**Z Language**

Based on set theory

| **Logical Operators** | |
| --- | --- |
| ¬ | negation |
| ∧ | conjunction (AND) |
| ∨ | disjunction (OR) |
| ⇒ | implication |
| ⇔ | equivalence |
| **Sets** | |
|  | * set of elements of X for which the predicate P is true. * E.g. ℕ is the set |
| (X)  (power set) | * the set of all subsets of X * E.g. P{a, b} = {∅, {a}, {b}, {a, b}} |
| #X | * X is any finite set → #X is cardinality in X * E.g. #P(A) = 2#A for any finite set A |
|  | * Set of all ordered pairs (a, b) with |
| Types | * + The set of all elements that result from evaluating expr(x) for all x of type T for which pred(x) holds * E.g. : the set of all squares of prime numbers * (x, y): A x B == x: A; y: B == x, y: A * Strongly typed language      * : for declaration, ∈ for predicate * ● means “such that” * … means some predicate |
| **Relations** | |
|  | * is a subset of * domR = * ranR = |
| Domain & Range restrictions | * S ◁ R: the subset of R with a in S * R ▷ T: the subset of R with b in T * Both are true        * (a, b) is in R → a has\_sibling b → female a has\_sibling b → a is a sister of b * a has\_sibling female b → a has sister b |
| Domain & Range subtraction | * (a, b) is in R but → a is a sibling of b where a is not female → a is brother of b |
| Relational image | * Suppose * → subset of ran R with the domain restricted to a subset of A |
|  |  |
| Inverse |  |
| Relational composition |  |
| **Functions** | |
| (Partial) Functions | * partial function f: is a subset of A X B such that for each a ∈ A, there is **at most one** b ∈ B with (a, b) ∈ f * Same meaning as Math |
| Function overriding | * If input in both A and B → output depends on g, else depends on f * i.e. * g is the overriding function, f is being overridden      * 1. domain remains the same   2. if a is part of domain g then overridden function should return the same value as g(a) |
| **Sequences** | |
| s : seq A | * A function where dom s = 1...n for some natural number n * E.g. <b, a, c, b> denotes sequence (function) * Maps an ordered set to tuples of (k, v) where k is the order that v appears in the sequence      * seq A includes empty set → no domain will be vacuously true |
| Concatenation | * same sequence appended |
| Head & Tail | * head<c,b,b> = c * tail<c, b, b> = <b, b> |
| **Schema** | |
| State schema | * State schema specifies a ‘snapshot’ of a system → static view * variables declared at the top part * predicate at the bottom part |
| Operation schema | * Taking an instance of the state schema and produce a new instance * Express the relationship between the initial state and new state as a predicate * Initial values: unprimed, e.g. x * New values: primed, e.g. x’ |
| Schema inclusion | * Including a schema in the declaration part of another schema * Declaration added to the new schema and predicated conjoined to the predicate of the new schema   → |
| ? | input |
| ! | output |
| Conjunction |  |
| Disjunction |  |
| Composition |  |
| Piping |  |
| **Buffer (FIFO) Case Study** | |
| [MSG] | Set of all possible messages that could be transmitted |
| max: N | Constant max number of messages (capacity) |
| * E.g. suppose MSG = {m1, m2, m3} and max = 4 * items = <m1, m2> is an (valid) instance * items = <m3, m1, m1, m2, m2> is not an instance as size > max * Operations are atomic | |
| Initial state | == |
| Join | * items: sequence of messages before Join * items’: sequence of messages after Join * There is implicit AND (⋀) between each line * #items <= max && #items’ <= max => buffer need to be valid before and after Join * #items < max => for Join to be possible, buffer must not be full * items’ = items ဂ <msg?> => input message is appended to the sequence of messages already in the buffer * E.g. MSG = {m1, m2, m3}, max = 4, items = <m1, m2, m1> and msg? = m3   + After Join: items = <m1, m2, m1, m3>   With schema inclusion |
| Leave | * items != ∅ => Leave is valid only if buffer is not empty * items = <msg!> ဂ items’ => output message is taken from the head of the sequence of messages in the buffer, leaving just the tail of the sequence in the buffer |
| JoinLeave (composition) | * Atomic operation with the effect of a Join followed by Leave      * Last predicate: there exists a middle state (that is a possible sequence of MSG) where middle = initial concat with input and middle = output concat final state * If use conjunction instead of composition → give false all the time |
| LeaveDuplicated | * Output variables of Leave and the input variables of Duplicate with identical bases (i.e. ignoring the decorations ‘?’ and ‘!’ respectively) have their values identified and hidden in Leave >> Duplicate |
| **Slow Buffer** | |
|  | |
| SlowJoin | == |
| SlowLeave |  |
| Tick |  |
| **Recorded Buffer** | |
|  | |
| RecordedJoin |  |
| RecordedLeave |  |
|  |  |

