# Temporal Logic
## Theory and Applications

Reuven Yakar
Tel-Aviv University
06/12/2007

# Today

- First-order linear temporal logic
- Applications of temporal logic – verification of programs
  - The Producer-Consumer Scenario
  - Peterson's Algorithm
    - Using FOLTL.
    - Using Propositional LTL.
  - Verification using BrTL.

# First-order Linear Temporal Logic

- Extension of propositional temporal logic.
- We introduce variables, functions and predicates.
- Up side: greater expressiveness.
- Down side: More complicated deduction techniques, and no completeness.
- Main application: specification, design and verification of concurrent systems (examples soon).

# Introduction to the Syntax: Global Objects and Local Objects

- Global objects: The meaning/truth-value is the same in all states. (example: “Dan” (a person))
- Local objects: The meaning/truth-value is state-dependant (example: “The prime minister”)

# Syntax - Symbols

1. The set of *logical connectives* ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow$),
2. The set of *temporal operators* ($\bigcirc, \Box, \Diamond, \mathcal{U}$),
3. The set of *quantifiers* ($\forall, \exists$),
4. The *equality* connective (=),
5. A set P of *global predicate constants* (with *arity*),
6. A set F of *global function constants* (with *arity*),
7. A set Q of *local propositions,*
8. A set L of *local individual constants,*
9. A set V of *global variables*.

# Syntax – Construction Rules

- A *term*: a variable, a local individual constant, or, recursively, a functional form (defined next).
- A *functional form*: the application of a function constant *f* with arity n to $t_1,\ldots,t_n$ terms, denoted $f(t_1,\ldots,t_n)$.

  if n=0, we write f, instead of f(), and call it a (global) individual constant.
- A *predicate form*: the application of a predicate constant *p* with arity n to $t_1,\ldots,t_n$ terms, denoted $p(t_1,\ldots,t_n)$.

  if n=0, we write p, instead of p(), and call it a (global) propositional constant.

# Syntax – Construction Rules

- An *atom* is a predicate form, a local proposition or an equality (defined next).
- An *equality* is an expression (s = t), where s and t are terms.
- A *formula* is:
  - An atom,
  - ¬A, □A, ◇A, ○A, A∧B, A∨B, A→B, A↔B, A$\mathcal{U}$B where A and B are formulas, or
  - ∃xA and ∀xA, where A is a formula and x is a variable.

# Semantics – Temporal Interpretation

- A *temporal interpretation* is a 5-tuple $\mathcal{I} = (S, R, D, I_c, I_v)$
- (S,R) is a linear temporal frame (same as LTL),
- D is a non-empty set called the *interpretation domain,*
- $I_c$ is an *interpretation function for constants* as follows:
  - For a global function constant *f* of arity n, $I_c(f): D^n \rightarrow D$,
  - For a global predicate constant *p* of arity m, $I_c(p): D^m \rightarrow \{\mathbf{T}, \mathbf{F}\}$,
  - For a pair $(s, \ell)$ where $s \in S$ is a state and $\ell \in L$ is a local individual constant, $I_c(s, \ell) \in D$,
  - For a pair (s,q) where $s \in S$ is a state and $q \in Q$ is a local proposition, $I_c(s,q) \in \{\mathbf{T}, \mathbf{F}\}$.
- $I_v$ is an *interpretation function for variables*, which maps each variable $x \in V$ to an element $I_v(x) \in D$

# Semantics – Interpretations Rules

- We need interpretation rules in order to associate a truth value $\mathcal{I}(A)$ with each formula A, and an element $\mathcal{I}(t)$ of D with each term t.
- Most of them are very trivial, or simply the same as for propositional temporal logic.
- Here they are:
  - If x is a variable, then $\mathcal{I}(s,x) =_{def} I_v(x)$.
  - If $\ell$ is a local individual constant, $\mathcal{I}(s,\ell) =_{def} I_C(s,\ell)$.
  - If f is a function constant of arity n and $t_1,\ldots,t_n$ are terms, then $\mathcal{I}(s,f(t_1,\ldots,t_n)) =_{def} I_C(f)(\mathcal{I}(s,t_1),\ldots,\mathcal{I}(s,t_n))$.
    - We interpret each argument, and then apply the function on their values, exactly like a function in the code of a program. The difference is that the values of the arguments may depend on the current state, but the function itself is the same for all states.

# Semantics – Interpretations Rules

- If p is a function constant of arity m and $t_1,\ldots,t_m$ are terms, then $\mathcal{I}(s,p(t_1,\ldots,t_m))=_{def} I_C(p)(\mathcal{I}(s,t_1),\ldots,\mathcal{I}(s,t_m))$.
- If q is a local proposition, then $\mathcal{I}(s,q)=_{def} I_C(s,q)$.
- If $t_1$ and $t_2$ are terms, then $\mathcal{I}(s,t_1=t_2)$ is **T** iff $\mathcal{I}(s,t_1)= \mathcal{I}(s,t_2)$.
- If A and B are formulas, then ¬A, A∧B, A∨B, A→B, A↔B, ○A, □A, ◇A, A$\mathcal{U}$B are interpreted as in propositional temporal logic.
- If A is a formula, and if x is a variable, then $\mathcal{I}(s,\forall xA)$ is **T** if $\mathcal{I}_{x/d}(A)$ is **T** for every d∈D, and $\mathcal{I}(s,\exists xA)$ is **T** if $\mathcal{I}_{x/d}(A)$ is **T** for at least one d∈D.
  - $\mathcal{I}_{x/d}$ denotes $\mathcal{I}$, except that $I_V(x)=d$.

# Semantics – Remarks

- As in propositional temporal logic, we can assume that S=$\mathbb{N}$, and R is the successor function, R(n)=n+1.
- Note that the role of variables is the same as in classical predicate logic (they can free or bound, and their semantics is the same).
- The notation $\mathcal{I} \models_s A$ is used where $\mathcal{I}(s,A) = \mathbf{T}$.
- We say $\mathcal{I} \models A$ if $\mathcal{I} \models_s A$ in all states $s \in S$.
- We say that the formula A is *valid* if $\mathcal{I} \models A$ holds for each temporal interpretation $\mathcal{I}$.

