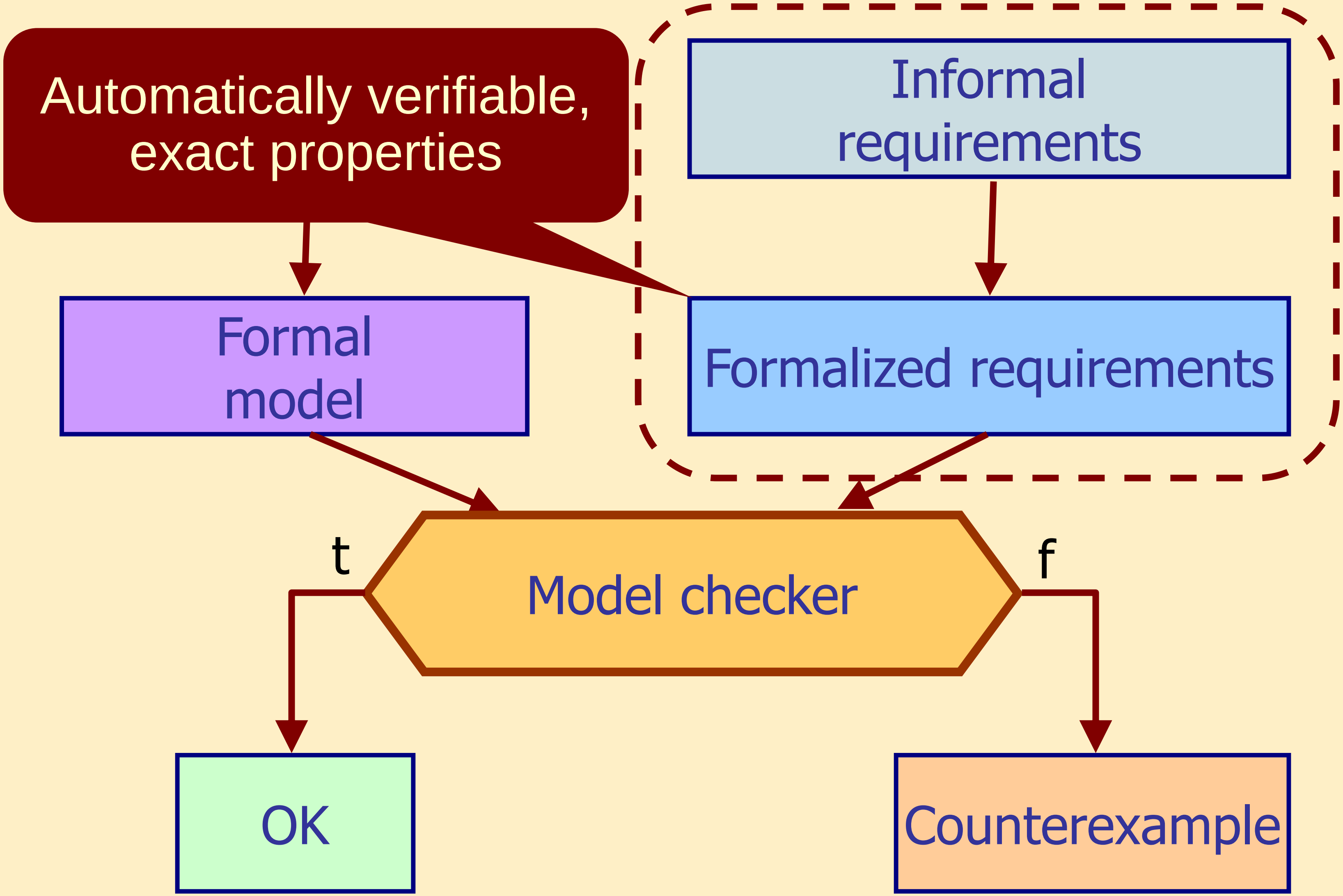
Formalizing requirements: Branching time temporal logics

dr. István Majzik
BME Department of Measurement and Information Systems

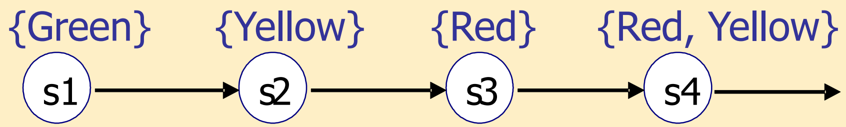
Our goal



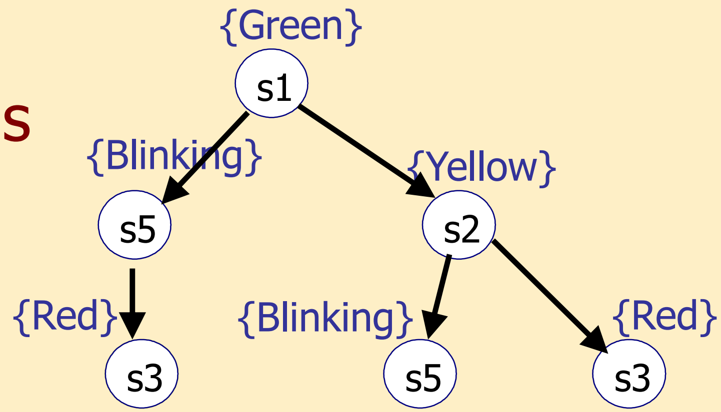
Automatically verifiable, exact properties
Informal requirements
Formal model
Formalized requirements
t
Model checker
f
OK
Counterexample

Classification of temporal logics

- Linear:
  - We consider individual executions of the system
  - Each state has exactly one subsequent state
  - Logical time along a linear timeline (trace)

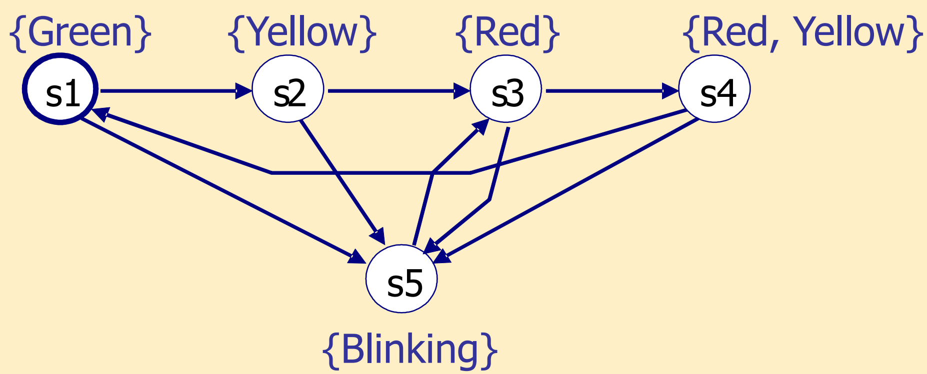


- Branching:
  - We consider trees of executions of the system
  - Each state possibly has many subsequent state
  - Logical time along a branching timeline (computation tree)