**CSP**

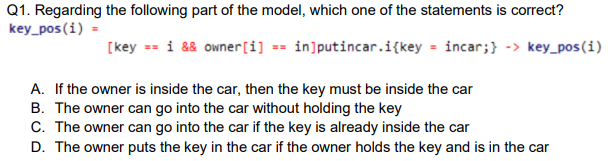
| **Denotational semantics (Mathematical definitions of CSP)** | | |
| --- | --- | --- |
| Process | * Defined by its behaviours (what it can do) * Interaction between a system and its environment, i.e. external (visible) behaviour * Denoted by UPPER CASE identifiers   + E.g. X, Y, Z ⇒ also denotes sets of events | |
| Event | * A process engages in events * Each event is an atomic action * E.g. events for a vending machine   + coin: insert a coin   + choc: extracts a chocolate * Denoted by lower case identifier   + E.g. x, y, z | |
| Alphabet (of process) | * The set of events that a process can possibly engage in * E.g. {coin, choc} * Denoted by αP where P is a process | |
| Trace | * A finite sequence of events * A (deterministic) process is specified by the set of traces denoting its possible behaviour * E.g. traces(a → b → STOP) = {<>, <a>, <a, b>} * E.g. traces of vending machine   + <>, <coin>, <coin, choc>, <coin, choc, coin>... * Any execution of the process will be one of these sequences. * If s ⌒ t is a trace of a process, then s is a trace of process. * i.e. the set of traces is prefix closed * Denoted by traces(P) where P is a process | |
| Trace notation | * If A is a set of event, seq A denotes the set of all finite sequences of events (i.e. set of all traces) from A * If s, t : seq A, s ⌒ t is the concatenation of s with t       E.g.    Traces of processes    Example 3:   * traces of x of type B being passed into process P * So t is an element of all possible sequences such that t is empty or first element of trace t is of type B and tail is first element after going through process P * Since t can be any possible sequence, add all (i.e. union) x of type B that fulfils traces(x → P(x)   Traces of recursive process | |
| Prefix | * a → P: A process which may participate in event a then act according to process description P * Event a is initially enabled by the process and occurs as soon as it is requested by its environment, all other events are refused initially * Event a is referred to as the guard of the process * E.g.   + VMU = coin → STOP   + SHORTLIFE = (beat → (beat → STOP)) = beat → beat → STOP   + VMS (S for single) = coin → choc → STOP | |
| Sequential composition | P; Q   * A distinguished event, e, used to represent a detect process termination * Process acts as P until P terminates by communicating e and then proceeds to act as Q * Termination signal is hidden from the process environment → terminal occurs as soon as enabled by P * Process to denote termination: SKIP | |
| Parallel composition | Synchronisation: P | [X] | Q   * Parallel composition of processes P and Q, synchronised on event set X ⇒ X is the barrier * No event from X may occurs unless enabled jointly by both P and Q * When events from X occur, they occur in both P and Q simultaneously → synchronisations * Events not in X occur in P/Q in non-deterministic order * E.g. (a → P) | [a] | (c → a → Q)   + c must occur first since its the only event that can enable a | |
| Asynchronous: P ||| Q   * Both P and Q execute concurrently w/o any synchronisations | |
| **Concurrency** | | |
| P || Q   * P & Q must cooperate on all common actions * to cooperate/synchronise means that both process must be at the same event (common events as defined by both alphabet implicitly or explicitly) at the same time * αP != αQ     Deadlock: <init → coin → bisc>   1. Both at coin → can only start on coin because coin must be in sync 2. VMC moved to bisc, CHOCLOV cannot move 3. VMC at coin, CHOCLOV still at choc ⇒ both waiting for diff events ⇒ cannot synchronise      * Enforce all events to be synchronised (even if bisc is not in CHOCLOV) * No deadlock after adding the alphabets   1. Both at coin   2. Since bisc is defined to be synchronised, bisc cannot happen as it does not exist in CHOCLOV      * Deadlock: <init → on → coin → bisc>   + Both at on and coin   + Since bisc is not in ɑVMH (does not need to be synchronised), CUST can move on to next event CUST and goes back to on   + VMH waits on choc while CUST waits on on ⇒ deadlock * on → curse…   + No deadlock as curse in not in the ɑVMH, so CHOCLOV can move on to coin → subsequent sequence will be the same as VMH      * Deadlock: <init → on → coin> or <init → on → coin → bisc>   + VMH waits on coin, CUST waits on curse      * Note: <up, rest> ∉ traces(SLOTREK)   + If first event for SLOCLIMB is up, it then waits on rest, but for SLOWALK to wait on rest, left/right has to occur   + <left/right, up, rest> / <up, left/right, rest> will be valid | | |
| Processes with disjoint alphabets (interleaving) | * No common events in the alphabets, TREK can have sequences in any order of L/R/U/D | |
| Laws for concurency |  | |
| Traces for concurrency | * If t ∈ seq A, then t ↾ B denotes the trace of t restricted to events in B (projection to elements in B)   + E.g. <c, a, d, b, a, c> ↾ {b, c} = <c, b, c> order is preserved     E.g. P = SLOCLIMB, Q = SLOWALK  <up, rest> ∉ traces(SLOWALK || SLOCLIMB)  <up, rest> ↾ {left, right, rest} = <rest> ∉ traces(SLOWALK)  If   * GREEDY = (choc -> GREEDY | bisc -> GREEDY) * VMC = coin -> (choc -> VMC | bisc -> VMC)   then (assume GREEDY and VMS have the same alphabet)   * GREEDY || VMC = STOP * Need to sync on coin but theres no coin in GREEDY   DT = getboard->playchess->DT [] getracket->playtennis->DT;  VP = getpieces->playchess->VP [] getball->playtennis->VP;  DP || VP   * Deadlock: <getpieces → getracket> * common events = {playcheess, playtennis} * one waits on playchess and another at playtennis   UN = getpieces -> getboard -> UN [] getracket->getball->UN;  playtogether = DT || VP || UN;   * No deadlock: force VP to getpieces and DT to getboard (same for racket and ball) | |
