Reasoning
Sequential program reasoning is usually done with a proof calculus like Hoare Logic:

$$\{\varphi\}\ P\ \{\psi\}$$

The program P starts in a state satisfying the pre-condition φ and it terminates, it will end in a state satisfying the post-condition ψ.

Property
A linear temporal property is a set of behaviours.
A **safety property** states that something bad does not happen. These are properties that may be violated by a finite prefix of a behaviour.
A **liveness property** states that something good will happen. These are properties that can always be satisfied eventually.

Limits
If σ is a behaviour, we write $\sigma|_k$ to denote the prefix of σ comprising its first k states.
The limit closure of a set $A \subseteq \Sigma^\omega$, denoted $\overline{A}$, is defined as follows:

$$\overline{A} = \{\sigma \in \Sigma^\omega \mid \forall n \in \mathbb{N}.\, \exists \sigma' \in A.\, \sigma|_n = \sigma'|_n\}$$

A set A is limit-closed if $\overline{A} = A$.
A set A is called dense if $A = \Sigma^\omega$, i.e., the closure is the space of all behaviours.
Safety properties are limit-closed, and liveness properties are dense.

Alpern & Schneider's Theorem
Every property is the intersection of a safety and a liveness property.

$$P = \underbrace{\overline{P}}_{\text{closed}} \cap \underbrace{\Sigma^\omega \setminus (\overline{P} \setminus P)}_{\text{dense}}$$

Semantics
Let $\sigma_0\sigma_1\sigma_2\sigma_3$ … be a behaviour. Then define notation:
- $\sigma|_0 = \sigma$
- $\sigma|_1 = \sigma_1\sigma_2\sigma_3$ …
- $\sigma|_{n+1} = (\sigma|_1)|_n$

The models of LTL are behaviours. For atomic propositions, we just look at the first state. We often identify states with the set of atomic propositions they satisfy.
$\sigma \vDash p \Leftrightarrow p \in \sigma_0$
$\sigma \vDash \varphi \wedge \psi \Leftrightarrow \sigma \vDash \varphi \ and \ \sigma \vDash \psi$
$\sigma \vDash \neg\, \varphi \Leftrightarrow\ \sigma \nvDash \varphi$
$\sigma \vDash \circ\, \varphi \Leftrightarrow \sigma|_1 \vDash \varphi$
$\sigma \vDash \varphi\ \mathcal{U}\ \psi \Leftrightarrow\ \exists i{:}\, \sigma|_i \vDash \psi\ \vee\ \forall j < i{:}\, \sigma|_j \vDash \varphi$
We say $P \vDash \varphi \Leftrightarrow \forall \sigma \in [\![P]\!].\ \sigma \vDash \varphi$.

Derived Operators
The operator $\Diamond\, \varphi$ ("finally" or "eventually") says that φ will be true at some point.
The operator $\Box\varphi$ ("globally" or "always") says that φ is always true from now on.

Critical Sections
A **critical section** is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point in time, only one process must be executing its critical section.

Desiderata
We want to ensure two main properties and two secondary ones:
- **Mutual Exclusion**: No two processes are in their critical section at the same time.
- **Eventual Entry** (or starvation-freedom): Once it enters its pre-protocol, a process will eventually be able to execute its critical section.
- **Absence of Deadlock**: The system will never reach a state where no actions can be taken from any process.
- **Absence of Unnecessary Delay**: If only one process is attempting to enter its critical section, it is not prevented from doing so.

Out of the above, Eventual Entry is a liveness property, and the rest are safety properties.

Fairness
The fairness assumption means that if a process can always make a move, it will eventually be scheduled to make that move.
Let enabled (π) and taken (π) be predicates true in a state iff an action π is enabled and taken respectively.
Then,
Weak fairness for action π is expressible as $\Box(\Box enabled(\pi) \Rightarrow \Diamond\ taken(\pi))$.
Strong fairness for action π is expressible as $\Box(\Box\, \Diamond\, enabled(\pi) \Rightarrow \Diamond\ taken(\pi))$.

Transition Diagrams
A transition diagram is a tuple (L, T, s, t) where:
- L is a set of locations (program counter values).
- T is a set of transitions.
- $s \in L$ is an entry location.
- $t \in L$ is an exit location.

A transition is written as $l_i \xrightarrow{g;f} \ell_j$ where:
- ℓi and ℓj are locations.
- g is a guard $\Sigma \to \mathbb{B}$.
- f is a state update $\Sigma \to \Sigma$.

Floyd Verification
The definition of a Hoare triple for partial correctness is $\{\varphi\}\ P\ \{\psi\}$ which states that if the program P successfully executes from a starting state satisfying φ, the result state will satisfy ψ. This is a *safety* property.
Given a transition diagram (L, T, s, t), we can verify partial correctness by
1. Associate with each location $\ell \in L$ an *assertion* $Q(\ell)$: $\Sigma \to \mathbb{B}$
2. Prove that this assertion network is *inductive*, that is, for each transition in T $l_i \xrightarrow{g;f} \ell_j$ show that $Q(\ell_i) \wedge g \Rightarrow Q(\ell_j) \circ f$.
3. Show that $p \Rightarrow Q(s)$ and $Q(t) \Rightarrow \psi$.