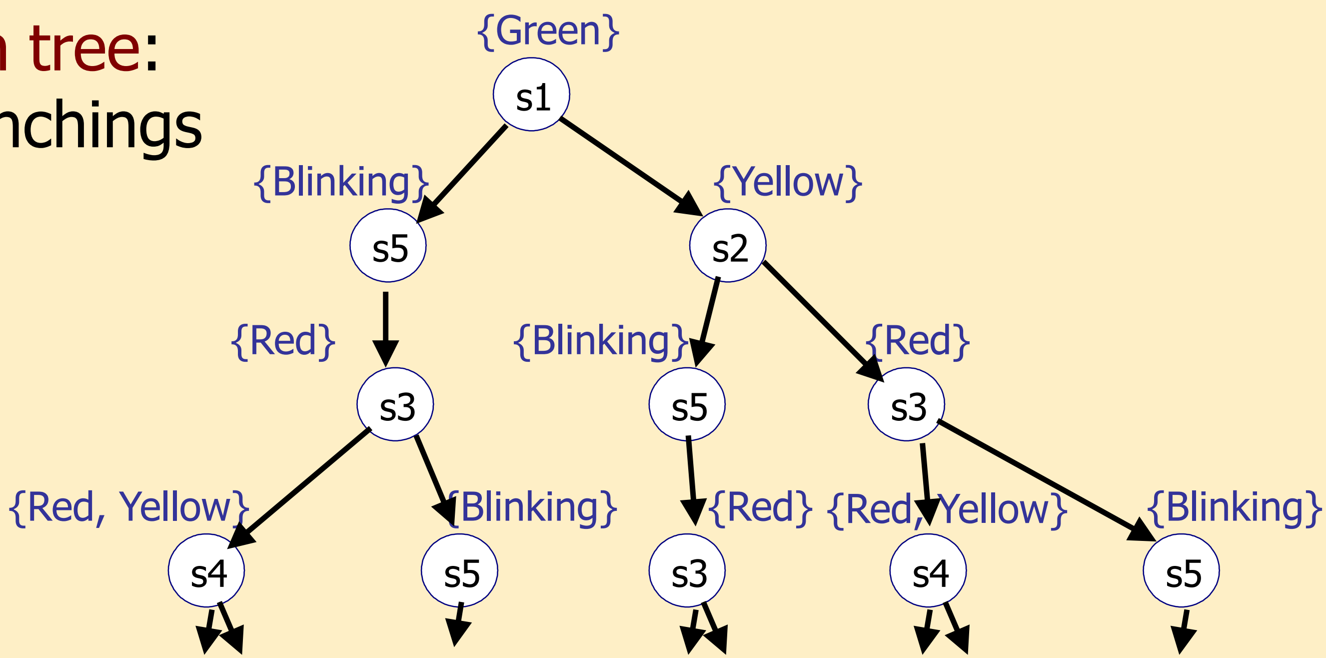


Computation tree

Kripke structure:



Computation tree: possible branchings

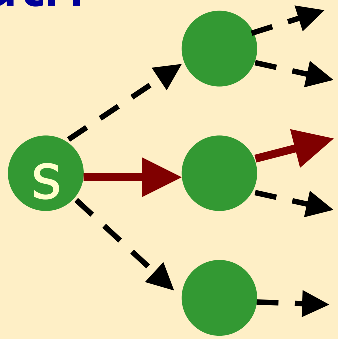


Branching time temporal logics: CTL, CTL*

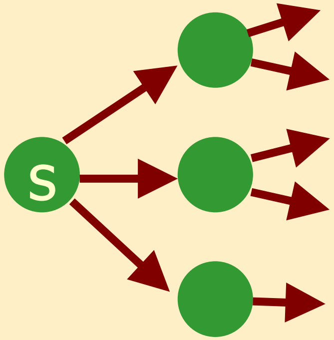
Branching

In a given state,
we can formulate requirements on the outgoing paths of the state:

- E p (Exists p): there exists at least one path from the state for which p holds
  - Requirement on a single path
  - Existential operator



- A p (for All p): for all paths from the state p holds
  - Requirement on all possible paths
  - Universal operator



Branching time temporal logics

- CTL*: Computational Tree Logic *

  An arbitrary combination of
  - Path quantifiers (E, A)
  - Path-specific temporal operators (X, F, G, U)

- CTL: Computational Tree Logic
  - An operator is a combination of a path quantifier and a path-specific operator
  - E.g. AX, E(_ U _)

CTL*: Computational Tree Logic *

CTL* operators (informal)

- Path quantifiers (interpreted over states):
  - A: “for All futures”,
    for all possible paths from the current state
  - E: “Exists future”, “for some future”,
    for at least one path from the current state
- Path-specific operators (interpreted over paths):
  - X p: “neXt”, for the next state p holds
  - F p: “Future”, for a state along the path p holds
  - G p: “Globally”, for each state of the path p holds
  - p U q: “p Until q”, for a state of the path q will hold, and until then for all states p holds

CTL* formulas

$$A(p \Rightarrow F\ q)$$

For all paths, we have that …

if initially p holds, …

then

in the future …

q eventually holds.

Examples for CTL* formulas

- $E(p \wedge G\, q)$

  There exists at least one path such that
  p holds (initially for the path) and for all suffices of the path q holds.

- $E(XXX\, p \vee F\, q)$

  There exists a path such that
  - p holds for its fourth state, or
  - eventually q holds

Formal treatment of CTL*

- So far: only an informal introduction
- To enable automatic formal verification, we need:
  - Syntax rules:
    What are the well-formed formulas in CTL*?
  - Semantic rules:
    When does a formula in CTL* hold for a given model?

CTL* syntax

- State formulas: evaluated over states
  - **S1**: an atomic proposition P is a state formula
  - **S2**: for state formulas p and q,
    we have state formulas $\neg p$ and $p \wedge q$
  - **S3**: for a path formula p,
    we have state formulas E p and A p

- Path formulas: evaluated over paths
  - **P1**: every state formula is a path formula
  - **P2**: for path formulas p and q,
    we have path formulas $\neg p$ and $p \wedge q$
  - **P3**: for path formulas p and q,
    we have path formulas X p and p U q

Well-formed formulas in CTL*: state formulas

CTL* semantics: notation

- $M = (S, I, R, L)$ Kripke structure
- $\pi = (s_0, s_1, s_2, \ldots)$ a path of M where $s_0 \in I$ and $\forall i \geq 0$: $(s_i, s_{i+1}) \in R$
  - $\pi^i = (s_i, s_{i+1}, s_{i+2}, \ldots)$ the suffix of π from i

- $M, \pi \models p$ (for a path formula p): in Kripke structure M, along path π, p holds

- $M, s \models p$ (for a state formula p): in Kripke structure M, in state s, p holds

CTL* semantics: state formulas

- **S1**:

  M,s |= P iff $P \in L(s)$

- **S2**:

  M,s |= $\neg$p iff not M,s |= p

  M,s |= $p \wedge q$ iff M,s |= p and M,s |= q

- **S3**:

  M,s |= E p (for path formula p)

  iff there exists a path $\pi=(s_0, s_1, s_2,\ldots)$ in M such that $s=s_0$ and M,π |= p.

  M,s |= A p (for a path formula p)

  iff for all paths $\pi=(s_0, s_1, s_2,\ldots)$ in M such that $s= s_0$ we have M,π |= p.

