Theory of concurrency

Lecture 12

Sequential Processes

Sequential Processes: **Introduction**

- Finished processes
  - the process *STOP*
    - results from a deadlock or other design error.
  - a successfully terminated process
    - has already accomplished everything that it was designed to do.
- Successful termination is a special event, denoted by the symbol ✓
- A *sequential process* has
  - has ✓ in its alphabet
    - ✓ can only be the last event in which the process engages.
    - ✓ cannot be an alternative in the choice construct
      - $(x : B \rightarrow P(x))$ is invalid if $✓ \in B$
- $SKIP_A$ is a process which does nothing but terminate successfully
  - $\alpha SKIP_A = A \cup \{✓\}$
    - we shall frequently omit the subscript alphabet.

Sequential Processes: **Introduction**

X1. A vending machine serves *only one* customer with chocolate or toffee and then terminate successfully:

$$VMONE = (coin \rightarrow (choc \rightarrow SKIP \mid toffee \rightarrow SKIP))$$

- A complex task can be split into two subtasks
  - one of which must be *completed successfully* before the other begins.
- The *sequential composition* is
  - $P \,;\, Q$
    - P and Q are sequential processes with the same alphabet.
  - a process which first behaves like P;
    - when P terminates successfully, $(P \,;\, Q)$ continues by behaving as Q.
    - If P never terminates successfully, neither does $(P \,;\, Q)$.

$VMONE = (coin \rightarrow (choc \rightarrow SKIP \mid toffee \rightarrow SKIP))$

Sequential Processes: **Introduction**

X2 A vending machine serves exactly two customers, one after the other:

$$VMTWO = VMONE\ ;\ VMONE$$

- A *loop* process which repeats similar actions as often as required
  - can be defined as a special case of recursion:

$$^{*}P = \mu\ X \bullet (P\ ;\ X) = P\ ;\ P\ ;\ P\ ;...$$

- $\alpha(^{*}P) = \alpha P - \{\checkmark\}$
  - such a loop will never terminate successfully.

X3 A vending machine which serves any number of customers is identical to *VMCT*:

$$VMCT = {}^{*}VMONE$$

$VMCT = \mu X \bullet coin \rightarrow (choc \rightarrow X \mid toffee \rightarrow X)$

Sequential Processes: **Introduction**

- A *sentence* of a process P is a sequence of events after which
  - P terminates successfully.
- The *language* accepted by P is
  - the set of all such sentences.

- The notations introduced for describing sequential processes may also be used to
  - *define the grammar* of a simple language.

Sequential Processes: Introduction

X4 A sentence of Pidgingol consists of a noun clause followed by a predicate.

- A predicate is a verb followed by a noun clause.
- A verb is either *bites* or *scratches*.
  - $\alpha PIDGINGOL = \{a, the, cat, dog, bites, scratches\}$

$$
\begin{aligned}
&PIDGINGOL = NOUNCLAUSE \,;\, PREDICATE \\
&PREDICATE = VERB \,;\, NOUNCLAUSE \\
&VERB = (bites \rightarrow SKIP \mid scratches \rightarrow SKIP) \\
&NOUNCLAUSE = ARTICLE \,;\, NOUN \\
&ARTICLE = (a \rightarrow SKIP \mid the \rightarrow SKIP) \\
&NOUN = (cat \rightarrow SKIP \mid dog \rightarrow SKIP)
\end{aligned}
$$

- Example sentences of Pidgingol are
  - *the cat scratches a dog*
  - *a dog bites the cat*

Sequential Processes: **Introduction**

- To describe languages with an unbounded number of sentences,
  - it is necessary to use some kind of iteration or recursion.