Parallel Composition
Given two processes P and Q with transition diagrams (L_P, T_P, s_P, t_P) and (L_Q, T_Q, s_Q, t_Q) respectively, the **parallel composition** of P and Q, written $P \parallel Q$ is defined as (L, T, s, t) where:
- $L = L_P \times L_Q$
- $s = s_P s_Q$
- $t = t_P t_Q$
- $p_i q_i \xrightarrow{g;f} p_j q_i \in T$ if $p_i \xrightarrow{g;f} p_j \in T_p$
- $p_i q_i \xrightarrow{g;f} p_i q_j \in T$ if $q_i \xrightarrow{g;f} q_j \in T_Q$

The problem with using the Floyd method is that the number of locations and transitions grows exponentially as the number of processes increases.

Owicki-Gries Method
To show $\{\varphi\}\ P \parallel Q\ \{\psi\}$:
1. Define local assertion networks $\mathcal{P}$ and $\mathcal{Q}$ for both processes and show that they're inductive.
2. For each location $p \in L_p$, show that $\mathcal{P}(p)$ is not falsified by any transition of Q, that is, for each $q \xrightarrow{g;f} q' \in T_Q : \mathcal{P}(p) \wedge \mathcal{Q}(q) \wedge g \Rightarrow \mathcal{P}(p) \circ f$
3. Vice versa for Q.
4. Show that $\varphi \Rightarrow \mathcal{P}(s_P) \wedge \mathcal{Q}(s_Q)$ and $\mathcal{P}(t_P) \wedge \mathcal{Q}(t_Q) \Rightarrow \psi$.

The Owicki-Gries method generalises to n processes just by requiring more interference freedom obligations.

Derived Assertion Network
The automatic assertion network we get for the parallel composition from the Owicki-Gries method is the conjunction of the local assertions at each of the component states.

Invariants
If we have an assertion network where every assertion is the same, then we don't need to prove interference freedom — the local verification conditions already show that the invariant is preserved. This is known as having an invariant.

Critical Section Algorithms
Linear waiting is the property that says the number of times a process is "overtaken" (bypassed) in the preprotocol is bounded by n (the number of processes).

Dekker's Algorithm

var wantp; wantq ← False; False
var turn ← 1

forever do	**forever do**
p1 non-critical section	q1 non-critical section
p2 wantp = True;	q2 wantq = True;
p3 **while** wantq **do**	q3 **while** wantp **do**
p4 **if** turn = 2 **then**	q4 **if** turn = 1 **then**
p5 wantp ← False;	q5 wantq ← False;
p6 **await** turn = 1;	q6 **await** turn = 2;
p7 wantp True	q7 wantq ← True
p8 **critical section**	q8 **critical section**
p9 turn ← 2	q9 turn ← 1
p10 wantp ← False	q10 wantq ← False

Peterson's Algorithm

integer array in[1..n] = [0, …, 0]
integer array in[1..n] = [0, …, 0]
forever do
p1: non-critical section
p2: **for all** processes j
p3: in[i] ← j
p4: last[j] = i
p5 **for all** processes k != i
p6: **await** in[k] < j or last[j] != i
p7: **critical section**
p8: in[i] ← 0

Bakery's Algorithm

boolean array[1..n] choosing ← [false, …]
integer array[1..n] number ← [0, …, 0]
forever do
p1: non-critical section
p2: choosing[i] ← true
p3: number[i] ← 1 + max(number)
p4: choosing[i] ← false
p5: **for all** other processes j
p6: **await** choosing[j] = false
p7: **await** (number[j] = 0) or (number[i] << number[j])
p8: **critical section**
p9: number[i] ← 0

Fast algorithm

integer gate1, gate2 ← 0
boolean wantp, wantq ← false

forever do		**forever do**	
p1:	gate1 ← p	q1:	gate1 ← q
p2:	wantp ← true	q2:	wantq ← true
p3:	**if** gate2 != 0	q3:	**if** gate2 != 0
p4:	wantp ← false	q4:	wantq ← false
p5:	**goto** p1	q5:	**goto** q1
p6:	gate2 ← p	q6:	gate2 ← q
p7:	**if** gate1 != p	q7:	**if** gate1 != q
p8:	wantp ← false	q8:	wantq ← false
p9:	**await** wantq = false	q9:	**await** wantp = false
p10:	**if** gate2 != p: **goto** p1	q10:	**if** gate2 != q: **goto** q1
p11:	**else:** wantp ← true	q11:	**else:** wantq ← true
p12:	**critical section**	q12:	**critical section**
p13:	gate2 ← 0	q13:	gate2 ← 0
p14:	wantp ← false	q14:	wantq ← false

Szymanski algorithm

integer array flag[1..n] ← [0, …, 0]

forever do	
p1:	non-critical section
p2:	flag[i] ← 1
p3:	**await** all j. flag[j] <3
p4:	flag[i] ← 3
p5:	**if exists** j. flag[j] = 1 **then**
p6:	flag[i] ← 2
p7:	**await exists** j. flag[j] = 4
p8:	flag[i] ← 4
p9:	**await all** j < i. flag[j] <3
p10:	**critical section**
p11:	**await all** j > i. flag[j] <2 **or** flag[j] > 3
p12:	flag[i] ← 0

Semaphores

A semaphore can be abstractly viewed as a pair (v, L) of a natural number v and a set of processes L. The semaphore must always be initialised to some $(v, \emptyset)$.

inline wait(S) { d_step { S > 0; S-- }} **inline signal(S) { d_step { S ++ }}**

There are two basic operations a process p could perform on a semaphore S:
wait(*S*) or $P(S)$, decrements v if positive, otherwise adds p to L and blocks p
signal(*S*) or $V(S)$, if $L \neq \emptyset$, unblocks a member of L, otherwise increments v
This type of semaphore is known as a *weak* semaphore.
A *busy-wait* semaphore is when the set L is implicitly the set of (busy-)waiting processes on the integer.