CTL* semantics: path formulas

- **P1**:

  $M,\pi \models p$ (for a state formula p) iff $M, s_0 \models p$

- **P2**:

  $M,\pi \models \neg p$ iff not $M,\pi \models p$

  $M,\pi \models p \wedge q$ iff $M,\pi \models p$ and $M,\pi \models q$

- **P3**:

  $M, \pi \models X\, p$ iff $M,\pi^1 \models p$

  $M, \pi \models p\, U\, q$ iff

  $\pi^j \models q$ for some $j \geq 0$ and
  $\pi^k \models p$ for all $0 \leq k < j$

Background: Computational complexity of evaluation

- Worst-case time complexity: at least $O(|S|^2 \times 2^{|p|})$
  - $|S|^2$ numer of transitions in the model (Kripke structure) in the worst case
  - $|p|$ number of temporal operators in the formula
- The exponential complexity seems frightening
  - Although temporal requirements tend to be short
- Goal: simplifying CTL*
  - Should remain usable in practice
  - Should reduce worst-case time complexity

CTL: Computational Tree Logic

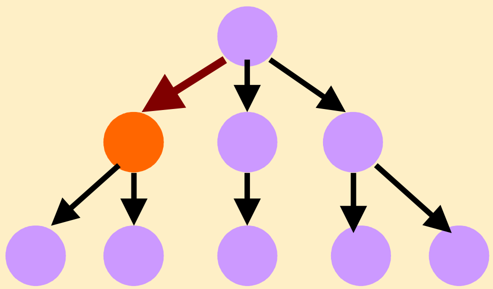
CTL operators (informal introduction)

Complex operators over sates:

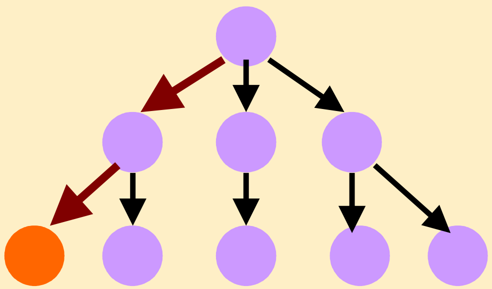
- EX p: there exists a path where p holds in the next state
- EF p: there exists a path where p holds in the future
- EG p: there exists a path where p holds globally
- E(p U q): there exists a path where p holds until q eventually holds

- AX p: for all paths p holds in the next state
- AF p: for all paths p holds in the future
- AG p: for all paths p holds globally
- A(p U q): for all paths p holds until q eventually holds

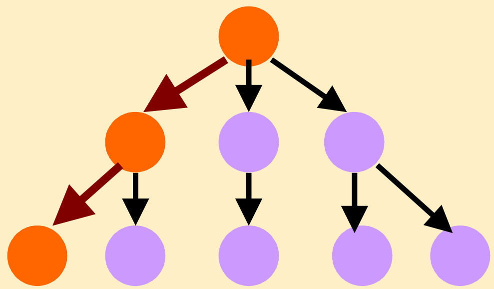
CTL opertators (examples)



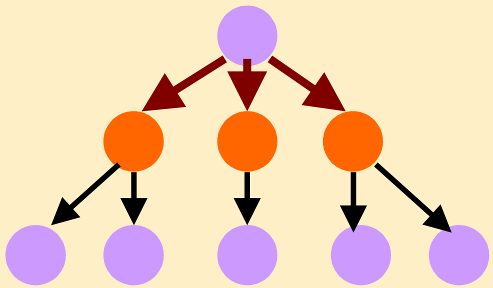
EX P



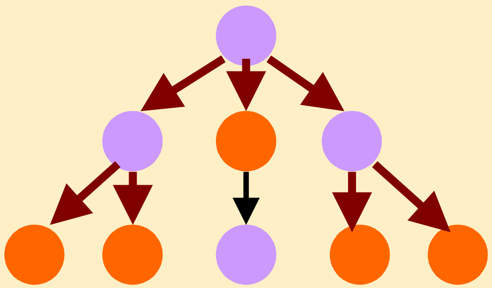
EF P



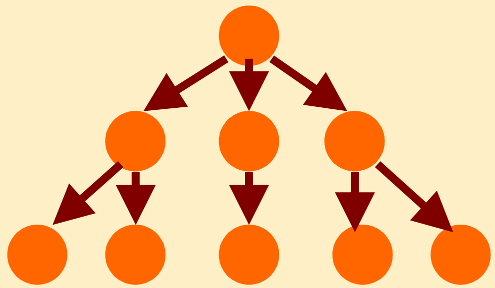
EG P



AX P



AF P



AG P

CTL formulas (examples)

- AG EF p

  starting from any state,
  a state can be reached where p holds

  - Example: AG EF Reset

- AG AF p

  starting from any state,
  we will encounter a state where p holds

  - Example: AG AF Terminated

- AG (p $\Rightarrow$ AF q)

  starting from any state,
  if we encounter a state where p holds,
  then we will eventually reach a state where q holds.

  - Example: AG (Request $\Rightarrow$ AF Reply)

CTL formulas (examples)

- EF AG p

  It is possible for the system to reach a state after which p will hold in all states

- AF AG p

  Along all paths we will eventually reach a state from which p will always hold

  - Example: AF AG Normal

- AG (p $\Rightarrow$ A (p U q))

  In all reachable states,
  if p holds in a state,
  then for all paths starting from that state,
  p holds until q eventually holds,

  - "p holds until q eventually holds": we will reach a state where q holds, and until then p holds in all states

Formalizing requirements: an example

- Two processes in a system: P1 and P2
- The state of processes w.r.t the requirements:
  - In critical section: C1, C2
  - Not in critical section: N1, N2
  - Waiting to enter critical section: W1, W2
- Atomic propositions:
  AP = {C1, C2, N1, N2, W1, W2}

Example (cont.)

- There is at most one process in the critical section:

  AG $(\neg(C1 \wedge C2))$

- If a process is waiting to enter the critical section, then it will eventually enter the critical section:

  AG $(W1 \Rightarrow AF(C1))$
  AG $(W2 \Rightarrow AF(C2))$

- Processes enter the critical section in alternating order; one exits, then the other enters:

  $AG(C1 \Rightarrow A(C1\ U\ (\neg C1 \wedge A((\neg C1)\ U\ C2))))$
  $AG(C2 \Rightarrow A(C2\ U\ (\neg C2 \wedge A((\neg C2)\ U\ C1))))$

P2 in critical section

P2 not in critical section

P1 enters the critical section

CTL syntax I.

State formulas:

- In CTL* we had:
  - **S1**: an atomic proposition P is a state formula
  - **S2**: for state formulas p and q,
    we have state formulas $\neg$p and p$\wedge$q
  - **S3**: for a path formula p,
    we have state formulas E p and A p

- In case of CTL, the same rules (**S1**, **S2**, **S3**) apply!

CTL syntax II.

Path formulas:

- In CTL* we had:
  - **P1**: every state formula is a path formula
  - **P2**: for path formulas p and q,
    we have path formulas $\neg p$ and $p \wedge q$
  - **P3**: for path formulas p and q,
    we have path formulas X p and p U q

- In case of CTL, we have a single rule instead:
  - **P0**: for <u>state formulas</u> p and q,
    we have path formulas X p and p U q

CTL syntax: Summary

State formulas:

- **S1**: an atomic proposition P is a state formula
- **S2**: for state formulas p and q,
  we have state formulas $\neg$p and p$\wedge$q
- **S3**: for a path formula p,
  we have state formulas E p and A p

Path formulas:

- **P0**: for state formulas p and q,
  we have path formulas X p and p U q

- Path formulas cannot be directly nested
- Path formulas are only used in rule **S3**
- Path formulas X p and p U q can only be nested under E and A

The consequences of formal syntax

- Path formulas cannot be directly nested
  - X and U can only be applied to state formulas
  - Boolean connectives can only be applied to state formulas
- Path formulas are only used in rule **S3**:
- Because of rule **S3**, only a path quantifier can be applied to path formulas X p and p U q
  hence operators "stick together"
  - EX, E(. U .),
  - AX, A(. U .)