X5. A noun clause which may contain any number of adjectives *furry* or *prize*

$$NOUNCLAUSE = ARTICLE \,;\, \mu X \bullet (furry \rightarrow X \mid prize \rightarrow X \mid cat \rightarrow SKIP \mid dog \rightarrow SKIP)$$

- Examples of a noun clause are
  - *the furry furry prize dog*
  - *a dog*

$PIDGINGOL = NOUNCLAUSE \,;\, PREDICATE$
$PREDICATE = VERB \,;\, NOUNCLAUSE$
$VERB = (bites \rightarrow SKIP \mid scratches \rightarrow SKIP)$
$NOUNCLAUSE = ARTICLE \,;\, NOUN$
$ARTICLE = (a \rightarrow SKIP \mid the \rightarrow SKIP)$
$NOUN = (cat \rightarrow SKIP \mid dog \rightarrow SKIP)$

Sequential Processes: **Introduction**

X6 A process which accepts any number of as followed by b and then the same number of cs, after which it terminates successfully

$$A^nBC^n = \mu\ X \bullet (b \rightarrow SKIP \mid a \rightarrow (X\,;\,(c \rightarrow SKIP)))$$

- If a b is accepted first, the process terminates;
  - no as and no cs are accepted, so their numbers are the same.
- If the second branch is taken,
  - the accepted sentence starts with a and ends with c, and
  - between these is the sentence accepted by the recursive call on the process X.
    - If we assume that the recursive call accepts an equal number of as and cs,
      - then so will the non-recursive call on A^nBC^n,
        - since it accepts just one more a at the start and one more c at the end.
- Here sequential composition, used in conjunction with recursion,
  - can define a machine with an infinite number of states.

$A^nBC^n = \mu X \bullet (b \to SKIP \mid a \to (X ; (c \to SKIP)))$

$C^nDE^n = \mu X \bullet (d \to SKIP \mid c \to (X ; (e \to SKIP)))$

Sequential Processes: **Introduction**

X7 A process which first behaves like A^nBC^n, but the accepts d followed by the same number of es

$$A^nBC^nDE^n = ((A^nBC^n) ; d \to SKIP) \parallel C^nDE^n$$

- $C^nDE^n = f(A^nBC^n)$ for f which maps a to c, b to d, and c to e.

- The process on the left of the ‖ is responsible for
  - ensuring an equal number of as and cs (separated by a b).
  - It will not allow a d until the proper number of cs have arrived.
  - The es (which are not in its alphabet) are ignored.

- The process on the right of ‖ is responsible for
  - ensuring an equal number of es as cs.
  - It ignores the as and the b, which are not in its alphabet.

- The pair of processes terminate together when
  - they have both completed their tasks.

$$CT_0 = (up \rightarrow CT_1 \mid around \rightarrow CT_0)$$
$$CT_{n+1} = (up \rightarrow CT_{n+2} \mid down \rightarrow CT_n)$$

Sequential Processes: **Introduction**

X8. A process which accepts any interleaving of *down*s and *up*s, except that it terminates successfully on the first occasion that the number of *down*s exceeds the number of *up*s

$$POS = (down \rightarrow SKIP \mid up \rightarrow (POS\ ;\ POS))$$

- If the first symbol is *down*, the task of *POS* is accomplished.
- But if the first symbol is *up*, it is necessary to accept *two* more *down*s than *up*s.
  - First to accept one more *down* than *up*; and
  - then again to accept one more *down* than *up*.
  - Two successive recursive calls on *POS* are needed, one after the other.

X9. The process C_0 behaves like CT_0:

$$\begin{aligned} C_0 &= (around \rightarrow C_0 \mid up \rightarrow C_1) \\ C_{n+1} &= POS\ ;\ C_n \\ &= \underbrace{POS\ ;\ \ldots\ POS;\ POS\ ;\ POS}_{n \text{ times}}\ ;\ C_0 \qquad \text{for all } n \geq 0 \end{aligned}$$

Sequential Processes: Introduction

X3. $ADD = DOWN_0$

$DOWN_i = (l.down \rightarrow DOWN_{i+1} \mid l.around \rightarrow UP_i)$

$UP_0 = P$

$UP_{i+1} = l.up \rightarrow m.up \rightarrow UP_i$

- Each operation on a subordinate process explicitly mentions the rest of the user process which follows it.
- Use *SKIP* and sequential composition.

X10 A *USER* process manipulates two count variables named l and m:

$$l : CT_0 \parallel m : CT_3 \parallel USER$$

- The following subprocess (inside the *USER*) adds the current value of l to m

$$ADD = (l.around \rightarrow SKIP \mid l.down \rightarrow (ADD \,;\, (m.up \rightarrow l.up \rightarrow SKIP)))$$

  - If the value of l is initially zero, nothing needs to be done.
  - If l is decremented, its reduced value is added to m (by the recursive call to *ADD*).
  - Then m is incremented once more, and l is also incremented,
    - to compensate for the initial decrementation and bring it back to its initial value.