# Axiomatic System

- The main idea is to combine:
  - The axiomatic system for classic predicate logic
  - The axioms and inference rules of equality
  - The axiomatic system for propositional temporal calculus (last week).
- A few changes due to the insertion of temporal aspects…

# Axiomatic System

- First, the schema $\forall x A(x) \rightarrow A(t)$, which is valid in classical logic, remains valid in temporal logic **iff** substituting the term t for the variable x doesn't introduce occurrences of local constants into the scope of a temporal operator. When this condition is satisfied, we say that "t is *substitutable* for x in A".
- Similarly, in order for the following to be valid, $t_i$ must be substitutable for $x_i$ in the appropriate formulas:

$$\frac{x_1=t_1 \wedge x_2=t_2 \wedge \ldots \wedge x_n=t_n}{p(x_1, x_2,\ldots, x_n) \leftrightarrow p(t_1, t_2,\ldots, t_n)} \qquad \frac{x_1=t_1 \wedge x_2=t_2 \wedge \ldots \wedge x_n=t_n}{f(x_1, x_2,\ldots, x_n) = f(t_1, t_2,\ldots, t_n)}$$

# Axiomatic System

- Furthermore, we should mention the relations between temporal operators and quantifiers:
  - $\forall x \bigcirc A \leftrightarrow \bigcirc \forall x A$
  - $\forall x \Box A \leftrightarrow \Box \forall x A$

# Axiomatic System

Axioms:

- Every valid schema of propositional temporal logic is an axiom. In addition:
- If A doesn't contain local objects, then $A \rightarrow \bigcirc A$.
- If x is a global variable, if A(x) is a formula and if the term t is subtitutable for x in A(x), then $\forall x A(x) \rightarrow A(t)$.
- If A is a formula and x is a global variable, then $\exists x \neg A \leftrightarrow \neg \forall x A$.

# Axiomatic System

- Inference rules:
  - Modus Ponens: If A and A→B are theorems, then B is a theorem.
  - Necessitation: If A is a theorem, then so is □A.
  - Generalization: If the global variable x doesn't occur free in A and if A→B is a theorem, then A→∀xB is a theorem.
- A few more axioms:
  - reflexivity axiom for equality.
  - substitutivity rules, restricted to substitutability of the terms, as explained before.

# Axiomatic System - Remarks

- As stated earlier, the temporal predicate calculus cannot be axiomatized completely, but this Axiomatic system is sufficient to deduce a lot of useful theorems, and for the applications we'll soon see.

# First-order Temporal Theories

- A *classical first-order theory* is achieved by adapting classical first-order logic to a specific domain. There are two components to these theories:
  - A *signature*: A set of function constants and predicate constants which will be interpreted. No other constant will occur in the formulas of the theory.
  - A set of *axioms*: which restrict the possible interpretations of elements of the signature. We only consider the interpretations for which the axioms are true.
- A similar concept exists for temporal logic, and called, surprisingly, a *temporal first-order theory*. The signature of a temporal first-order theory can include global constants, but also local individual constants, and local propositions. Examples ahead.

# Applications of Temporal Logic

- Temporal logic is useful to specify and verify concurrent systems. That is because it allows us to express statements like “something (e.g. program termination) will eventually happen”, and “something will happen at the next step”.
- We will try to determine whether a given system is correct with respect to its specification, stated as a temporal logic formula.

# Two Approaches

- The first, based on predicate temporal logic. It's more intuitive and gives a good insight into the programs under investigation.
- The second, based on propositional temporal logic, will allow the automation of the proof process (when it can be applied).
- But first, an elementary example to introduce the concepts needed to specify and verify concurrent systems:

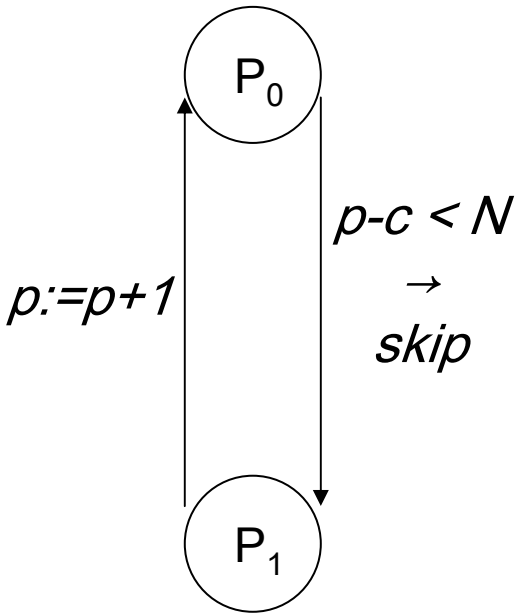
# The Producer-Consumer Problem

```
        P:
while   true do
        begin
        await p-c < N do produce;
        p := p + 1
        end
```

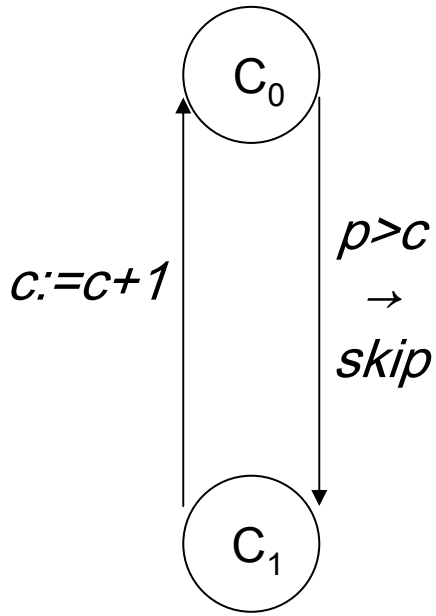
```
        C:
while   true do
        begin
        await p > c do consume;
        c := c + 1
        end
```

- Memory sharing
- Initially, p=c=0.
- "*produce;*" and "*consume*" don't alter the values of p,c.
- '*await B do S*' means "wait until condition B is true, and then execute S". If B = *F* forever, there's an infinite delay.

# Graphical Model




*Process P*




*Process C*

# Modeling

- A *computation state* is an instance of the 4-tuple ($L_P$,$L_C$,p,c), where $L_P$ is either $P_0$ or $P_1$, and similarly for $L_C$. p and c are the current values of the counters p and c.
- For example, if N=2, a possible computation is:

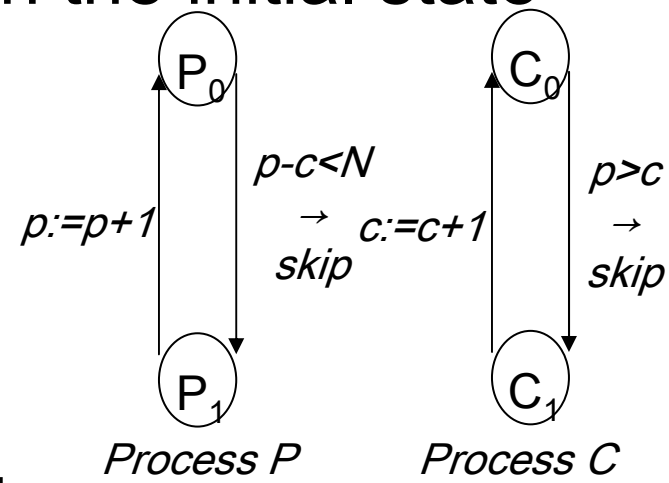
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $L_P$ | $P_0$ | $P_1$ | $P_0$ | $P_1$ | $P_1$ | $P_0$ | $P_0$ | $P_0$ | $P_0$ | $P_1$ | $P_0$ |
| $L_C$ | $C_0$ | $C_0$ | $C_0$ | $C_0$ | $C_1$ | $C_1$ | $C_0$ | $C_1$ | $C_0$ | $C_0$ | $C_0$ |
| p | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 3 |
| c | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 2 |