Formulas in CTL and CTL*

- Derived operators of CTL
  - EF p means E (true U p)
  - AF p means A (true U p)
  - EG p means $\neg$AF ($\neg$p)
  - AG p means $\neg$EF ($\neg$p)

- CTL* but not CTL
  - E(X Red $\vee$ F Yellow)

    Boolean connective between path formulas
  - A(X G (Red $\wedge$ Yellow)),
    E(XXX Red)

    Nested path formulas

CTL formal semantics

- State formulas:
  - rules **S1**, **S2**, **S3** (see CTL*) remain unchanged
- Path formulas:
  - rules **P1**, **P2**, **P3** are replaced by a new rule **P0**:

P0:

- $M,\pi \models X\ p$ where p is a state formula iff
  $M,s_1 \models p$
- $M,\pi \models p\ U\ q$ where p,q are state formulas iff
  $M,s_j \models q$ for some $j \geq 0$ and
  $M,s_k \models p$ for all $0 \leq k < j$

Here we have state formulas according to syntax rule **P0**

Background: Computational complexity of evaluation

- Worst case time complexity: $O(|S|^2 \times |p|)$
  - $|S|^2$ numer of transitions in the model (Kripke structure) in the worst case
  - $|p|$ number of temporal operators in the formula
- Lower than in case of CTL*:
  - No $2^{|p|}$ factor
  - Expressive enough for many practical requirements
    - Safety requirements: AG
    - Liveness requirements: EF, AF
- What is the cost?

Expressive power

- A temporal logic is at least as expressive as an other temporal logic iff it is able to formalize all properties that the other logic can.
- It is more expressive iff furthermore there is a property that can be expressed in the logic but not in the other logic.
- Experience so far:
  - LTL can not consider branching (implicitly „for all paths”)
  - CTL is more restricted than CTL*, hence it is less expressive
  - CTL* also includes all properties expressible in LTL

Expressive power – Formally

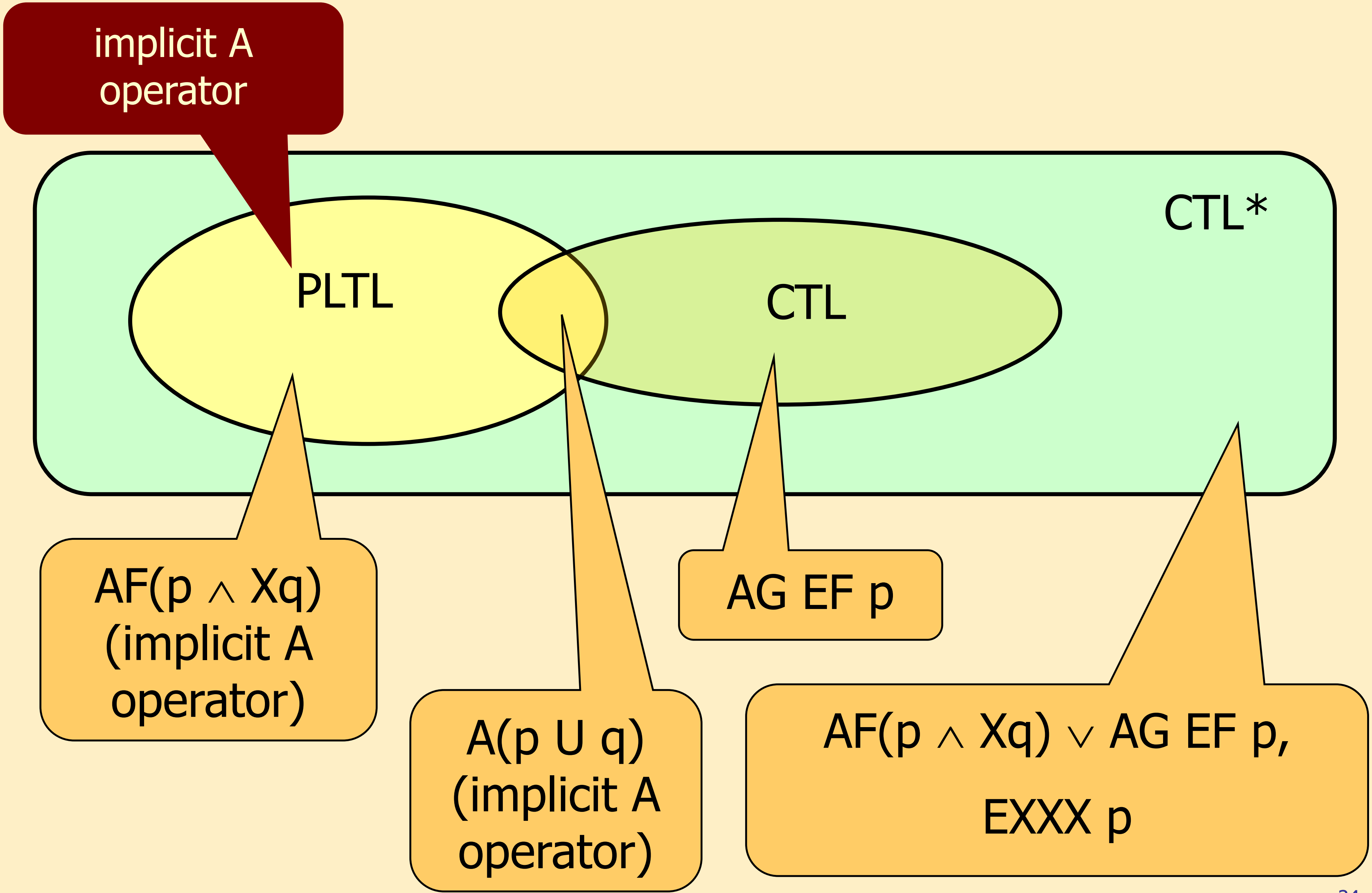
- The expressive power of TL2 is at least as big as the expressive power of TL1 iff
  for all Kripke structure M and for all its states s:

$$\forall p \in TL1:$$

$$\exists q \in TL2: \ (M, s \models p \iff M, s \models q)$$

- Iff this relation holds mutually then TL2 and TL1 have the same expressive power.