Strong Semaphores

Replace the set L with a queue, such that processes are woken up in FIFO order. This guarantees linear waiting but is of course harder to implement and potentially more expensive than our previous weak or busy-wait semaphores.

Reasoning About Semaphores

For a semaphore $S = (v, L)$ initialised to $(k, \emptyset)$, the following invariants always hold:

- $v = k + \#signal(S) - \#wait(S)$
- $v \geq 0$

where

- $\#signal(S)$ is the number of times **signal(*S*)** has successfully executed
- $\#wait(S)$ is the number of times **wait(*S*)** has successfully executed

Here we define successful execution to mean that the process has proceeded to the following statement. So, if a process is blocked on a **wait(*S*)**, then $\#wait(S)$ will not increase until the process is unblocked.

Safety Properties

We observe the invariant that the number of processes in their CS is $wait(S) - \#signal(S)$.

Mutual Exclusion

We know:

- $v = 1 + \#signal(S) - \#wait(S)$
- $v \geq 0$
- $\#CS = \#wait(S) - \#signal(CS)$

From these invariants it is possible to show that $\#CS \leq 1$, i.e., mutual exclusion.

Absence Of Deadlock

Assume that deadlock occurs by all processes being blocked on **wait**, so no process can enter its critical section ($\#CS = 0$).
Then $v = 0$, contradicting our semaphore invariants above. So, there cannot be deadlock.

Liveness Properties

Eventual Entry For p

To simplify things, we will prove for only two processes, p and q.
Assume that p is starved, indefinitely blocked on the **wait**. Therefore $S = (0, L)$ and $p \in L$. We know therefore, substituting into our invariants:

- $0 = 1 + \#signal(S) - \#wait(S)$
- $\#CS = \#wait(S) - \#signal(S)$

From which we can conclude that $\#CS = 1$. Therefore q must be in its critical section and $L = \{p\}$.

By a progress assumption (that processes that are able to move will do so when asked by the scheduler), we know that eventually q will finish its CS and **signal(*S*)**. Thus, p will be unblocked, causing it to gain entry – a contradiction.

Producer Consumer Problem

A producer process and a consumer process share access to a shared buffer of data. This buffer acts as a queue. The producer adds messages to the queue, and the consumer reads messages from the queue. If there are no messages in the queue, the consumer blocks until there are messages.

bounded[N] queue[T] buffer = empty queue
semaphore full = (0, {})
semaphore empty = (N, {})

producer		consumer	
T d		T d	
loop forever		**loop forever**	
p1:	d = produce	p1:	wait(full)
p2:	wait(empty)	p2:	d = take(buffer)
p3:	append(d, buffer)	p3:	signal(empty)
p4:	signal(full)	p4:	consume(d)

Disadvantages

- **Lack of structure**: when building a large system, responsibility is diffused among implementers
- **Global visibility**: when something goes wrong, the whole program must be inspected, deadlocks are hard to isolate

Monitors

Condition variables are named FIFO queues of blocked processes. Processes executing a procedure of a monitor with condition variable cv can:

- Voluntarily suspend themselves using **waitC(*cv*)**
- Unblock the first suspended process by calling **signalC(*cv*)** or
- Test for emptiness of the queue: **empty(*cv*)**

Signalling disciplines

- S the signalling process
- W waiting on a condition variable
- E waiting on entry

E < S < W in Hoare's paper, E = S < W in Java.

Producer Consumer Problem in Monitor

```
bufferType buffer   empty
condition notEmpty
condition notFull

                                    operation take()
operation append(datatype V)          datatype W
  if buffer is full                   if buffer is empty
  waitC(notFull)                      waitC(notEmpty)
  append(V, buffer)                   W ← head(buffer)
  signalC(notEmpty)                   signalC(notFull)
                                      return W
```

Message Passing

- $ch \Leftarrow x$ to mean "send a message x into channel ch " (ch ! x)
- $ch \Rightarrow y$ to mean "receive a message from ch and store in y" (ch ? x)

If a channel is **synchronous**, the queue has capacity 0. Both the send and the receive operation block until they both are ready to execute. When they are, they proceed at the same time and the value of x is assigned to y.
If a channel is **asynchronous**, the send operation doesn't block. It appends the value of x to the FIFO queue associated with the channel ch. Only the receive operation blocks until the channel ch contains a message. When it does, the oldest message is removed, and its content is stored in y.

Producer Consumer Problem in Channel

integer n ← 0; integer high ← 0
set of node IDs deferred ← {};
boolean requestCS ← false;

Main		Receive	
forever do		**forever do**	
p1:	non-critical section	p1:	receive(request, source, reqNum)
p2:	requestCS ← true	p2:	high ← max(high, reqNum)
p3:	n ← high + 1	p3:	**if** not requestCS or
p4:	**for all** other nodes N		(reqNum, source) < lex(n, id)
p5:	send(request, N, myID, n)	p4:	send(reply, source, id)
p6:	**await** replies from other	p5:	**else** add source to deferred
p7:	**critical section**		
p8:	requestCS ← false		
p9:	**for all** nodes N in deferred		
p10:	**remove** N from deferred		
p11:	send(reply, N, myID)		

Termination

For programs that do terminate, termination is often the most important liveness property. In addition to the typical cause of non-termination for sequential programs, namely divergence, concurrent programs can also deadlock. Termination = convergence + deadlock freedom.