# Formulas State Specifications

- Soon we will see how to obtain axioms from the code of the program, that describe its computations. They will define the temporal theory of the program.
- For now, we can state specifications of the program as temporal formulas. if we prove that these formulas are theorems of the theory, we can verify the correctness of the program.
- For example:

# In Our Simple Case

- One property is that the processes are adequately synchronized, given the initial state is acceptable. Formally:
  - Init: (at-$P_0$ ∧ at-$C_0$ ∧ p=0 ∧ c=0) ,
  - Z: (0 ≤ p-c ≤ N) ,
  - (Init → □Z)
- And if we look at:
  - I: [at-$P_0$ →p-c≤N] ∧ [at-$P_1$→p-c<N] ∧ [at-$C_0$→p≥c] ∧ [at-$C_1$ → p>c].




It's rather clear that I→○I (we'll see how to prove this later). As a consequence, if I is true initially, it's always true, and also, (Init → □Z) is valid.

# Program Modeling

- A set of processes: $\{P_1,\ldots, P_n\}$ (like {P,C} we saw)
- A set (*memory*) of *program variables* (as opposed to *logical variables*): $X = \{x_1,\ldots, x_k\}$ $\{p,c,N\}$
- Each process is represented in a *flowchart*, which is a finite, connected and directed graph.
- Initial node for each process.
- Each node has a *label*. $(P_0, P_1, C_0, C_1)$
- Each arc is associated with a statement.
- A statement is an ordered pair (C,A) (denoted C→A). C is a *condition*. A is an *assignment*.

Example: $p\text{-}c < N \rightarrow skip$

# Program Modeling

- C is a Boolean term involving global constants and program variables.
- A has the form: $(x_1,\ldots, x_k):=(t_1,\ldots, t_k)$. We omit variables that we don't change, and short by "*skip*", if we change none.
- We use "*C?*" instead of "$C\rightarrow$*skip*", and A instead of "*true*$\rightarrow A$".
- Sometimes a lexical representation is more convenient, so the graph is simply represented as a list of transitions of the form $(\ell, C\rightarrow A, \ell')$.
- For each process P: $L_P$ is the set of its labels and $Tr_P$ is the set of its transitions.
  - $L_P=\{P_0,P_1\}$
  - $Tr_P=\{(P_0, p\text{-}c\prec N\rightarrow skip, P_1), (P_1, (true\rightarrow)p:=p+1, P_0)\}$

# Program Modeling

- The *output condition* of a label $\ell$ is *out($\ell$),* the OR on all the C's on transitions that originate from $\ell$ (false by default, if no such transition exists).
  - If the process is at the node with label $\ell$, and *out($\ell$)*=F, it means that the process is "stuck"; it cannot proceed.
- A *memory state* is a total function on the domain X, mapping each variable to its value. The set of memory states is denoted Σ.
- *control state* is an element of $\Gamma = L_{P1} x \ldots x L_{Pn}$.
- A *system state* is a pair s=(L,σ) where L is a control state and σ is a memory state. (slide 23)
- A state is *final* if for all i, $out(\ell_i)(\sigma)$=F. (all the processes are "stuck").

# Program Modeling

- A transition $T=(\ell, C \rightarrow A, \ell')$ of any process $P_i$ defines a transformation of the system state.
- This transformations is modeled by a (partial) function $T:\Gamma x\Sigma \rightarrow \Gamma x\Sigma$ defined as follows:
- If $s=(L,\sigma)$ and $t=(L',\sigma')$ are two system states, $t=T(s)$ holds iff the following three:
  - L and L' are the same, except for one $1 \leq i \leq n$, for which, $L(i)=\ell$, and $L'(i)=\ell'$. (only one process is in a different node after the transition)
  - $\sigma$ satisfies C (the condition of the transition).
  - $\sigma' = A(\sigma)$. (the memory is changed according to the assignment)

# Program Modeling

- A computation of the system is a sequence ($s_0$, $s_1$,…) of states.
- $s_0$ is the initial state, meaning its control state is the array of the initial states of the processes.
- If $s_{k+1}$ exists, then there exists a transition T such that $s_{k+1}=T(s_k)$.

# Break

# Example: Peterson's Algorithm

```
        P:
while true do
        begin
        non-critical section;
        inP := T;
        turn := q;
        await ¬inQ ∨ turn = p do skip;
        critical section;
        inP := F
        end
```

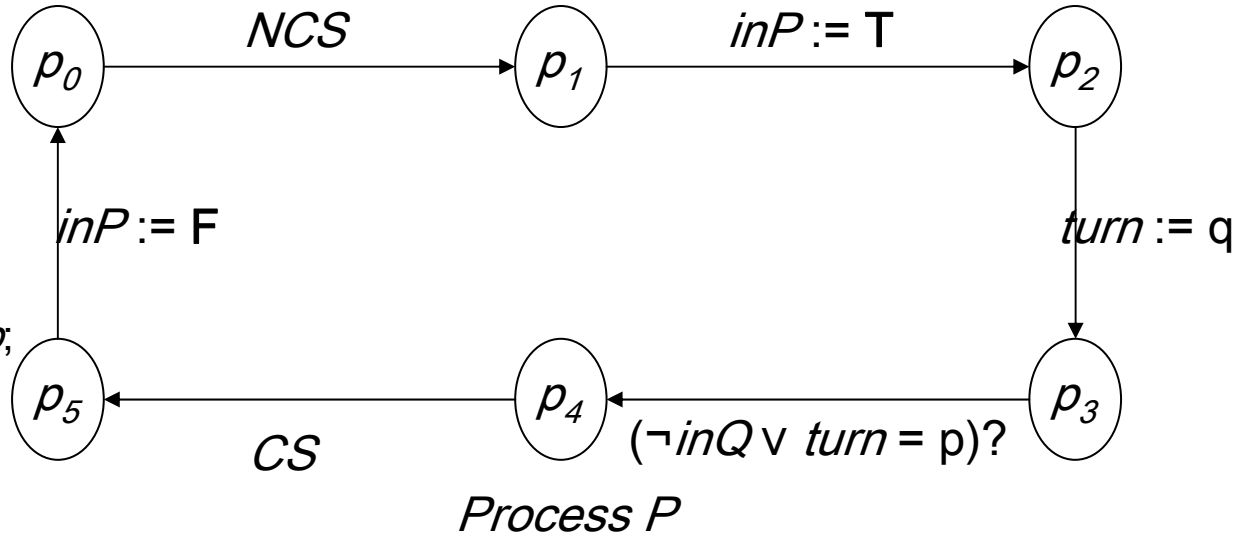
```
        Q:
while true do
        begin
        non-critical section;
        inQ := T;
        turn := p;
        await ¬inP ∨ turn = q do skip;
        critical section;
        inQ := F
        end
```