Expressive power of LTL, CTL, CTL*



Supplementary: Extensions

Stochastic logics:

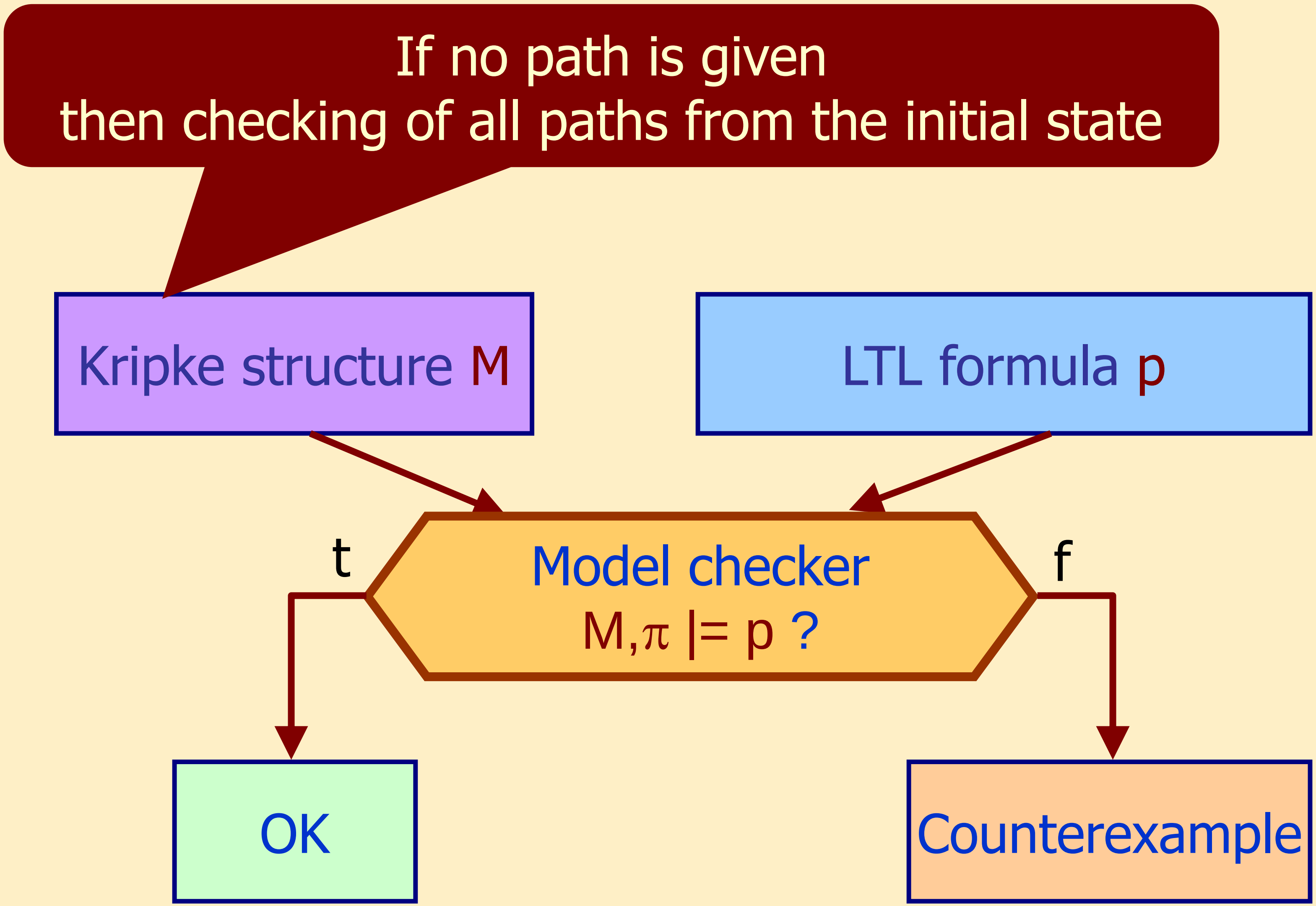
- Reliability and timing requirements:
  - E.g.: if the current state is ERROR then there is a probability less than 30% that this condition holds after 2 time units as well
- Extension of CTL:
  - Over Continuous-time Markov chains (not a Kripke structure)
  - Probability criteria for state reachability (steady state), path traversal
  - Timing criteria (time intervals) for operators X and U

Real-time logics:

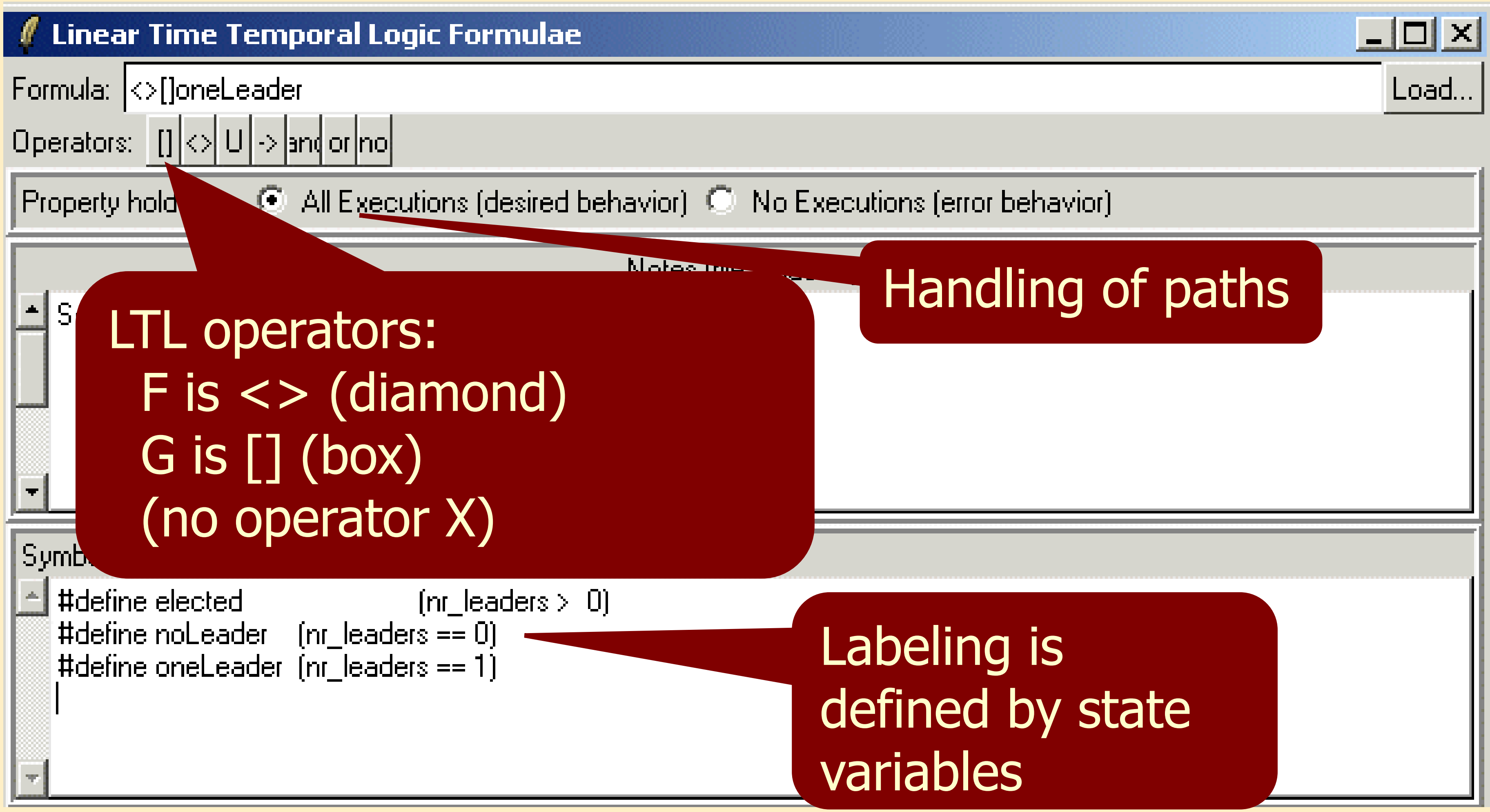
- Requirements of real-time systems
  - The logic can reference clock variables
  - Handling of time intervals