| Starvation (Fairness) | * Deadlock: <p.sits, q.sits, p.gold.up, q.silver.up>   + P and Q are contending for the same resource * Resolve with FOOT = p.sits → p.leaves → FOOT | q.sits → q.leaves → FOOT   + NEWCOLLEGE = COLLEGE || FOOT * NEWCOLLEGE may lead to starvation if p is always chosen * Replace FOOT with FAIRFOOT = p.sits → p.leaves → q.sits → q.leaves → FAIRFOOT   + Fair but restrictive | |
| Choice | External choice: a → P ◻ b → Q   * Allows a process to act according to events requested by its environment * a → P ◻ b → Q begins with **both a and b enabled,** subsequent behaviour depends on whether a or b is selected   + Selecting a leads to P   + Selecting b leads to Q * E.g. user makes the decision for vending machine (VM) | |
| Internal choice: a → P ⊓ b → Q   * Represents a variation in behaviour determined by the internal state of the process * a → P ⊓ b → Q begins with **either a or b or both enabled** as it wishes, but must act subsequently according to the event that actually occurred * Environment CANNOT affect the internal choice * E.g. VM makes the decision for user | |
| Channel (FIFO) | * A collection of events of the form c.n   + Prefix c: channel name   + Collection of suffixes: values of the channel * When an event c.n occurs → value n is communicated on channel c * Input: value of a communication is determined by the environment (external choice) * Output: value of a communication is determined by the internal state of the process (internal choice) * To describe behaviour over a range of allowed input   + c?n : N → P(n) == ◻ n : N ● c.n → P(n)   + c!n : N → P(n) == ⊓ n : N ● c.n → P(n) * E.g.   + COPYBIT = (in.0 → out.0 → COPYBIT) ◻ (in.1 → out.1 → COPYBIT)   + αCOPYBIT = {in.0, out.0, in.1, out.1} | |
| **Synchronous** | **Asynchronous** |
| * Buffer size 0 * Communicate through a pairwise handshaking mechanism * A process ch!exp → P which is ready to perform an output through ch will be enabled if another process ch?m → P(m) is ready to perform an input through ch simultaneously, and vice versa | * Predefined buffer size * process ch!exp → P evaluates the exp and puts the value of exp into the tail of the respective buffer and behaves as P; process ch?m → P(m) gets the top/front element in the respective buffer, assigns it to m and then behaves as P |
| Interrupt process | P1 ▽ e → P2   * Process acts as P1 until the first occurrence of the interrupt event e, then control is passed to P2 | |
| Recursion | * Give a finite representation of non-terminating processes * E.g. μP ● a?n : ℕ → b!f(n) → P describes a process which repeatedly inputs a natural number n on channel a, then calculates some function f of the input and outputs the result on channel b | |
| State parameters | * Represents a (possibly) finite family of definitions, one for each possible value of n | |
| **Summary** | | |
| Primitives | * STOP communicates nothing * SKIP terminates process | |
| Event prefixing | a → P   * a is an action | |
| Channel | * Output: ch!exp → P * Input: ch?m → P(m) | |
| Data operation prefixing | e{prog} → P   * prog is executed atomically with the occurrence of e * e is non-communicating event * prog contains updates to shared variables | |
| State guard | [b]P   * Process [b]P waits until condition b becomes true and then behaves as P | |
| Conditional choice | if (b) {P} else {Q} | |
| Choices | External: P ◻ Q   * resolved only by occurrence of visible event   Internal: P ⊓ Q   * Resolved non-deterministically   General: P [] Q   * Resolved by any event | |
| Sequential composition | P; Q   * Behaves as P until P terminates, then behaves as Q | |
| hiding | P \ X   * Hides all occurrences of the set of actions in X | |
| Parallel | P || Q   * P, Q run in parallel and synchronise on common communication events | |
| Interleaving | P ||| Q   * P, Q run independently (except for communications thorugh synchronous channels and shared variables) | |
| Interrupt | P △ Q   * Process behaves as P until the first occurrence of a visible event from Q | |
| Process reference | * A process expression may be given a name for referencing * Recursion is supported by process referencing | |
| **Keyless Car System** | | |
| * keylockinside is INVALID   Tutorial  Your tasks is to extend the current system with two more operations:   1. window(i) which can open (totally or partially) the car door window 2. throwKey(i) which captures that the key-fob can be throw in/out of the car.     #define **keylockinside** (key == incar && door == lock && owner[0] != in && owner[1] != in);  // can key be locked inside  #assert car reaches keylockinside;   * keylockinside is VALID * <init -> towards.0 -> unlockopen.0 -> getin.0 -> turnon.0 -> open\_window.0 -> goout.0 -> close.0 -> outsidelock.0 -> throw\_key\_in.0> * However, if throw\_key\_in is designed to change door state to “open” when key is in the car → key will not be locked inside | | |