- *inQ* and *inP* are true when the appropriate process requests access. Initially, they are **F**.
- *turn* indicates which process has the priority, if both want in. Initial value can be either p or q, arbitrarily.

# Formal Model

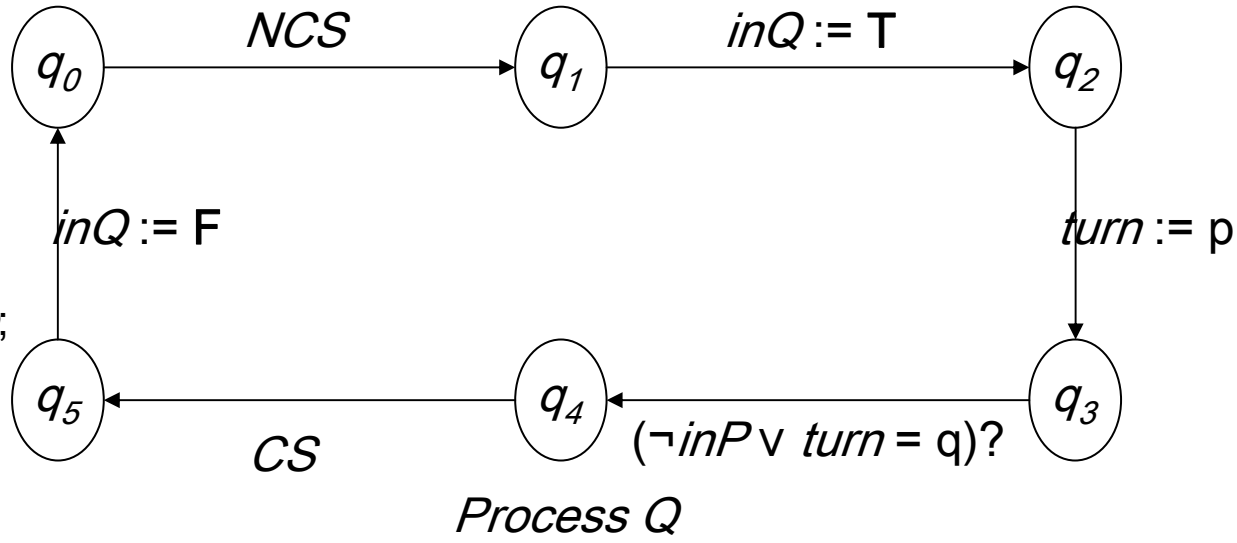
P:

```
while true do
      begin
      non-critical section;
      inP := T;
      turn := q;
      await ¬inQ ∨ turn = p do skip;
      critical section;
      inP := F
       end
```




Q:

```
while true do
      begin
      non-critical section;
      inQ := T;
      turn := p;
      await ¬inP ∨ turn = q do skip;
      critical section;
      inQ := F
       end
```



# A Program as a Temporal F-O Theory

Signature:

- Global objects:
  - Function constants and predicate constants associated with the types of the program variables.
    - For example: - and <.
  - As many logical variables as we want.
- Local objects: to describe the system state
  - For each label $\ell$, *at-*$\ell$=T iff the control state contains $\ell$.
  - For each transition T, *Next(T)* = T in state s iff the next state is the T-successor of s.
  - For each process P, *Next(P)*=$\vee_{T \in P}$*Next(T).*
  - For each program variable *x*, there's a local logical constant (also denoted *x*) whose value in state s is the value of the program variable x in s.

# A Program as a Temporal F-O Theory

Axioms:

1. The axioms of temporal predicate calculus, restricted to the language defined by the signature.
2. Axioms that describe the data types used by the system. These give the semantics of the function constants and the predicate constants of the language.
   - Example: if we have integers, we include the axioms of Number Theory, so we can use predicates like <, and functions like +.
3. Axioms that give a formal description of the system itself. An axiom schema is associated with each transition.
4. Axioms to model the semantics of the local propositions associated with the processes and the control points (nodes).
5. A special axiom to formalize the execution mechanism.

# Axioms

- For each process P, for each transition

  T: $(\ell\ ,\ C\rightarrow(x_1,\ldots,x_k):=(t_1,\ldots,t_k)\ ,\ \ell')\ ;\ T\in Tr_P$

  we add the axiom schema:

  $B_T$: $\Box\{[\mathit{Next(T)} \wedge Z(x_1/t_1,\ldots,x_k/t_k,\mathit{at\text{-}\ell'}\leftarrow T)] \rightarrow$
  $[\mathit{at\text{-}\ell} \wedge C \wedge \bigcirc(\mathit{at\text{-}\ell'} \wedge Z)]\}$

  where Z is any formula without temporal operators.

  $B_T$ indicates when T can be executed, and formalized its effect.

# Axioms

- For example, let's use T: ($P_1$, p:=p+1, $P_0$) of the P-C system. We'll take the formula I for Z. We get the axiom:

$$\Box\{[\mathit{Next(T)} \wedge \mathrm{I}(p/p+1, \mathit{at\text{-}P_0} \leftarrow \mathrm{T})] \rightarrow [\mathit{at\text{-}P_1} \wedge \mathrm{T} \wedge \bigcirc(\mathit{at\text{-}P_0} \wedge \mathrm{I})]\}$$

which "reduces" to:

$$\Box\{[\mathit{Next(T)} \wedge p\text{-}c<N \wedge (\mathit{at\text{-}C_0} \rightarrow p \geq c) \wedge (\mathit{at\text{-}C_1} \rightarrow p>c)] \rightarrow [\mathit{at\text{-}P_1} \wedge \bigcirc(\mathit{at\text{-}P_0} \wedge p\text{-}c<N \wedge [\mathit{at\text{-}C_0} \rightarrow p \geq c] \wedge [\mathit{at\text{-}C_1} \rightarrow p>c])]\} .$$