A program is ϕ-convergent if it cannot diverge (run forever) when started in an initial state satisfying ϕ. Instead, it must terminate, or become deadlocked.

To prove convergence, we prove that there is a bound on the remaining computation steps from any state that the program reaches.

Ordered & Wellfounded Sets

In maths, this bound condition is formalised by the concept of a wellfounded set. Recall that, on a set W, the binary relation $\prec \subseteq W_2$ is a (strict) partial order, if it is:

- irreflexive ($a \not\prec a$)
- asymmetric ($a \prec b \implies b \not\prec a$)
- transitive ($a \prec b \wedge b \prec c \implies a \prec c$)

A partially ordered set $(W, \prec)$ is wellfounded if every descending sequence $\langle w_0 \succ w_1 \succ \cdots \rangle$ in $(W, \prec)$ is finite.

Floyd's Wellfoundedness Method

Given a transition diagram $P = (L, T, s, t)$ and a precondition ϕ, we can prove ϕ-convergence of P by:

1. finding an inductive assertion network $Q: L \to (\Sigma \to \mathbb{B})$ and showing that $\vDash \phi \implies Q_s$
2. choosing a wellfounded set $(W, \prec)$ and a network $(\rho_\ell)_{\ell \in L}$ of partially defined ranking functions from Σ to W such that:
   - Q_ℓ implies that ρ_ℓ is defined
   - every transition $\ell \xrightarrow{b;f} \ell' \in T$ decreases the ranking function, that is: $\vDash Q_\ell \wedge b \implies \rho_\ell \succ (\rho'_\ell \circ f)$

This method is sound, that is, it indeed establishes ϕ-convergence.

This method is also semantically complete, that is, if P is ϕ-convergent, then there exist assertion and ranking function networks satisfying the verification conditions for proving convergence.

The Calculus of Communicating Systems (CCS)

- Is a process algebra, a simple formal language to describe concurrent systems
- Is given semantics in terms of labelled transition systems
- Was developed by Turing-award winner Robin Milner in the 1980s
- Has an abstract view of synchronization that applies well to message passing

It provides a symbolic way to describe transition diagrams, and reason about them symbolically rather than diagrammatically.

Processes

Processes in CCS are defined by equations.

For example, the equation

$$\mathbf{CLOCK} = tick$$

defines a process **CLOCK** that simply executes the *action* "tick" and then terminates. This process corresponds to the first location in this *labelled transition system* (LTS):

$$\cdot \xrightarrow{tick} \cdot$$

An LTS is like a transition diagram, save that our transitions are just abstract actions, and we have no initial or final location.

Action Prefixing

If a is an action and P is a process then $x.P$ is a process that executes x before P. This brackets to the right, so $x.y.z.P = x.(y.(z.P))$.

For example, the equation

$$\mathbf{CLOCK_2} = tick.tock$$

defines a process called $\mathbf{CLOCK_2}$ that executes the action "tick" then the action "tock" and then terminates.

$$\cdot \xrightarrow{tick} \cdot \xrightarrow{tock} \cdot$$

Stopping

The process with no transitions is STOP.

Loops

Up to now, all processes make a finite number of transitions and then terminate. Processes that can make a infinite number of transitions can be pictured by allowing loops. We accomplish loops in CCS using *recursion*.

For example, the equation

$$\mathbf{CLOCK_3} = tick.\mathbf{CLOCK_3}$$

executes the action "tick" forever.

Choice

If P and Q are processes then $P + Q$ is a process which can either behave as the process P or the process Q.

We have the following identities about choice:

$$P + (Q + R) = (P + Q) + R \quad \text{associativity}$$
$$P + Q = Q + P \quad \text{commutativity}$$
$$P + \mathbf{STOP} = P \quad \text{neutral element}$$
$$P + P = P \quad \text{idempotence}$$

Expansion Theorem

Let P and Q be processes. By expanding recursive definitions and using our existing equations for choice we can express P and Q and n-ary choices of action prefixes:

$$P = i \in \sum_{i \in I} \alpha_i .\ P_i \text{ and } Q = \sum_{j \in J} \beta_j .\ Qj$$

Then, the parallel composition can be expressed as follows:

$$P \mid Q = \sum_{i \in I} \alpha_i . (P_i \mid Q) + \sum_{j \in J} \beta_j . (P \mid Q_j) + \sum_{i \in I, j \in J, \alpha_i = \beta_j}$$

From this, many useful equations are derivable:

$$\boldsymbol{P \mid Q = Q \mid P}$$
$$\boldsymbol{P \mid (Q \mid R) = (P \mid Q) \mid R}$$
$$P \mid \mathbf{STOP} = P$$

Restriction

If P is a process and a is an action (not τ), then $P \setminus a$ is the same as the process P *except* that the actions a and $\bar{a}$ may not be executed. We have

$$(a.P) \setminus b = a.(P \setminus b) \text{ if } a \notin b, \bar{b}$$

Merge & Guards

If P is a value-passing CCS process and φ is a formula about the variables in scope, then $[\varphi]P$ is a process that executes just like P if φ holds for the current state, and like **STOP** otherwise.