Sequential Processes: **Laws**

- The laws for sequential composition are similar to those for catenation,
  - with *SKIP* as the unit

L1. $SKIP \,;\, P = P \,;\, SKIP = P$

L2. $(P \,;\, Q) \,;\, R = P \,;\, (Q \,;\, R)$

L3. $(x : B \rightarrow P(x)) \,;\, Q = (x : B \rightarrow (P(x) \,;\, Q))$

- The law for the choice operator has corollaries:

L4. $(a \rightarrow P) \,;\, Q = a \rightarrow (P \,;\, Q)$

L5. $STOP \,;\, Q = STOP$

- When sequential operators are composed in parallel, the combination terminates successfully just when *both* components do so:

L6. $SKIP_A \parallel SKIP_B = SKIP_{A \cup B}$

- A successfully terminating process participates in no further event offered by a partner

L7. $((x : B \rightarrow P(x)) \parallel SKIP_A) = (x : (B - A) \rightarrow (P(x) \parallel SKIP_A))$

L4. $(x : A \rightarrow P(x)) \parallel (y : B \rightarrow Q(y)) = (z : (A \cap B) \rightarrow (P(z) \parallel Q(z)))$

Sequential Processes: **Laws**

- Successful termination of a concurrent composition of a sequential with a nonsequential processes.
- If the alphabet of the sequential process wholly contains that of its partner,
  - termination of the partnership is determined by *the sequential process*,
    - since the other process can do nothing when its partner is finished.

L8. $STOP_A \parallel SKIP_B = SKIP_B$ **if** $\checkmark \notin A \wedge A \subseteq B$.

L1. $SKIP \,;\, P = P \,;\, SKIP = P$
L2. $(P \,;\, Q) \,;\, R = P \,;\, (Q \,;\, R)$
L3. $(x : B \rightarrow P(x)) \,;\, Q = (x : B \rightarrow (P(x) \,;\, Q))$

$$CT_0 = (up \rightarrow CT_1 \mid around \rightarrow CT_0)$$
$$CT_{n+1} = (up \rightarrow CT_{n+2} \mid down \rightarrow CT_n)$$

Sequential Processes: **Laws**

$$\begin{aligned} C_0 &= (around \rightarrow C_0 \mid up \rightarrow C_1) \\ C_{n+1} &= POS \,;\, C_n \\ &= \underbrace{POS \,;\, \ldots\, POS;}_{n \text{ times}} POS \,;\, POS \,;\, C_0 \quad \text{for all } n \geq 0 \end{aligned}$$

$$POS = (down \rightarrow SKIP \mid up \rightarrow POS \,;\, POS)$$

- Prove that C_0 behaves like CT_0.
- Show that C satisfies the set of
  - guarded recursive equations used to define CT.
- The equation for CT_0 is the same as that for C_0
- For $n > 0$, we need to prove $C_n = (up \rightarrow C_{n+1} \mid down \rightarrow C_{n-1})$
- *Proof*

$$\begin{aligned} LHS &= POS \,;\, C_{n-1} && [\text{def } C_n] \\ &= (down \rightarrow SKIP \mid up \rightarrow POS \,;\, POS) \,;\, C_{n-1} && [\text{def } POS] \\ &= (down \rightarrow (SKIP \,;\, C_{n-1}) \mid up \rightarrow (POS \,;\, POS) \,;\, C_{n-1}) && [\text{L3}] \\ &= (down \rightarrow C_{n-1} \mid up \rightarrow POS \,;\, (POS \,;\, C_{n-1})) && [\text{L1, L2}] \\ &= (down \rightarrow C_{n-1} \mid up \rightarrow POS \,;\, C_n) && [\text{def } C_n] \\ &= RHS && [\text{def } C_n] \end{aligned}$$

- Since C_n obeys the same set of guarded recursive equations as CT_n, they are the same.
- An attempt to use induction on n will fail, because
  - the definition of CT_n contains the process CT_{n+1}.