The model checking problem

LTL model checking

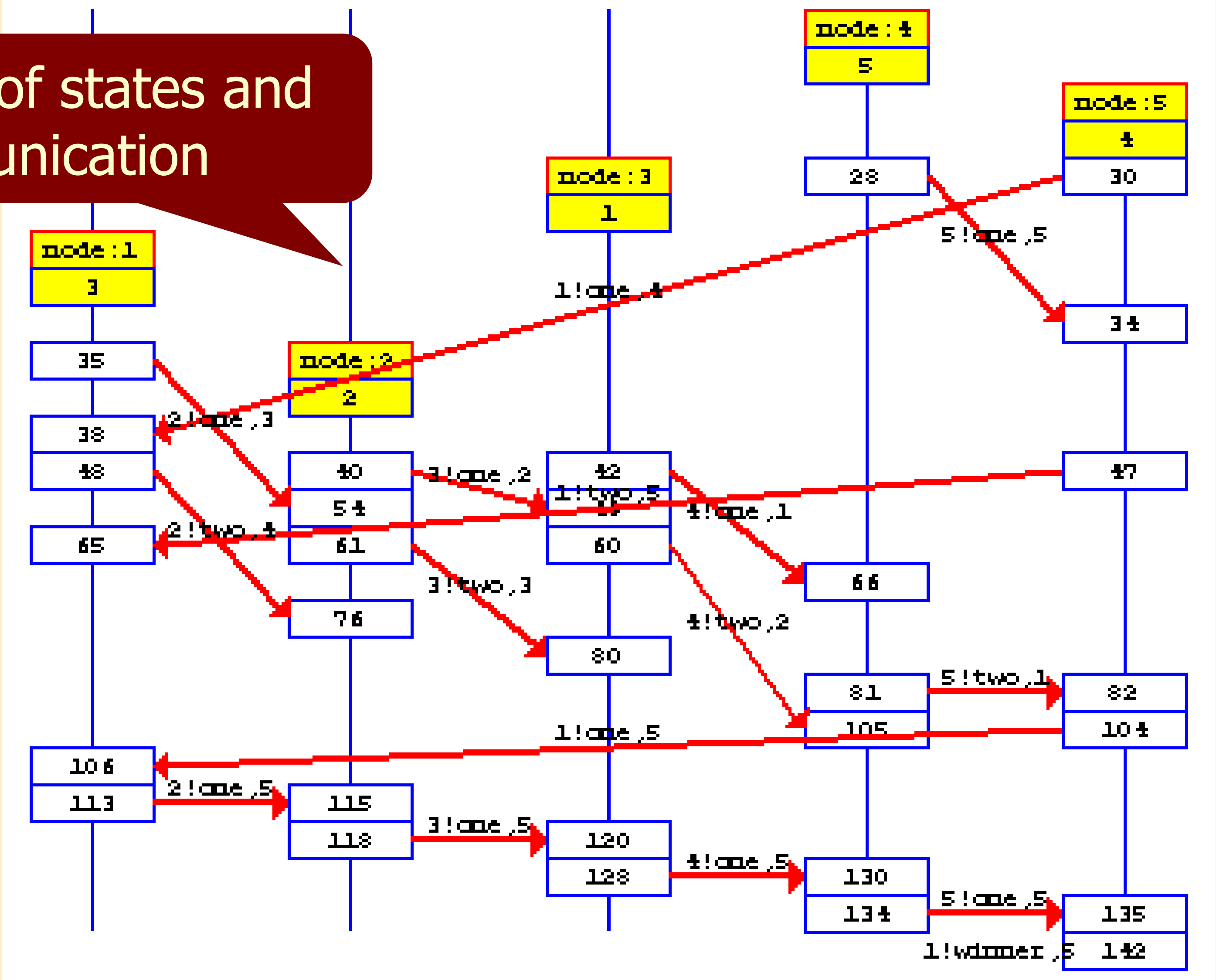


The model checker SPIN (old interface)