We can then define an **if** statement like so:

$$\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q \equiv ([\varphi].P) + ([\neg\varphi].Q))$$

Operational Semantics

$$\frac{}{a.P \xrightarrow{a} P}\ \text{Act} \qquad \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}\ \text{Choice}_1 \qquad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}\ \text{Choice}_2$$

$$\frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q}\ \text{Par}_1 \qquad \frac{Q \xrightarrow{a} Q'}{P \mid Q \xrightarrow{a} P \mid Q'}\ \text{Par}_2 \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ \text{Sync}$$

$$\frac{P \xrightarrow{a} P' \quad a \notin \{b, \bar{b}\}}{P \setminus Q \xrightarrow{a} P' \setminus b}\ \text{Restrict}$$

Concurrency vs. Sequential Computation: Concurrency is Execution of tasks with potential overlap. Sequential is Execution of tasks one after another. Concurrency might have overlapping tasks, while sequential doesn't.

Strong Semaphore vs. Weak Semaphore: Strong Semaphore is Guarantees the order of acquiring resources. Weak Semaphore is Doesn't guarantee order. Order assurance.

Strong Fairness vs. Weak Fairness: Strong Fairness is Ensures every waiting process gets a turn. Weak Fairness is Ensures no infinite waiting, but no guarantees on order. Guarantees on task scheduling.

Synchronous vs. Asynchronous Message Passing: Synchronous is Sender waits for the receiver to acknowledge. Asynchronous is Sender doesn't wait. Waiting for acknowledgment.

Shared-variable Concurrent Programming vs. Distributed Programming: Shared-variable is Threads access shared variables. Distributed is Processes run on different machines, communicating over a network. Internal vs. external communication.

Classical Monitors vs. Monitors in Java: Classical Monitors is Encapsulation with condition variables included. Java Monitors is Uses synchronized keyword and wait/notify methods. How they're implemented and the features they provide.

If Statements in Promela vs. Java: Promela is Non-deterministic; any branch can be taken. Java is Deterministic; follows condition logic. How branches are chosen.

FLP Theorem Setting vs. Byzantine Generals Crash Failures Resilience: FLP Theorem is In an asynchronous system, consensus is impossible with even one faulty process. Byzantine Generals is System can function even with nodes that behave unpredictably. The context and assumptions about system faults.

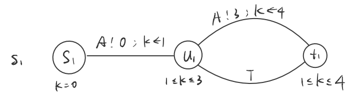
Linear-time Property: A linear-time property concerns the order or sequence of events in a system. It focuses on properties that can be described based on the order in which events occur over time.

Owicki-Gries & Interference Freedom Checks: Owicki-Gries method is used for verifying concurrent programs. Interference freedom checks are required to ensure that parallel sections of a program do not inappropriately interfere with each other, maintaining the correctness of the program.

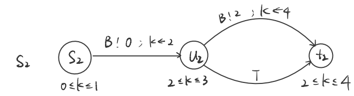
Distributed Consensus Algorithms & FLP Theorem: While the FLP theorem states that perfect consensus is impossible in asynchronous systems with even one faulty process, distributed consensus algorithms often circumvent this by making practical compromises, such as allowing occasional failures or incorporating timing assumptions.

Spin & the Bakery Algorithm: Spin is a model checker primarily designed for verifying asynchronous systems. If we try to model check the Bakery algorithm, which involves unbounded integers for ticket numbers, we'd run into limitations because Spin struggles with systems having unbounded variables.

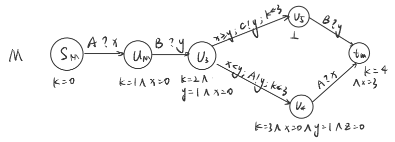
Permission-based vs. Token-based Distributed Mutual Exclusion: In a permission-based system, a process requires permission from other processes to enter a critical section. In contrast, a token-based system uses a singular token; the process holding the token can enter the critical section. The difference lies in the mechanism controlling access to the critical section.



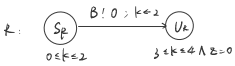
$S_1(s_1) = (k=0)$
$S_1(U_1) = (1 \le k \le 3)$
$S_1(t_1) = (1 \le k \le 4)$



$S_2(s_2) = (0 \le k \le 1)$
$S_2(U_2) = (2 \le k \le 3)$
$S_2(t_2) = (2 \le k \le 4)$



$M(S_m) = (k=0)$
$M(U_m) = (k=1 \wedge x=0)$
$M(U_3) = (k=2 \wedge x=0 \wedge y=1)$
$M(U_4) = (k=3 \wedge x=0 \wedge y=1 \wedge z=0)$
$M(U_5) = \bot$
$M(t_m) = (k=4 \wedge x=3 \wedge y=1 \wedge z=0)$



$R(S_R) = (0 \le k \le 2)$
$R(t_R) = (3 \le k \le 4 \wedge z=0)$

① $S_1(s_1) \wedge M(S_m) \rightarrow S_1(U_1) \wedge M(U_m)[x/0][k/1]$

② $S_2(S_2) \wedge M(U_m) \rightarrow S_2(U_2) \wedge M(U_3)[y/1][k/2]$

③ $R(S_R) \wedge M(U_3) \rightarrow R(t_R) \wedge M(U_4)[z/x][k/3]$

④ $R(S_R) \wedge M(U_3) \rightarrow R(t_R) \wedge M(U_5)[z/y][k/3]$.