Sequential Processes: **Deterministic processes**

- Operations on deterministic processes are defined in terms of the traces.
- The first and only action of the process *SKIP* is successful termination:

L0. $traces(SKIP) = \{\langle\rangle, \langle\checkmark\rangle\}$

- If s and t are traces and s does not contain ✓
- $(s\ ;\ t) = s$ and $(s\ \hat{}\ \langle\checkmark\rangle)\ ;\ t = s\ \hat{}\ t$
- A trace of $(P\ ;\ Q)$ consists of a trace of P, and if this trace ends in ✓, the ✓ is replaced by a trace of Q:

L1. $traces(P\ ;\ Q) = \{s\ ;\ t \mid s \in traces(P) \wedge t \in traces(Q)\}$

- An equivalent definition is

L1A. $traces(P\ ;\ Q) = \{s \mid s \in traces(P) \wedge \neg \langle\checkmark\rangle \text{ in } s\} \cup$

$$\{s\ \hat{}\ t \mid s\ \hat{}\ \langle\checkmark\rangle \in traces(P) \wedge t \in traces(Q)\}$$

Sequential Processes: **Deterministic processes**

- The intention of the ✓ symbol is that it should terminate the process:

L2. $P\ /\ s = SKIP$ **if** $s \wedge \langle\checkmark\rangle \in traces(P)$

  - This law is essential in the proof of $P\ ;\ SKIP = P$

- It is not in general true:
  - if $P = (SKIP_{\{\}} \parallel c \rightarrow STOP_{\{c\}})$ then $traces(P) = \{\langle\rangle, \langle\checkmark\rangle, \langle c\rangle, \langle c,\checkmark\rangle, \langle\checkmark, c\rangle\}$, but
    - $P / \langle\rangle \neq SKIP$, even though $\langle\checkmark\rangle \in traces(P)$.

- Revise alphabet constraints on parallel composition:
  - $(P \parallel Q)$ must be regarded as invalid unless
    - $aP \subseteq aQ \vee aQ \subseteq aP \vee \checkmark \in (aP \cap aQ \cup \underline{aP} \cap \underline{aQ})$

- alphabet change must be guaranteed to leave ✓ unchanged:
  - $f(P)$ is invalid unless $f(\checkmark) = \checkmark$

- if m is a process name then $m.\checkmark = \checkmark$

- Never use ✓ in the choice construct: $\cancel{(\checkmark \rightarrow P \mid c \rightarrow Q)}$
  - Rules out RUN_A when $\checkmark \in A$.

Sequential Processes: **Non-deterministic processes**

- A nondeterministic process like $SKIP \sqcap (c \rightarrow SKIP)$ does not satisfy the law **L2**.
- Weaken this law to

$f(\checkmark) = \checkmark$
$m.\checkmark = \checkmark$
$\cancel{(\checkmark \rightarrow P \mid c \rightarrow Q)}$

L2A. $s \,\hat{}\, \langle \checkmark \rangle \in traces(P) \Rightarrow (P \,/\, s) \sqsubseteq SKIP$

  - Whenever P can terminate, it can do so
    - without offering any alternative event to the environment.

- To maintain the truth of L2A, all restrictions of the previous section must be hold, and:
  - $SKIP$ must never appear unguarded in an operand of $\square$
  - ✓ must not appear in the alphabet of either operand of |||
- A divergent process remains divergent:

L5. $P \,|||\, RUN = RUN$ **if** P does not diverge

L1. $CHAOS \,;\, P = CHAOS$

- Sequential composition distributes through nondeterministic choice:

L2A. $(P \sqcap Q) \,;\, R = (P \,;\, R) \sqcap (Q \,;\, R)$

L2B. $R \,;\, (P \sqcap Q) = (R \,;\, P) \sqcap (R \,;\, Q)$

Sequential Processes: **Non-deterministic processes**

Refusals, divergences, and failures in $(P \,;\, Q)$.

- If P can refuse X, and cannot terminate successfully,
  - hence $X \cup \{\checkmark\}$ is also a refusal of P.
    - hence X is a refusal of $(P \,;\, Q)$.

- If P terminates successfully, then
  - in $(P \,;\, Q)$ this transition from P to Q may occur autonomously; its occurrence is concealed, and
    - any refusal of Q is also a refusal of $(P \,;\, Q)$.