| **Operational Semantics** | |
| --- | --- |
| * used in PAT as execution firing rules   e.g. | |
| Primitives | * STOP * SKIP |
| Prefixing |  |
| Choices | * External (a is a visible event)        * Internal (τ is a silent invisible event) |
| Sequential composition | P; Q   * Let ✓ be a distinguished event denoting termination * P executes first and Q starts only when P has terminated |
| Interrupt | P ▽ Q   * Whenever an event is engaged by Q, P is interrupted and Q takes control      * [interrupt1]: no interrupt occurred → P transits to P’ * [interrupt2]: interrupt occurred → process becomes Q’ |
| Interleaving | P ||| Q   * P & Q behaves independently (except termination) * Assume a != termination |
| Synchronisation | P | [X] | Q   * Event from X must occur jointly by P & Q → when it occurs, will occur in both P & Q |
| **Labelled Transition System** | |
| E.g. Dining philosophers       * State 4 is deadlock | |
| **Denotational Semantics in Unified Theory of Programming (UTP)** | |
| * Use relations to define denotational semantics * Alphabet: a set of observational variable names * Signature: syntax for denoting objects of the theory * Healthiness conditions: identifying valid predicates | |
| Failure | A pair (s, X) consisting of a trace s, and a refusal set X which identifies the event that a process may refuse once it has executed the trace s.  E.g.   * failures((a → STOP) ◻ (b → STOP)) = {(< >, ∅), (<a>, {a, b}), (<b>, {a, b})}   + After executing a → terminate → refuse both a&b subsequently   + After executing b → terminate → refuse both a&b subsequently * failures((a → STOP) ⊓ (b → STOP)) = {(< >, {a}), (< >, {b}), (<a>, {a, b}), (<b>, {a, b})}   + <> → process only chooses one to execute? → refuse the event in the other choice |
| Divergence | * A process is divergence if it performs an endless series of hidden actions |