⑤ $M(U_4) \wedge S_1(U_1) \rightarrow M(t_m) \wedge S_1(t_1)[x/3][k/4]$.

⑥ $M(U_5) \wedge S_2(U_2) \rightarrow M(t_m) \wedge S_1(t_1)[y/2][k/4]$.

So, $S_1 \| S_2 \| M \| R$ is valid network.
We finally show that.

$(k=0) \rightarrow S_1(S_1) \wedge S_2(S_2) \wedge M(S_m) \wedge R(S_R)$

And that

$S_1(t_1) \wedge S_2(t_2) \wedge M(t_m) \wedge R(t_R)$

So $S_1(t_1) \wedge S_2(t_2) \wedge M(t_m) \wedge R(t_R) \Rightarrow (x=3 \wedge y=1 \wedge z=0)$

So, $\{\top\}\ S_1 \| S_2 \| M \| R\ \{x=3 \wedge y=1 \wedge z=0\}$

A(L_1): $|S| = |S_0| \wedge m > \{s \in S/x\}$ (formally requires another aux var/flag for first time)

$A(L_2): |S| = |S_0| \wedge m > \{s \in S/m\}$

$A(L_3): |S| = |S_0| - 1 \wedge m > \{s \in S\}$

$|S| = |S_0| \wedge x > \{s \in S/x\}$

$|T| = |T_0| \wedge x < \{t \in T/n\}$

$n = x \Longrightarrow$

$|T| = |T_0| \wedge |S| = |S_0| \wedge \{s \in S/x\} < x < \{t \in T/x\}$

$A(L_1) \wedge (m \neq x) \Longrightarrow A(L_2)\ o\ m := max\ S$

$|S| = |S_0| \wedge (m \neq x) \Longrightarrow |S| = |S_0| \wedge m > \{s \in S/m\}[m/maxS]$

$|S| = |S_0| \wedge (m \neq x) \Longrightarrow |S| = |S_0| \wedge maxS > \{s \in S/maxS\}$

$A(L_2) \wedge A!m \Longrightarrow A(L_3)\ o\ S := S/m$

$|S| = |S_0| \wedge m > \{s \in S/m\} \Longrightarrow |S| = |S_0| - 1 \wedge m > \{s \in S\}[|S|/|S| - 1]$

$|S| = |S_0| \wedge m > \{s \in S/m\} \Longrightarrow |S| - 1 = |S_0| - 1 \wedge m > \{s \in S/m\}$

$A(L_3) \Longrightarrow A(L_1)\ o\ S := S \cup x$

$|S| = |S_0| - 1 \wedge m > \{s \in S\} \Longrightarrow |S| + 1 = |S_0| \wedge m > \{s \in S\}$

A(L_1'): $|T| = |T_0| \wedge n < \{t \in T/n\}$ (formally requires another aux var/flag for first time)

$A(L_2'): |T| = |T_0| + 1 \wedge n < \{t \in T/y\}$

$A(L_3'): |T| = |T_0| + 1 \wedge n < \{t \in T/n\}$

Invariant: $@L_1 \Longrightarrow (@(L_1' \vee T_2) \wedge x = n) \vee @S_2 \vee ft$ AND termination⟹n=x

Proof of invariant:

Transitions that can falsify the invariant are when you are $@L_1$ and you leave the set $(L_1' \vee T_2) \wedge x = n$ without leaving L_1 or you leave S_2 without leaving L_1.

$S_2 \rightarrow L_1' \Longrightarrow (@L_1 \Longrightarrow (@(L_1' \vee T_2) \wedge x = n) \vee @S_2 \vee ft)$ (ft would be T)

$L_1' \rightarrow L_2'(matches\ with)L_2 \rightarrow L_3 \Longrightarrow (@L_1 \Longrightarrow (@(L_1' \vee T_2) \wedge x = n) \vee @S_2 \vee ft$ (ft would be T))

$L_3' \rightarrow L_1'(matches\ with)L_3 \rightarrow L_1 \Longrightarrow @L_1 \Longrightarrow (@(L_1' \vee T_2) \wedge x = n) \vee @S_2 \vee ft[x/n]$

ft is a flag for first time reaching L_1'

(c) Prove interference freedom for $\mathcal{P}$.
The following proof obligations arise. They are all instances of the general schema

$$\mathcal{P}(p) \wedge \mathcal{Q}(q) \wedge g \Rightarrow \mathcal{P}(p) \circ f$$

$$n \geq 0 \wedge n \geq 0 \Rightarrow n \geq 0 \quad (1) \qquad n \geq 0 \wedge y \geq 0 \Rightarrow y + 1 \geq 0 \quad (2)$$

$$x \geq 0 \wedge n \geq 0 \Rightarrow x \geq 0 \quad (3) \qquad x \geq 0 \wedge y \geq 0 \Rightarrow x \geq 0 \quad (4)$$

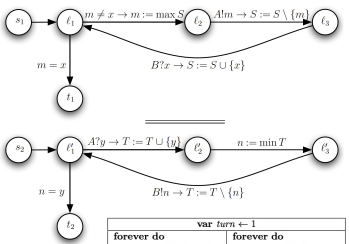
$$n > 0 \wedge n \geq 0 \Rightarrow n > 0 \quad (5) \qquad n > 0 \wedge y \geq 0 \Rightarrow y + 1 > 0 \quad (6)$$

Proof obligations 1–2 is for interference with the annotation at location p_1, obligations 3–4 for location p_2, and obligations $5 - -6$ for location p_3. Odd-numbered obligations are for the $q_1 \rightarrow q_2$ transition, and even-numbered obligations for the $q_2 \rightarrow q_3$ transition. The proof obligations are all trivial to discharge.