- If successful termination of P is nondeterministic is also treated:

D1. $refusals(P \,;\, Q) = \{X \mid (X \cup \{\checkmark\}) \in refusals(P)\} \cup \{ X \mid \langle\checkmark\rangle \in traces(P) \wedge X \in refusals(Q) \}$

$failures(P) = \{(s, X) \mid s \in traces(P) \wedge X \in refusals(P / s)\}$

Sequential Processes: **Non-deterministic processes**

- The traces of $(P \,;\, Q)$ are defined as for deterministic processes.

- $(P \,;\, Q)$ diverges whenever P diverges; or
  - when P has terminated successfully and then Q diverges:

D2. $divergences(P \,;\, Q) = \{s \mid s \in divergences(P) \wedge \neg <\surd> \text{ in } s\} \cup$

$\{s \,\hat{}\, t \mid s\hat{}<\surd> \in traces(P) \wedge \neg <\surd> \text{ in } s \wedge t \in divergences(Q)\}$

- Any failure of $(P \,;\, Q)$ is either a failure of P, or
  - it is a failure of Q after P has terminated successfully

D3. $failures(P \,;\, Q) = \{(s, X) \mid (s, X \cup \{\surd\}) \in failures(P)\} \cup$

$\{(s\hat{}t, X) \mid s\hat{}<\surd> \in traces(P) \wedge (t, X) \in failures(Q)\} \cup$

$\{(s, X) \mid s \in divergences(P \,;\, Q)\}$

Sequential Processes: **Interrupts**

- An *interrupt* $(P \triangle Q)$
  - is a special sequential composition,
  - does not depend on successful termination of P.
  - The progress of P is interrupted on occurrence of the first event of Q; and
    - P is never resumed.
- A trace of $(P \triangle Q)$ is
  - a trace of P up to an arbitrary point when the interrupt occurs,
    - followed by any trace of Q.
- $a(P \triangle Q) = aP \cup aQ$
  - ✓ must not be in aP.
- $traces(P \triangle Q) = \{s \hat{} t \mid s \in traces(P) \land t \in traces(Q)\}$

Sequential Processes: **Interrupts**

- The environment determines when Q shall start, by
  - selecting an event which is initially offered by Q but not by P

L1. $(x : B \rightarrow P(x)) \triangle Q = Q \,\square\, (x : B \rightarrow (P(x) \triangle Q))$

- If $(P \triangle Q)$ can be interrupted by R, this is the same as P interruptible by $(Q \triangle R)$:

L2. $(P \triangle Q) \triangle R = P \triangle (Q \triangle R)$

- $STOP$ is a unit of

L3. $P \triangle STOP = P = STOP \triangle P$
  - $STOP$ offers no first event, hence it can never be triggered by the environment.
  - If $STOP$ is interruptible, only the interrupt can actually occur.

- Interrupt distributes through nondeterministic choice

L4A. $P \triangle (Q \sqcap R) = (P \triangle Q) \sqcap (P \triangle R)$

L4B. $(Q \sqcap R) \triangle P = (Q \triangle P) \sqcap (R \triangle P)$
  - the interrupt operator executes both of its operands at most once.

Sequential Processes: **Interrupts**

L5. $CHAOS \mathbin{\triangle} P = CHAOS = P \mathbin{\triangle} CHAOS$

- one cannot cure a divergent process by interrupting it;
- it is not safe to specify a divergent process after the interrupt.

- Further, the possible initial events of the interrupting process are
  - outside the alphabet of the interrupted process.
    - The occurrence of interrupt is visible and controllable by the environment,
      - this restriction preserves determinism.
- The extended definition of the choice operator emphasises the preservation of determinism:
- $(x : B \rightarrow P(x) \mid c \rightarrow Q) \equiv (x : (B \cup \{c\}) \rightarrow (\mathbf{if}\ x = c\ \mathbf{then}\ Q\ \mathbf{else}\ P(x)))$
  - provided that $c \notin B$

Sequential Processes: Interrupts: **Catastrophe**

- Let $↯$ be a symbol standing for a *catastrophic interrupt* event
  - not be caused by P: $↯ \notin \alpha P$
- $P \curlyvee Q = P \triangle (↯ \rightarrow Q)$
  - a process which behaves like P up to catastrophe and thereafter like Q.
    - Q may be intended to effect a recovery after catastrophe.
- A formulation of the informal description of the operator:

L1. $(P \curlyvee Q) \,/\, (s\,\hat{}\,\langle ↯ \rangle) = Q \qquad \text{for } s \in traces(P)$

- In the deterministic model,
  - this single law uniquely identifies the meaning of the operator.
- In a nondeterministic model,
  - uniqueness requires laws for strictness and distributivity in both arguments.