# Axioms

- Notation: if a,b,c are propositions, then “a+b+c=1” means that one and only one of these three is true.
- The axioms about the control points and the execution mechanism:
  - For each process P, $\Box(\Sigma_{l\in Lp} at\text{-}\ell = 1)$.
  - For each process P, $\Box[Next(P) = \Sigma_{T\in Trp} Next(T)]$.
  - If $P_i \neq P_j$ then, $\Box\neg[Next(P_i) \wedge Next(P_j)]$.
  - The execution mechanism: For each control state $(\ell_1,\ldots,\ell_n)$, $\Box\{(at\text{-}\ell_1 \wedge\ldots\wedge at\text{-}\ell_n)\rightarrow [(\vee_{j=1:n} Next(P_j) \wedge out(\ell_j)) \vee (\wedge_{i=1:n}\neg out(\ell_j))]\}$.
  - Fairness: For each process P, $\Box[\Diamond Next(P) \vee \vee_{\ell\in Lp}(at\text{-}\ell \wedge \neg\Box\Diamond out(\ell))]$ .

# Program Properties

- Initial conditions:

  Init: *at-$p_0$* ∧ *at-$q_0$* ∧ ¬inP ∧ ¬inQ ∧ (turn=p ∨ turn=p).
- Invariance properties: Init → □Z (Z with no temporal ops).
  - An *invariant* : A formula I with no temporal operators such that

    Init → I, □(I→○I), I → Z
  - Already saw examples for the P-C program.
  - Examples for Peterson's algorithm, soon.
- Liveness properties: Init → □(Y→◇Z) (Y,Z with no temporal ops).
- It's possible to generalize liveness and invariance properties s.t. any property modeled by a temporal formula becomes the conjunction of an invariance and a liveness properties.
- Intuitively, an invariance property asserts that nothing bad happens, and a liveness property asserts that something good will eventually happen.

# Invariance Properties for Peterson's Algorithm

- The mutual exclusion property:

  Init $\rightarrow \Box\neg[(at\text{-}p_4 \vee at\text{-}p_5) \wedge (at\text{-}p_4 \vee at\text{-}p_5)]$ .

- Deadlock freedom: All computations are infinite. The output condition of all nodes is T except $p_3$ and $q_3$, but $out(p_3) \vee out(q_3) = \mathsf{T}$, which implies the desired result: No final states.

- We will prove that the following is an invariant:

  $I : ((at\text{-}p_0 \vee at\text{-}p_1) \leftrightarrow \neg inP) \wedge ((at\text{-}q_0 \vee at\text{-}q_1) \leftrightarrow \neg inQ)$

  $\wedge$ (turn=p $\vee$ turn=q)

  $\wedge$ (at-$q_3$ $\vee$ at-$q_4$ $\vee$ at-$q_5$)$\rightarrow$(turn=p $\vee$ at-$p_3$)

  $\wedge$ (at-$p_3$ $\vee$ at-$p_4$ $\vee$ at-$p_5$)$\rightarrow$(turn=q $\vee$ at-$q_3$)

# Invariance Properties for Peterson's Algorithm

- First, when Init is T then $I$ reduces to:

  $((T \vee F) \leftrightarrow T) \wedge ((T \vee F) \leftrightarrow T)$

  $\wedge T$

  $\wedge (F \vee F \vee F) \rightarrow (turn=p \vee F)$

  $\wedge (F \vee F \vee F) \rightarrow (turn=q \vee F)$

  which reduces to T , implying that $(Init \rightarrow I)$ is valid.

# Invariance Properties for Peterson's Algorithm

- Secondly, we need to prove that $\Box(I \rightarrow \bigcirc I)$.
- Sufficient to prove that for each transition T, $(I \wedge \mathit{Next}(T)) \rightarrow \bigcirc I$.
- Let's consider $T_2^{(p)}$: ($p_2$, *turn*:=q, $p_3$)
- To create the axiom associated with this transition, we need to replace I for Z in $B_T$:

$$B_{T2(p)}: \Box\{[\mathit{Next}(T_2^{(p)}) \wedge I(\mathit{turn}/q, \mathit{at\text{-}p_3} \leftarrow T)] \rightarrow [\mathit{at\text{-}p_2} \wedge \bigcirc(\mathit{at\text{-}p_3} \wedge I)]\}$$

# Invariance Properties for Peterson's Algorithm

- The required property will be proved if we prove the following:

  A: $[I \wedge \mathit{Next}(T_2^{(p)})] \rightarrow [\mathit{Next}(T_2^{(p)}) \wedge I(\mathit{turn}/q, \mathit{at\text{-}p_3} \leftarrow T)]$.

- Taking into account that $(\mathit{Next}(T_2^{(p)}) \rightarrow \mathit{at\text{-}p_2})$ , A's first part is:

  $\mathit{Next}(T_2^{(p)}) \wedge \mathit{inP} \wedge ((\mathit{at\text{-}q_0} \vee \mathit{at\text{-}q_1}) \leftrightarrow \neg \mathit{inQ})$

  $\wedge$ (turn=p $\vee$ turn=q)

  $\wedge$ (at-$q_3$ $\vee$ at-$q_4$ $\vee$ at-$q_5$)$\rightarrow$turn=p ,

- and A's second part is:

  $\mathit{Next}(T_2^{(p)}) \wedge \mathit{inP} \wedge ((\mathit{at\text{-}q_0} \vee \mathit{at\text{-}q_1}) \leftrightarrow \neg \mathit{inQ})$ ,

  which means that A is valid.

- We handle the rest of the transitions similarly.

# Invariance Properties for Peterson's Algorithm

- Last, we need to prove (I→Z), where

  Z: ¬[(*at-p*$_4$ ∨ *at-p*$_5$) ∧ (*at-q*$_4$ ∨ *at-q*$_5$)] (mutual exclusion in a single state)
- The negation of (I→Z) is:

  I ∧ (*at-p*$_4$ ∨ *at-p*$_5$) ∧ (*at-q*$_4$ ∨ *at-q*$_5$) ,
- which can be written as:

  (*at-p*$_4$ ∨ *at-p*$_5$) ∧ (*at-q*$_4$ ∨ *at-q*$_5$) ∧ *inP* ∧ *inQ*

∧ (*turn* = p ∨ *turn* = q)

∧ T→(*turn* = p ∨ F)

∧ T→(*turn* = q ∨ F)

- which implies (*turn* = p ∧ *turn* = q) , which is a contradiction.