| **Timed Extension of CSP** | |
| --- | --- |
| Semantics | A concrete system configuration is a pair (V, P) where V is a variable valuation function and P is a process. |
| Delay | Wait[d]   * delays the system execution for a period of t time units then terminates      * If t <= d, decrement waiting time (d) by t, else → has waited for sufficient time (delay) → can move to next event/terminate |
| Timeout | P timeout[d] Q   * passes control to process Q if no event has occurred in process P before d time units have elapsed. * E.g. if process (a → P) timeout[d] Q engages in event a before d time units have elapsed, the process is transformed to P;   + If a hasn't occurred at time d, the process transforms to Q      * [to1]: P’ happened before timeout → P transits to P’ * [to2]: * [to3]: time passed but have not timed out yet → reduce timeout * [to4]: timeout → control passed to Q |
| Timed interrupt | P interrupt[d] Q   * behaves as P until d time units elapse and then switches to Q. * E.g. process (a → b → c → ...) interrupt[d] Q may engage in event a, b, c, ... as long as d time units haven't elapsed. Once t time units have elapsed, then the process transforms to Q      * [it1]: some event x happened that caused V to transit to V’ immediately??? * [it2]: P happens during the specified time [it2]. * [it3]: After the specified time, P is interrupted and control is given to Q |
| Deadline | P deadline[d]   * P is constrained to terminate within d time units.      * [dl1]: event transits before deadline * [dl2]: time t passed, decrement deadline by t |
|  |  |
| **Fischer’s Mutual Exclusion** | |
| #define N 4;    #define Delta 3;  #define Epsilon 4;  #define Idle -1;    var x = Idle;  var counter;  P(i) = ifb(x == Idle) {  ((update.i{x = i} -> Wait[Epsilon]) within[Delta]);  if (x == i) {  cs.i{counter++} -> exit.i{counter--; x=Idle} -> P(i)  } else {  P(i)  };    FischersProtocol = ||| i:{0..N-1}@**P**(i);    #assert FischersProtocol deadlockfree;  *//mutual exclusion testing*  #define MutualExclutsionFail counter > 1;  #assert FischersProtocol reaches MutualExclutsionFail; | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Exam Questions**

CSP Paper:



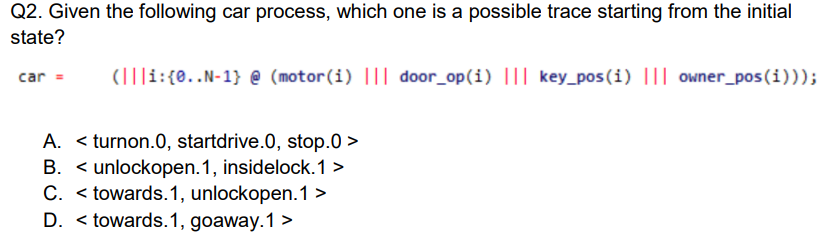
A is incorrect → key must be with owner for owner to getin.i

B is incorrect → if owner getin.i then door must be open; door is open if:

1. unlockopen.i → key must be held by owner
2. justopen.i → door unlocked already
   1. close.i

C is incorrect → if key in car means putincar.i so owner must already be in for process putincar to happen

D is correct → basically describing key\_pos first guard process

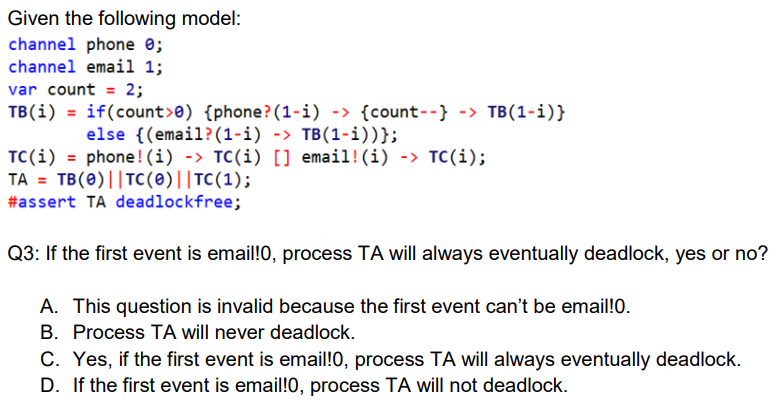


A → cannot start with turnon.0 as owner must be in car first so getin.0 must happen first

Key initially with owner 0 first so any process that calls key\_pos(1) cannot happen

B and C not possible

D → correct since towards car and goaway does not require key == 1



A is incorrect TC(0) → email!0 → TC(0) puts 0 into email channel

phone channel is synchronous; email channel is async

email!0 puts a value 0 into buffer email

TB(0) waiting for phone?1

TC(1) phone!1 → count = 1 → TB(1)