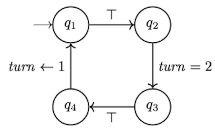
(d) Show that the precondition implies the entry annotations, and that the exit annotations imply the postcondition. The (trivial) proof obligations are as follows:

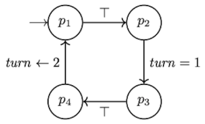
$$n = 0 \Rightarrow n \geq 0 \wedge n \geq 0 \qquad n > 0 \wedge n > 0 \Rightarrow n > 0$$

No meaningful interference is possible here. Q cannot interfere with the P's assertion turn =1 because Q never assigns any other value to turn other than 1. Same thing, mutatismutandis, holds for P's interference with Q's only assertion turn = 2.



var *turn* ← 1	
forever do	**forever do**
p1 *non-critical section*	q1 *non-critical section*
p2 **await** *turn* = 1;	q2 **await** *turn* = 2;
p3 **critical section**	q3 **critical section**
p4 *turn* ← 2	q4 *turn* ← 1





Here's the assertion network:

$$\begin{array}{ll} \mathcal{P}(p_1) = \top & \mathcal{Q}(q_1) = \top \\ \mathcal{P}(p_2) = \top & \mathcal{Q}(q_2) = \top \\ \mathcal{P}(p_3) = (turn = 1) & \mathcal{Q}(q_3) = (turn = 2) \\ \mathcal{P}(p_4) = (turn = 1) & \mathcal{Q}(q_4) = (turn = 2) \end{array}$$

Inductivity is straightforward. Mutual exclusion follows because

$$\mathcal{P}(p_3) \wedge \mathcal{Q}(q_3) \quad \Leftrightarrow \quad (turn = 1) \wedge (turn = 2) \quad \Leftrightarrow \quad \bot$$

```
monitor B
  condition direction;
  int current_direct;
  int cars; // currently o-n the bridge

  entry(dir) {
    if(dir ≠ current_direct ∧ cars > 0
      waitC(direction)
    cars++;
    current_direct = dir;
  }

  exit(dir) {
    cars--;
    if(cars==0)
      while(¬emptyC(direction))
        signalC(direction)
  }

while(true)
  gather();
  deposit ⇐ ();
  ack ⇒ x;
  if(x = ⊤)
    wake_bear ⇐()

=== Bear ===
while(true)
  wake_bear ⇒ _;
  x = T;
  while(x)
    take ⇒ x; // transmit a boolean,
     //T if there's more honey

==Pot ===
int honey=0  (# units of honey)
===========
while(true)
  while(honey < k)
    deposit ⇒ ();
    ack ⇐ (honey = k);
    honey++;
  while(0 < honey)
    honey--;
    take ⇐ (honey ≠ 0);
```

$S_1 \xrightarrow{s=1;\ s--} L_1 \xrightarrow{a=x} L_2 \xrightarrow{x=a+1} L_3 \xrightarrow{s++} T_3$

$A(L_1) = s=0 \wedge !$ (other process L)
$A(L_2) = s=0 \wedge !$ (other process L)
$A(L_3) = s=0 \wedge !$ (other process L)
$A(T_3) = n=1 +$ (num other process T)

$S_1 \rightarrow L_1\ g \wedge s=1 \wedge s=0 \wedge !$ (other process L)
$\Rightarrow s=0 \wedge !$ (other process L) $[s/s-1]$

$F \Rightarrow \ldots$

prioritisation

Synchronous Transition Diagrams

A synchronous transition diagram is a parallel composition $P_1 \parallel \ldots \parallel P_n$ of some (sequential) transition diagrams $P_1, \ldots, P_n$ called processes.

The processes P_i:

- Do not share variables
- Communicate along unidirectional channels $C, D, \ldots$ connecting at most 2 different processes by way of
  - output statements $C \Leftarrow e$ for sending the value of expression e along channel C
  - input statements $C \Rightarrow x$ for receiving a value along channel C into variable x

For shared variable concurrency, labels $b; f$ where b is a Boolean condition and f a state transformation sufficed. Now, we call such transitions *internal*.

We extend this notation to message passing by allowing the guard to be combined with an input or an output statement.

Closed Product

Given $P_i = (L_i, T_i, s_i, t_i)$ for $1 \leq i \leq n$ with disjoint local variable sets, define their *closed product* as $P = (L, T, s, t)$ such that $L = L_1 \times \cdots \times L_n$, $s = \langle s_1, \ldots, s_n \rangle$, $t = \langle t_1, \ldots, t_n \rangle$ and $\ell \xrightarrow{a} \ell' \in T$ iff either

1. $\ell = \langle \ell_1, \ldots, \ell_i, \ldots, \ell_n \rangle$, $\ell' = \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n \rangle$, $\ell \xrightarrow{a} \ell_i' \in T_i$ an internal transition or
2. $\ell = \langle \ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots, \ell_n \rangle$, $\ell' = \langle \ell_1, \ldots, \ell_i', \ldots, \ell_j', \ldots, \ell_n \rangle$, $i \neq j$ with $\ell_i \xrightarrow{b; C \Leftarrow e; f} \ell_i' \in T_i$ and $\ell_j \xrightarrow{b'; C \Rightarrow x; g} \ell_j' \in T_j$ and a is $b \wedge b'; f \circ g \circ [\![x \leftarrow e]\!]$

Verification

To show that $\{\phi\}P_1 \parallel \ldots \parallel P_n\{\psi\}$ is valid, we could simply prove $\{\phi\}\ P\ \{\psi\}$ for P being the closed product of the P_i. This can be done using the same method as for ordinary transition diagrams because there are no I/O transitions left in P.