# Fairness of Peterson's Algorithm

- Want to prove that the delay between a request for access to the critical section and the access itself is always finite. This needs to be proved for each process, but the code is symmetric.
- Formally: $\Box(\text{at-p}_3 \rightarrow \Diamond \text{at-p}_4)$ or, $\Box(\text{at-p}_3 \wedge I \rightarrow \Diamond \text{at-p}_4)$.
- Proof is based on the next basic theorems:
  1. $\Box[(\textit{at-p}_3 \wedge \textit{at-q}_3 \wedge I \wedge \textit{turn}=\text{p}) \rightarrow \bigcirc \textit{at-p}_4]$
  2. $\Box[(\textit{at-p}_3 \wedge \textit{at-q}_3 \wedge I \wedge \textit{turn}=\text{q}) \rightarrow \bigcirc(\textit{at-p}_3 \wedge \textit{at-q}_4 \wedge I)]$
  3. $\Box[(\textit{at-p}_3 \wedge \textit{at-q}_4 \wedge I) \rightarrow \bigcirc(\textit{at-p}_3 \wedge \textit{at-q}_5 \wedge I)]$
  4. $\Box[(\textit{at-p}_3 \wedge \textit{at-q}_5 \wedge I) \rightarrow \bigcirc(\textit{at-p}_3 \wedge \textit{at-q}_0 \wedge I)]$
  5. $\Box[(\textit{at-p}_3 \wedge \textit{at-q}_0 \wedge I) \rightarrow \bigcirc(\textit{at-p}_4 \vee (\textit{at-p}_3 \wedge \textit{at-q}_1 \wedge I))]$
  6. $\Box[(\textit{at-p}_3 \wedge \textit{at-q}_1 \wedge I) \rightarrow \bigcirc(\textit{at-p}_4 \vee (\textit{at-p}_3 \wedge \textit{at-q}_2 \wedge I))]$
  7. $\Box[(\textit{at-p}_3 \wedge \textit{at-q}_2 \wedge I) \rightarrow \bigcirc(\textit{at-p}_4 \vee (\textit{at-p}_3 \wedge \textit{at-q}_3 \wedge I \wedge \textit{turn}=\text{p}))]$

# Fairness of Peterson's Algorithm

- The proof:

| | | |
|---|---|---|
| 8. | $\Box[(at\text{-}p_3 \wedge at\text{-}q_3 \wedge \mathrm{I} \wedge turn{=}\mathrm{p}) \rightarrow \Diamond at\text{-}p_4]$ | (1) |
| 9. | $\Box[(at\text{-}p_3 \wedge at\text{-}q_2 \wedge \mathrm{I}) \rightarrow \Diamond at\text{-}p_4]$ | (7,8) |
| 10. | $\Box[(at\text{-}p_3 \wedge at\text{-}q_1 \wedge \mathrm{I}) \rightarrow \Diamond at\text{-}p_4]$ | (6,9) |
| 11. | $\Box[(at\text{-}p_3 \wedge at\text{-}q_0 \wedge \mathrm{I}) \rightarrow \Diamond at\text{-}p_4]$ | (5,10) |
| 12. | $\Box[(at\text{-}p_3 \wedge at\text{-}q_5 \wedge \mathrm{I}) \rightarrow \Diamond at\text{-}p_4]$ | (4,11) |
| 13. | $\Box[(at\text{-}p_3 \wedge at\text{-}q_4 \wedge \mathrm{I}) \rightarrow \Diamond at\text{-}p_4]$ | (3,12) |
| 14. | $\Box[(at\text{-}p_3 \wedge at\text{-}q_3 \wedge \mathrm{I}) \rightarrow \Diamond at\text{-}p_4]$ | (8,2,13) |
| 15. | $\Box[(at\text{-}p_3 \wedge \mathrm{I}) \rightarrow \Diamond at\text{-}p_4]$ | (9,10,11,12,13,14) |

- Fairness of the scheduler is not required for that proof, but what if we are interested in a slightly stronger property of fairness: $\Box(\text{at-p}_0 \rightarrow \Diamond \text{at-p}_4)$ , we need to assume that the scheduler itself is fair.

# Fairness of Peterson's Algorithm

- Since we already proved that $\Box(at\text{-}p_3 \rightarrow \Diamond at\text{-}p_4)$ , it's enough to prove that $\Box(at\text{-}p_0 \rightarrow \Diamond at\text{-}p_3)$ , or equivalently, $\Box(at\text{-}p_0 \rightarrow \Diamond at\text{-}p_1)$, $\Box(at\text{-}p_1 \rightarrow \Diamond at\text{-}p_2)$, $\Box(at\text{-}p_2 \rightarrow \Diamond at\text{-}p_3)$ .
- Formally, the fairness of the scheduler is a set of axioms, one for each process. For P,

  $\Box[\Diamond Next(\mathrm{P}) \vee (at\text{-}p_3 \wedge \neg\Box\Diamond(\neg inQ \vee turn{=}\mathrm{p}))]$
- Proof that $\Box(at\text{-}p_0 \rightarrow \Diamond at\text{-}p_1)$ :

  1. $\Box[at\text{-}p_0 \rightarrow \Diamond Next(\mathrm{P})]$ (fairness)
  2. $\Box[(at\text{-}p_0 \wedge \Diamond Next(\mathrm{P})) \rightarrow \bigcirc at\text{-}p_1]$ (semantic axiom)
  3. $\Box[at\text{-}p_0 \rightarrow \Diamond Next(\mathrm{P})]$ (1,2)
- The remaining are proved in the same way.
- **The key in program verification is finding an appropriate invariant!**

# Verification Using Propositional Temporal Logic

- The method described is widely used, but:
  - It's sometimes incredibly hard to find the adequate invariants, and
  - The proofs can be tedious.
- The alternative method uses propositional temporal logic (the one from last week), and allows **automation** of the verification process.
- Problem is: It only works if the system has only a **finite** number of possible states ("finite-state system"):
  - **Finite number of control states (always true), and**
  - **Finite number of possible values for each variable (sometimes not).**
- We will show that any finite-state system can be represented as a Büchi automaton, so that the words it accepts correspond to the executions of the program.

# Verification Using Propositional Temporal Logic

- We saw last week that for a given temporal logic formula, there is a way to build a Büchi automaton that accepts exactly the sequences that satisfy the formula. Our method of proof will be based on that. The method is:
  1. Build the automaton for the negation of the formula, and the automaton describing the program.
  2. Intersect the two.
  3. Check that the intersection automaton is empty.
  - If it is empty, there is no execution of the program that satisfies the negation of the formula. Meaning, all executions of the program, satisfy the formula – exactly what we want to prove.
  - Hadn't we used the negation of the formula, we would've had to check the complement of the automaton, but complementing an automaton is an expensive operation, that we wish to avoid.