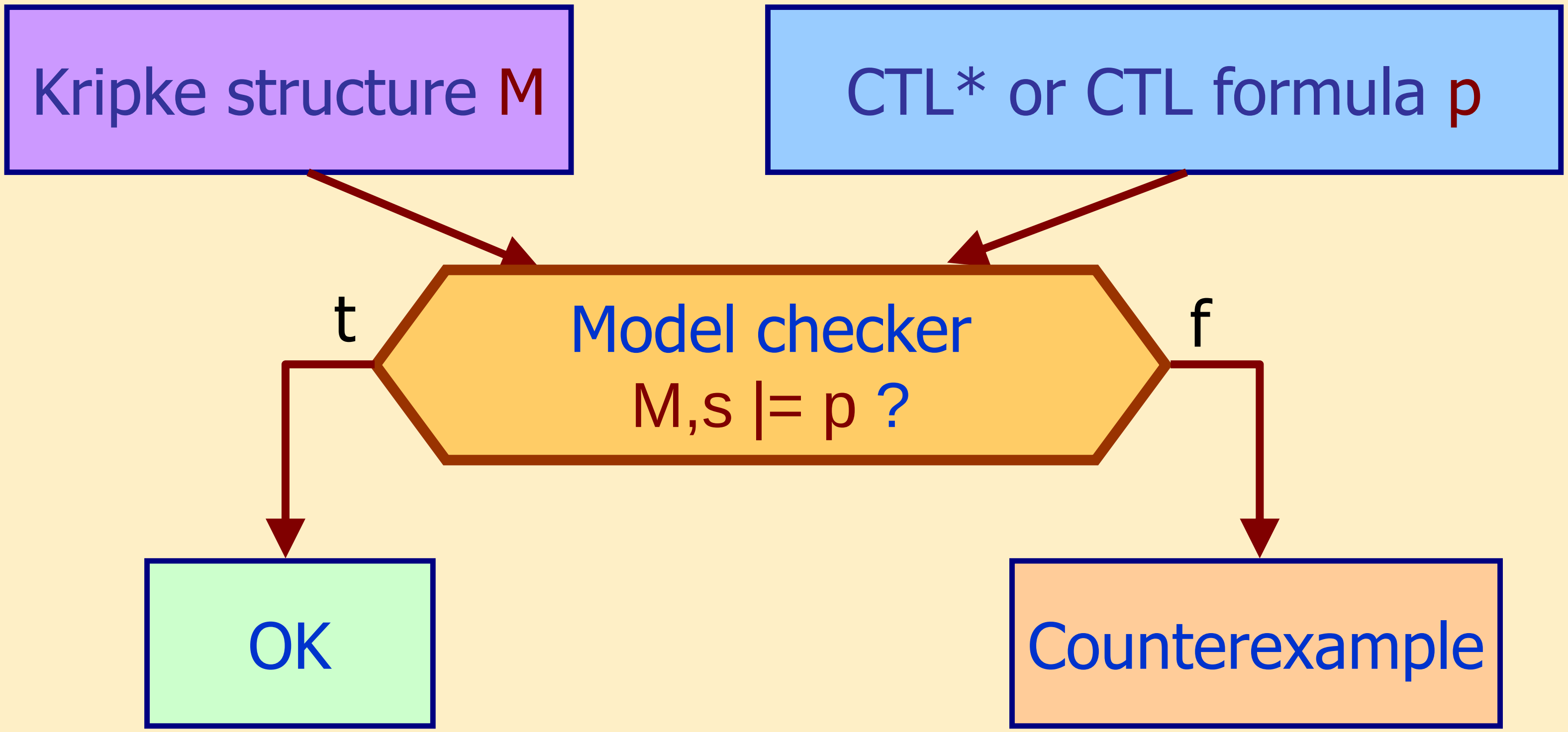


Counterexample in SPIN

Order of states and communication



CTL* or CTL model checking



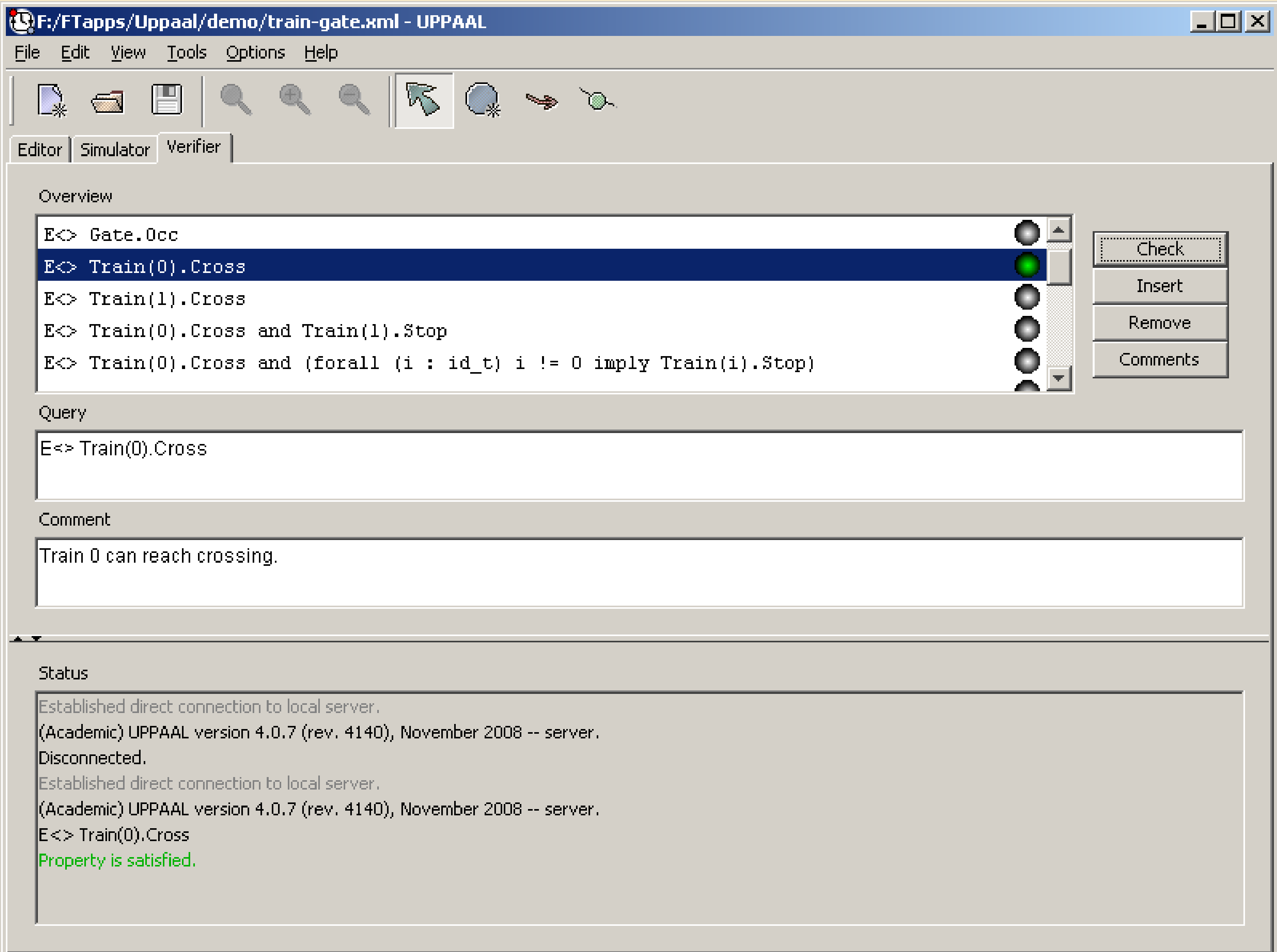
Model checking in UPPAAL

- Atomic propositions:
  - Predicates over state variables: a!=1
    - Terms: integer arithmetic, bitwise operators, ? : (if-then-else)
  - Reference for a location: Train(0).cross
    - For parameterized processes: forall, exists
  - Deadlock: deadlock expression (no action)
- Boolean connectives:
  - and, or, imply, not
- Temporal connectives: restricted CTL
  - Notation: [] (box) for G, <> (diamond) for F
    - Hence: A[], A<>, E[], E<>
    - E[] also for finite traces (to terminal state)
  - Temporal connectives can not be nested
    - One option though: p --> q for A[] (p imply A<> q)