- distributes back through $\rightarrow$:

L2. $(x : B \rightarrow P(x)) \curlyvee Q = (x : B \rightarrow (P(x) \curlyvee Q) \mid ↯ \rightarrow Q)$

- This law uniquely defines the operator on deterministic processes.

Sequential Processes: Interrupts: **Restart**

- Let P be a process such that $↯ \notin \alpha P$.
- *A restartable process* $\acute{P}$ is a process which
  - behaves as P until ↯ occurs, and after each ↯ behaves like P from the start again.
  - $\alpha\acute{P} = \alpha P \cup \{↯\}$
  - $\acute{P} = \mu X \bullet (P \,\hat{}\, X) = P \,\hat{}\, (P \,\hat{}\, (P \,\hat{}\, \ldots))$
    - a guarded recursion, since the occurrence of X is guarded by ↯.
  - $\acute{P}$ is a cyclic process, even if P is not.
- Restarting the unsatisfactory process.
  - a game, some slow running program (a verifier).
- The formalized definition of $\acute{P}$:

L1. $\acute{P} \,/\, s\frown\langle ↯ \rangle = \acute{P}$ $\quad$ for $s \in traces(P)$
  - This law does not uniquely define $\acute{P}$, since
    - it is equally well satisfied by RUN.
  - $\acute{P}$ is the smallest deterministic process that satisfies L1.

Sequential Processes: Interrupts: Alternation

- Let P and Q be processes which play games
  - a human player plays both games simultaneously, alternating between them.
- A symbol $\otimes$ denotes *alternation* between the two games P and Q.
  - the current game is interrupted at an arbitrary point
    - the current state of the current game is preserved,
      - the game can be resumed when the other game is later interrupted.
- The process $(P \otimes Q)$ plays the games P and Q simultaneously.

L1. $\otimes \in (\alpha(P \otimes Q)) - \alpha P - \alpha Q)$

L2. $(P \otimes Q) \ / \ s = (P \ / \ s) \otimes Q$ $\qquad$ if $s \in traces(P)$

L3. $(P \otimes Q) \ / < \otimes > \ = (Q \otimes P)$

Sequential Processes: Interrupts: **Alternation**

- We want the smallest operator that satisfies L2 and L3.
- A more constructive description of the operator can be derived from these laws;
- it shows how x distributes backward through $\rightarrow$

L4. $(x : B \rightarrow P(x)) \otimes Q = (x : B \rightarrow (P(x) \otimes Q) \mid \otimes \rightarrow (Q \otimes P))$

- The alternation operator for operating systems
  - Alternating between system utilities.
    - You do not wish to lose your place in the editor
      - on switching to a "help" program, nor *vice versa*.

Sequential Processes: Interrupts: **Checkpoints**

A checkpoint against loosing data.

- A process P describes the behaviour of a long-lasting data base system.
  - After ↯, is very annoying to restart P in its initial state.
  - It is better to return to some recent good state of the system: a checkpoint.
- A *checkpoint* © happens when the current state of the system is satisfactory.
- When ↯ occurs,
  - the most recent checkpoint is restored; or
  - if there is no checkpoint the initial state is restored.
- *Ch(P)* is the process that behaves as P, but
  - responds in the appropriate fashion to events © and ↯
    - they are not in the alphabet of P.

Sequential Processes: Interrupts: **Checkpoints**

- The formal definition of *Ch(P)*:

L1. $Ch(P) \,/\, (s\hat{}\langle ↯ \rangle) = Ch(P)$ for $s \in traces(P)$

L2. $Ch(P) \,/\, (s\hat{}\langle © \rangle) = Ch(P/s)$ for $s \in traces(P)$

- A process with checkpoint can be defined with the operator *Ch2(P,Q)*
  - *P* is the current process
  - *Q* is the most recent checkpoint waiting to be reinstated.
  - If catastrophe occurs before the first checkpoint, the system restarts

L3. $Ch(P) = Ch2(P, P)$

L4. If $P = (x : B \rightarrow P(x))$ **then** $Ch2(P,Q) = (x : B \rightarrow Ch2(P(x),Q) \mid ↯ \rightarrow Ch2(Q,Q) \mid © \rightarrow Ch2(P, P))$

- A practical implementation of checkpoints:
  - when © occurs, the current state is copied as the new checkpoint;
  - when ↯ occurs, the checkpoint is copied back as the new current state.