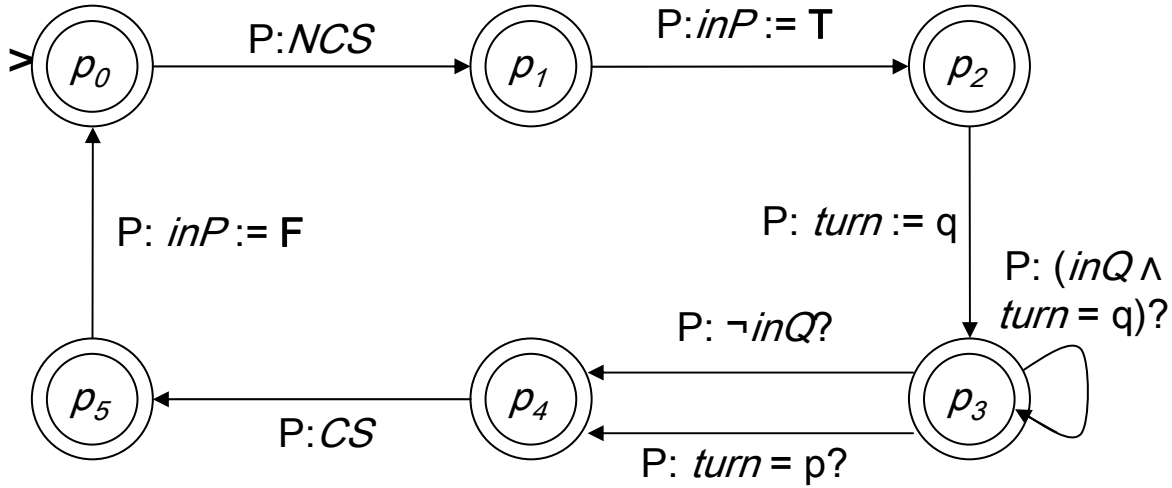
# Modeling Programs with Finite Automata

- We will demonstrate the method on Peterson's algorithm.
  - Each process can be viewed as an automaton. The nodes correspond to states of the automaton, and the arcs – to transitions. Therefore, Σ = the set of statements of the program.
  - We still need to represent the variables and their values.
  - We also need to combine the two process into a single automaton.
  - Solution:
    - We will represent each variable as a separate automaton.
    - We will define an operation of finite-state processes that represent the concurrent execution of them.
    - That will allow us to combine the two processes and the three variables into a single automaton representing the algorithm.
  - First, we'll see the automata, and then how to combine them.

# The Automata Representing the Processes

P:

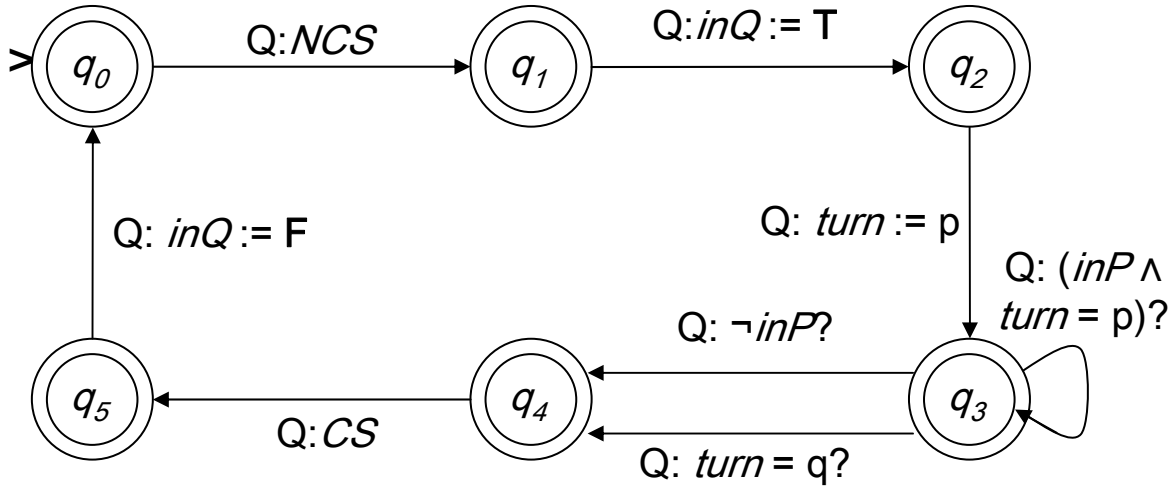
while *true* do
begin
*non-critical section;*
*inP* := T;
*turn* := q;
await ¬*inQ* ∨ *turn* = p do *skip*;
*critical section;*
*inP* := F
end




*Process P*

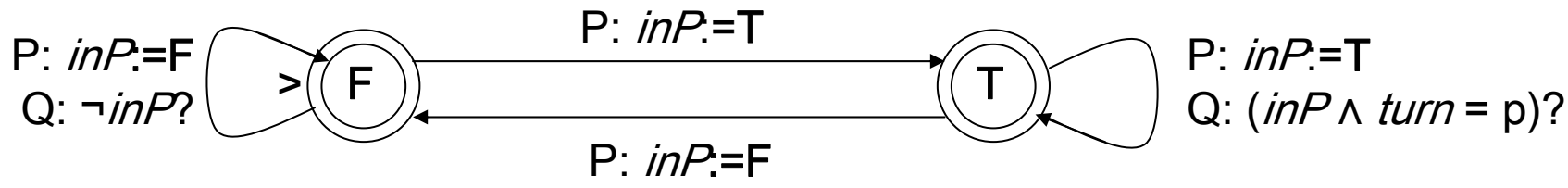
Q:

while *true* do
begin
*non-critical section;*
*inQ* := T;
*turn* := p;
await ¬*inP* ∨ *turn* = q do *skip*;
*critical section;*
*inQ* := F
end