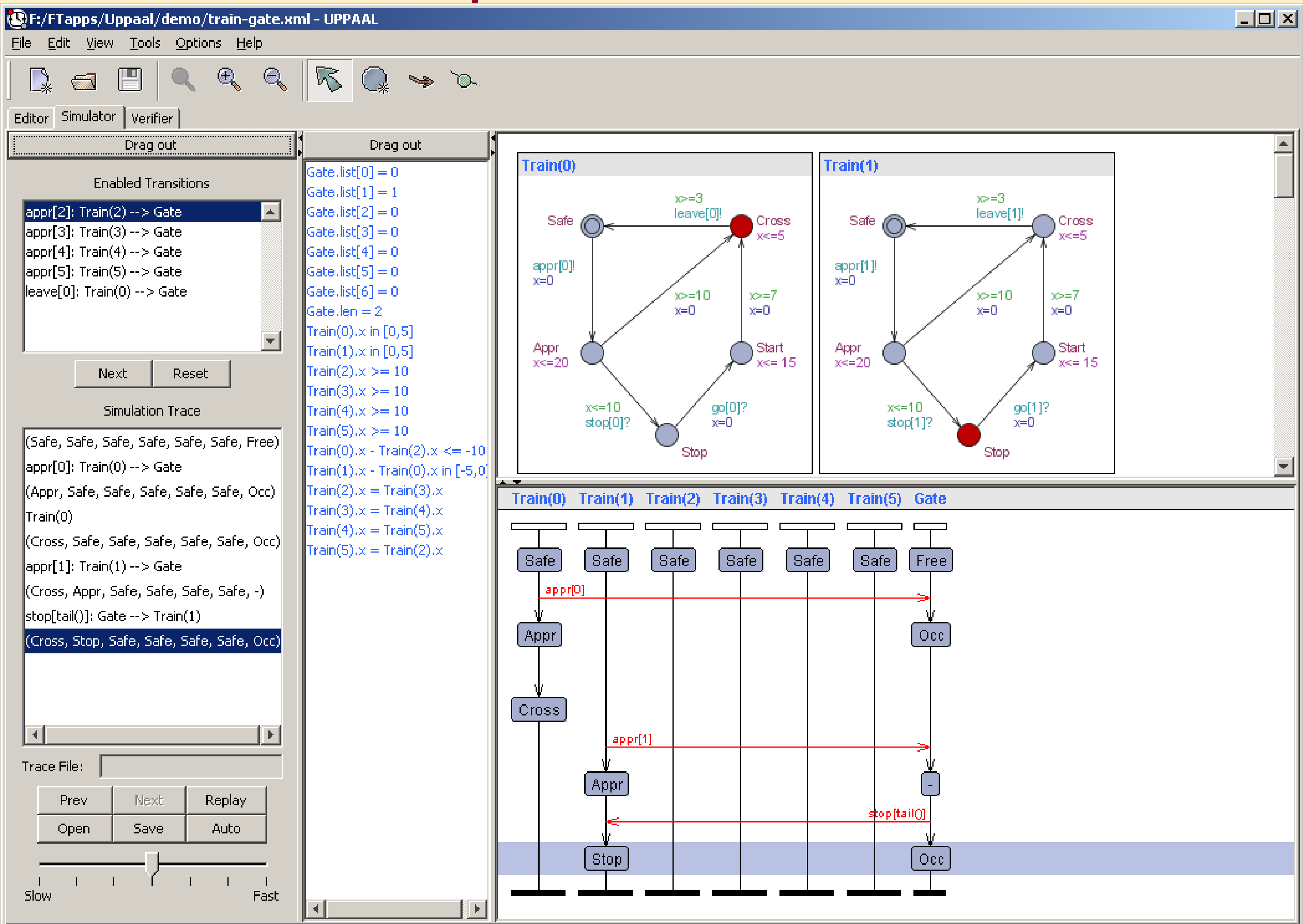
Checking requirements in UPPAAL

- Editable list of requirements
- Requirements can be checked one by one
- Counterexample can be generated:
  - Shortest, fastest, any
  - Can be replayed in simulator
- Traversal of the state space:
  - Depth-first search
  - Breadth-first search
- State representation:
  - Reduction
  - Approximate (under- or overapproximation)
  - The size of the hash table can be parameterized

The model checker interface in UPPAAL



Countereample in UPPAAL's simulator



Completing the motivating example

Motivating example: Mutual exclusion

- 2 processes, 3 shared variables (H. Hyman, 1966)
  - **blocked0**: process 1 (P0) wants to enter
  - **blocked1**: process 2 (P1) wants to enter
  - **turn**: which process is allowed to enter (0 for P0, 1 for P1)

P0

```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

P1

```
while (true) {
    blocked1 = true;
    while (turn!=1) {
        while (blocked0==true) {
            skip;
        }
        turn=1;
    }
    // Critical section
    blocked1 = false;
    // Do other things
}
```

Is the algorithm correct?

The model in UPPAAL (version 1)

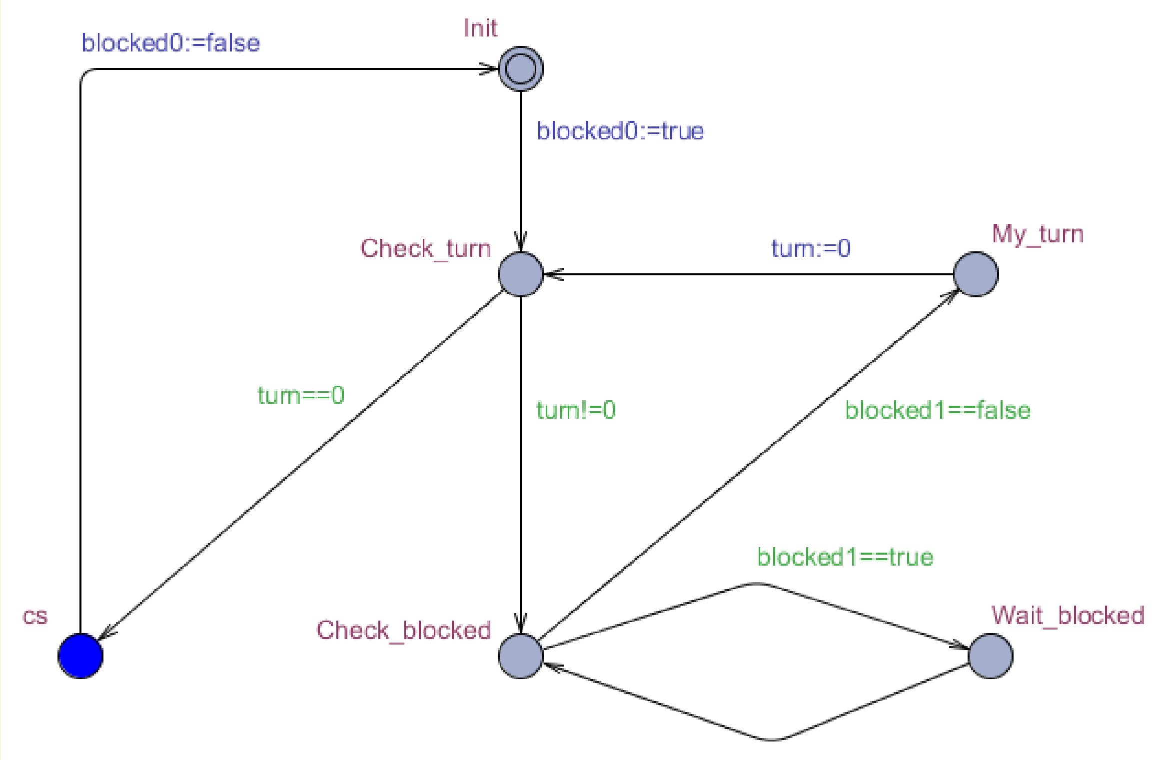
Declarations:

```
bool blocked0;
bool blocked1;
int[0,1] turn=0;
system P0, P1;
```

Modeling idioms used:

- Global variables
- Variables with restricted domain

Automaton P0:



P0

```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

UPPAAL: formalizing requiremetns

- Mutual exclusion:

  At most one process is allowed to be in the critical section

- Deadlock freedom:

  It is not possible that processes are mutually waiting for each other

- The expected behavior is possible:
  - For P0 it is possible to enter the critical section:
  - For P1 it is possible to enter the critical section:
- Starvation freedom:

  P0 will eventually enter the critical section:

  P1 will eventually enter the critical section:

Labels: P0.cs, P1.cs, deadlock

UPPAAL: formalizing requiremetns

- Mutual exclusion:

  At most one process is allowed to be in the critical section

  A[] not (P0.cs and P1.cs)

- Deadlock freedom:

  It is not possible that processes are mutually waiting for each other

  A[] not deadlock

- The expected behavior is possible:
  - For P0 it is possible to enter the critical section: E<>(P0.cs)
  - For P1 it is possible to enter the critical section: E<>(P1.cs)

- Starvation freedom:

  P0 will eventually enter the critical section: A<>(P0.cs)

  P1 will eventually enter the critical section: A<>(P1.cs)

Labels: P0.cs and P1.cs

UPPAAL: Results of model checking

- Mutual excusion is not ensured!
  - Counterexample: interleaving between the two processes (can be replayed in simulator)
- No deadlocks
- The expected behavior is possible
- Starvation freedom cannot be analyzed without specification of timing
  - Trivial counterexample: time elapses indefinitely in the initial location
    - A special consequence of timed behavior
    - Enforcing progress: urgent location or invariants
  - Starvation freedom?
    - The system is not starvation free (cyclic counterexample)

Fixing the algorithm

Peterson's algorithm

- For process P0 (P1 analogously):

Hyman:

```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

Peterson:

```
while (true) {
    blocked0 = true;
    turn=1;
    while (blocked1==true &&
            turn!=0) {
                skip;
    }

    // Critical section
    blocked0 = false;
    // Do other things
}
```