As with the standard product construction for shared-variable concurrency, the closed product construction leads to a number of verification conditions exponential in the number of processes. Therefore, we are looking for an equivalent of the Owicki/Gries method for synchronous message passing.

A Simplistic Method

For each location ℓ in some L_i, find a local predicate Q_ℓ, only depending on P_i's local variables.

1. Prove that, for all i, the local verification conditions hold, i.e. $\models Q_\ell \wedge b \rightarrow Q_{\ell'} \circ f$ for each $\ell \xrightarrow{b; f} \ell' \in T_i$
2. For all $i \neq j$ and matching pairs of I/O transitions $\ell_i \xrightarrow{b; C \Leftarrow e; f} \ell_i' \in T_i$ and $\ell_j \xrightarrow{b'; C \Rightarrow x; g} \ell_j' \in T_j$ show that $\models Q_{\ell_i} \wedge Q_{\ell_j} \wedge b \wedge b' \implies (Q_{\ell_i'} \wedge Q_{\ell_j'}) \circ f \circ g \circ [\![x \leftarrow e]\!]$
3. Prove $\models \phi \implies Q_{s_1} \wedge \cdots \wedge Q_{s_n}$ and $\models Q_{t_1} \wedge \cdots \wedge Q_{t_n} \implies \psi$

The simplistic method is sound but not complete. It generates proof obligations for all syntactically-matching I/O transition pairs, regardless of whether these pairs can actually be matched semantically (in an execution).

Remedy 1: Adding Shared Auxiliary Variables

Use shared auxiliary variables to relate locations in processes by expressing that certain combinations will not occur during execution. Only output transitions need to be augmented with assignments to these shared auxiliary variables.

Pro: easy.

Con: re-introduces interference freedom tests for matching pairs $\ell_i \xrightarrow{b_i; C \Leftarrow e; f_i} \ell_i' \in T_i$ and $\ell_j \xrightarrow{b_j; C \Rightarrow x; f_j} \ell_j' \in T_j$ and location ℓ_m of process P_m, $m \neq i, j$:

$$\models Q_{\ell_i} \wedge Q_{\ell_j} \wedge Q_{\ell_m} \wedge b_i \wedge b_j \implies Q_{\ell_m} \circ f_i \circ f_j \circ [\![x \leftarrow e]\!]$$

This method is due to Levin & Gries.

Remedy 2: Local Auxiliary Variables + Invariant

Use only local auxiliary variables and a global communication invariant I to relate values of local auxiliary variables in the various processes.

Pro: no interference freedom tests.

Con: more complicated proof obligation for communication steps:

$$\models Q_{\ell_i} \wedge Q_{\ell_j} \wedge b \wedge b' \wedge I \implies (Q_{\ell_i'} \wedge Q_{\ell_j'} \wedge I) \circ f \circ g \circ [\![x \leftarrow e]\!]$$

This method is the AFR-method.

2. Here are the assertion networks:

$$\begin{array}{ll} \mathcal{P}(p_1) = n \geq 0 & \mathcal{Q}(q_1) = n \geq 0 \\ \mathcal{P}(p_2) = x \geq 0 & \mathcal{Q}(q_2) = y \geq 0 \\ \mathcal{P}(p_3) = n > 0 & \mathcal{Q}(q_3) = n > 0 \end{array}$$

Here, we need to resist the temptation of annotating the entry locations with the precondition $n = 0$. This assertion would not be robust to interference, hence we weaken = to $\geq$. n can change beneath our feet, but at least it can't shrink beneath our feet.

By the Owicki-Gries method, we have the following tasks in front of us to prove the Hoare triple $\{n = 0\}\ P \| Q\ \{n > 0\}$.

(a) Prove $\mathcal{P}$ inductive.
The following proof obligations arise. They are the instantiations of the general schema $\mathcal{P}(\ell_i) \wedge g \Rightarrow \mathcal{P}(\ell_j) \circ f$, for each of the two transitions in the diagram.

$$n \geq 0 \wedge \top \Rightarrow n \geq 0 \quad (1) \qquad x \geq 0 \wedge \top \Rightarrow x + 1 > 0 \quad (2)$$

None of them are very exciting. (1) is vacuously true, and (2) follows by elementary arithmetic. This is a good sign—usually, discharging the proof obligations is the easy part of the proof. The tricky part is finding the annotations that make it easy.

(b) Prove $\mathcal{Q}$ inductive.
The following proof obligations arise.

$$n \geq 0 \wedge \top \Rightarrow n \geq 0 \quad (1) \qquad y \geq 0 \wedge \top \Rightarrow y + 1 > 0 \quad (2)$$

These are identical to the proof obligations for $\mathcal{P}$, up to variable renaming.