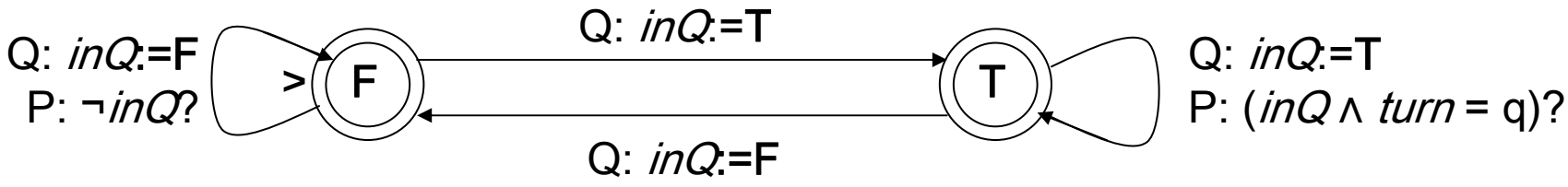



*Process Q*

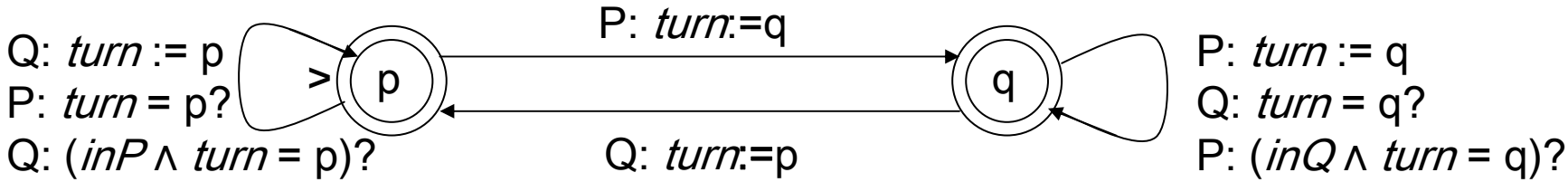
# The Automata Representing the Variables




*Automaton for inP*




*Automaton for inQ*



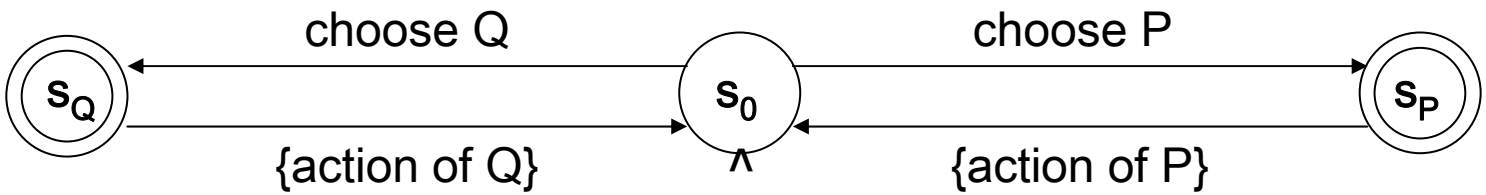

*Automaton for turn*

# How to Combine the Automata

- Let's explain the combination of just two automata first.
  - We have $A_1=(\Sigma_1,S_1,\rho_1,S_{01},F_1)$ and $A_2=(\Sigma_2,S_2,\rho_2,S_{02},F_2)$.
  - The *partly synchronized product* of $A_1$ and $A_2$ is the generalized Büchi automaton $A=(\Sigma,S,\rho,S_0,\mathscr{F})$ formally defined by:
    - $\Sigma = \Sigma_1 \cup \Sigma_2$
    - $S=S_1 \times S_2$ , $S_0=S_{01} \times S_{02}$
    - $\mathscr{F} = \{F_1 \times S_2, S_1 \times F_2\}$
    - $(u,v) \in \rho((s,t),a)$ when:
      - $a \in \Sigma_1 \cap \Sigma_2$ and $u \in \rho_1(s,a)$ and $v \in \rho_2(t,a)$,
      - $a \in \Sigma_1 \setminus \Sigma_2$ and $u \in \rho_1(s,a)$ and $v=t$,
      - $a \in \Sigma_2 \setminus \Sigma_1$ and $u=s$ and $v \in \rho_2(t,a)$.

# Modeling the Fairness Hypothesis

- The fairness hypothesis: “Any unblocked process will eventually execute an action”.
- We built the automata representing P and Q, we added transition to make sure there is no possibility of blocking.
- So, we can use a simpler fairness hypothesis: “Every process will eventually execute some action”. In Peterson’s case, it means that the only allowable infinite executions are those in which actions of both P and Q occur infinitely often.
- We enforce this by adding one more automaton to the partially synchronized product - the one of the scheduler:



# Specification and Verification

- What did we want to prove?
  - Mutual exclusion: Two symmetrical conditions, one for each process:
    - $\Box[(\mathrm{P}{:}\mathit{CS}) \rightarrow (\neg(\mathrm{Q}{:}\mathit{CS})\,\mathcal{U}(\mathrm{P}{:}\mathit{inP}{:=}\mathrm{F}))]$
    - $\Box[(\mathrm{Q}{:}\mathit{CS}) \rightarrow (\neg(\mathrm{F}{:}\mathit{CS})\,\mathcal{U}(\mathrm{Q}{:}\mathit{inQ}{:=}\mathrm{F}))]$
  - Liveness property: If a process wants in, it will eventually get in:
    - $\Box[(\mathrm{P}{:}\mathit{inP}{:=}\mathrm{T}) \rightarrow \Diamond(\mathrm{P}{:}\mathit{CS})]$
    - $\Box[(\mathrm{Q}{:}\mathit{inQ}{:=}\mathrm{T}) \rightarrow \Diamond(\mathrm{Q}{:}\mathit{CS})]$
- The verification process is not designed to be applied manually, but rather by a computer. A few notes on complexity:
  - The size of the product automata can grow as the product of the sizes of the automata representing the processes and the variables.
  - The size of the automaton representing a formula can grow exponentially as a function of the length of the formula.
  - However, once we have the intersection automaton, that we need to check that is empty, the time complexity of checking emptiness is linear in the size of the automaton.

# BrTL Application

- A similar approach to program verification uses BrTL.
- We can think of the automaton that represents the program as a branching-time temporal interpretation.
- Verifying that all executions of the program satisfy a specification ϕ, is equivalent to verifying that the interpretation is a model of the formula ∀ϕ.
- If we restrict ourselves to CTL, the problem is, given a finite set P of propositions, an Interpretation $\mathcal{I}$, and a formula A involving only elements of P, to determine whether $\mathcal{I}$ is a model of A.

# Branching-time Application

- Basically, the interpretation gives us a table for each state s, that tells us the truth value of each element of P in s.
- We need to complete these tables to contain the truth values of all sub-formulas of A, including a itself.
- First we consider the sub-formulas of A that include only one logical connective, or one compound temporal operator ($\forall\bigcirc, \exists\bigcirc, \forall\Box, \exists\Box, \forall\Diamond, \exists\Diamond, \forall\mathcal{U}, \exists\mathcal{U}$).
- Example: $\forall\bigcirc$p=**T** iff p=**T** in all the immediate successors of s.
- If the tables contain the truth values for sub-formulas containing n connectives, it's clear how to determine the values of the sub-formulas containing n+1 connectives.
- The time complexity is O((size of the formula a) * (size of the interpretation $\mathcal{I}$)).

# The ∃nd