Sequential Processes: Interrupts: Multiple checkpoints

- A system *Mch(P)* retains all checkpoints back to the beginning of time.
  - it may happen that a checkpoint is declared in error.
    - cancel the most recent checkpoint, and go back to the one before.
  - Each occurrence of ↯ returns to the state just *before* the most recent ©,
    - rather than the state just after it.
  - $\alpha Mch(P) = \alpha P \cup \{©, ↯\}$
- A ↯ before a © goes back to the beginning

L1. $Mch(P) \;/\; s\frown\langle ↯ \rangle = Mch(P)$ for $s \in traces(P)$

- A ↯ after a © cancels the effect of everything that has happened back to and including the most recent ©

L2. $Mch(P) \;/\; s\frown\langle © \rangle\frown t\frown\langle ↯ \rangle = Mch(P) \;/\; s$ for $(s \restriction \alpha P)\frown t \in traces(P)$

Sequential Processes: Interrupts: **Multiple checkpoints**

- A process with multiple checkpoints can be defined with the operator *Mch2(P,Q)*
  - *P* is the current process and
  - *Q* is the stack of checkpoints waiting to be resumed if necessary.
- The initial content of the stack is an infinite sequence of copies of *P*

L3. $Mch(P) = \mu X \bullet Mch2(P, X) = Mch2(P, Mch(P)) = Mch2(P, Mch2(P, Mch2(P,\ldots)))$

- On occurrence of © the current state is pushed down; on occurrence of ↯ the whole stack is reinstated

L4. If $P = (x : B \rightarrow P(x))$ **then**

$$Mch2(P,Q) = (x : B \rightarrow Mch2(P(x),Q) \mid © \rightarrow Mch2(P, Mch2(P,Q)) \mid ↯ \rightarrow Q)$$

- The multiple checkpoint facility could be very expensive to implement in practice
  - when the number of checkpoints gets large.

Sequential Processes: Interrupts: **Implementation. Promela. Catastrophe**

```
mtype:process = {Pp,Qq};
mtype:process act;
bool ctstrf = false;

proctype P {
...
   if :: ctstrf -> act = Qq;
      :: else -> skip;
   fi
...
}
proctype Q provided (act == Qq){
...
}
```

Sequential Processes: Interrupts:
Implementation. Promela. Catastrophe

```
mtype:process = {Pp,Qq, ...};
mtype:process act;
bool ctstrf_P = false;

proctype P provided ( !ctstrf_P ) {
...
}
proctype Q provided (act == Qq){
...
}
proctype R {
...
   do :: ctstrf_P -> act = Qq;
      :: ctstrf_X -> act = Yy;
      :: ...
   od
...
}
```

Sequential Processes: Interrupts: **Implementation. Promela. Alternation**

```
mtype:process = {Pp,Qq};
mtype:process act;

proctype P provided (act == Pp) {
...
}
proctype Q provided (act == Qq){
...
}
proctype R {
...
    act = Pp;
...
    act = Qq;
...
}
```

L3. $Ch(P) = Ch2(P, P)$

L4. If $P = (x : B \rightarrow P(x))$ **then** $Ch2(P,Q) = (x : B \rightarrow Ch2(P(x),Q)$
$\mid ↯ \rightarrow Ch2(Q,Q) \mid © \rightarrow Ch2(P, P))$

Sequential Processes: Interrupts:
Implementation. Promela. Checkpoints

```
mtype:process = {Pp,Qq, ...};
mtype:process act, checkpnt;
bool ctstrf = false;
bool good = false;

proctype P provided (act == Pp) {
...
   if :: good -> checkpnt = Pp;
      :: else -> skip;
   fi
goodpnt: skip;
...
   if :: ctstrf -> act = checkpnt;
      :: else -> skip;
   fi
...
}
```