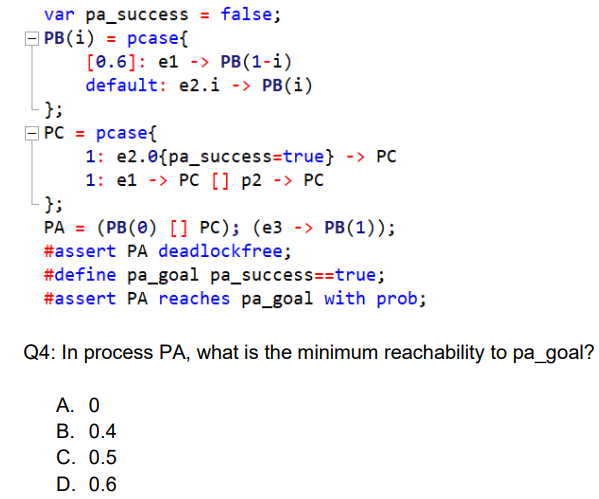
TB(1) waiting for phone?0

TC(0) phone!0 → count = 0 → TB(0)

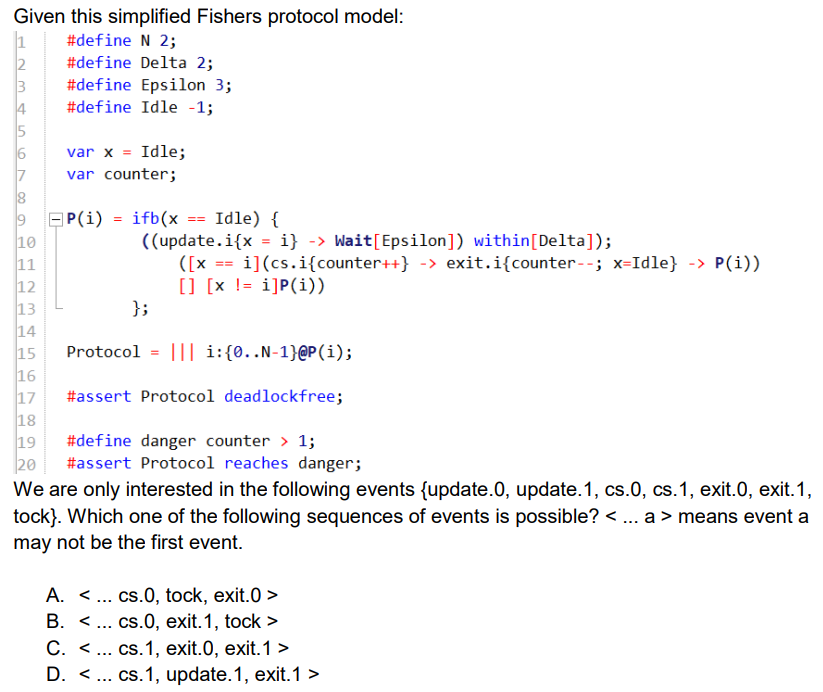
TB(0) → email?1

TC(1) → email!1

Answer: C → email buffer space taken up by email!0 first event so TC(1) cannot run email!1 to fulfil TB(0) email?1



A. if choose PB(0) then the probability of running e2.0 is never true



A is possible

B is not possible since cs.0 means process 0 in critical section so there should not be cs.1 at the same time

C same as B

D is not possible → must update then enter cs so <...update.1, cs.1, exit.1>

To continue after revision

* Intro to z p43

## CSP Basics

Communication sequential processes

| Process |
| --- |
| A process is determined by what it can do  => The interaction btw a system and the environment |
| Upper case identifiers denotes Process  X, Y are variables denoting Process |

| Event |
| --- |
| A process engages in events.  Each event is an atomic action |
| Lower case identifiers denotes events  x, y, z are variables denoting Event  A,B,C denotes sets of events |

| Alphabet |
| --- |
| The set of events that a process can possibly engage in |
| i.e. Vending machine - insert / extract  alphabet -> {insert, extract} |
| If P is a process,  ⍺P denotes the alphabet of P |

| Traces |
| --- |
| A finite sequence of events  A deterministic process is specified by the set of traces denoting its possible behavior  i.e. <>  <coin>  <coin, choc>  <coin, choc, coin> … |
| If P is a process,  traces(P) denotes the set of traces of P |
|  |
|  |

| Prefix |
| --- |
|  |
| The guard of the process |
|  |