Applied Concurrency Theory
Lecture 2 : process calculi

Hubert Garavel

Alexander Graf-Brill

Process calculi
Process algebras

Quick history of process calculi (1/3)

- Research on process calculi started in the late 70s
- Finding a better paradigm than shared memory
- Earlier attempts:
  - Actor model (Hewitt, 1973)
  - monitors (Hoare 1974, Brinch Hansen 1975)
  - guarded commands (Dijkstra, 1975)
- Communicating Sequential Processes (CSP)
  - a new language proposed by C.A.R. Hoare (1978)
  - finite set of concurrent processes
  - message passing communications (‘rendezvous’)
  - binary communication scheme (one sender, one receiver)

Quick history of process calculi (2/3)

- Calculus of Communicating Systems (CCS)
  - a small language and a book by Robin Milner (1980)
  - underlying semantic model: labelled transition systems (LTS)
  - formally-defined operational semantics (SOS rules)
  - use of equivalence relations (bisimulations) to compare LTS
  - algebraic theorems
  - new book by Robin Milner (1989)
- Theoretical CSP
  - revised version of CSP (Brookes, Hoare, Roscoe, 1984)
  - book by C.A.R. Hoare (1985)
  - multiway rendez-vous (more than two parties)

Quick history of process calculi (3/3)

- Algebra of Communicating Processes (ACP)
  - papers by Bergstra, Baeten, Klop (1984-1987)
  - emphasis on algebraic semantics (rather than operational)
  - symmetric sequential composition
- Then, a plethora of derived languages
  - CHP, CIRCAL, FSP, LOTOS, μCRL, OCCAM, pi-calculus, PSF, etc.
- Tool development: compilers, verifiers, etc
  - for CSP: FDR2
  - for CCS: CWB (Concurrency Workbench)
  - for FSP: LTSA
  - for LOTOS: CADP (Construction and Analysis of Distributed Processes)

Process calculi as ‘models’

- Different stages in system/software life cycle
  - Requirements → Models → Programs
  - models are higher level (more abstract) than programs
  - models may be formal or not
  - models may be executable or not
  - models help to detect errors as early as possible

- Process calculi = models for concurrency
  - focus on control aspects (later only, data aspects)
  - process calculi are formal models for mathematical studies
  - process calculi were not necessarily meant to be executable

Process calculi: Scope

- A general computation model
  - quest for generality and abstraction
  - not restricted to software (contrary to shared variables)
  - applicable to hardware, software, security, biology, music etc.
  - but not really intended to complex sequential algorithms!

- Key ideas
  - system = set of actors (or processes) executing in parallel
  - no shared memory (if needed, it can be modelled explicitly)
  - message-passing communication (based on rendezvous)

Syntax

- A minimal (or small) set of algebraic operators
  - each operator does one single thing
  - operators can be combined freely (the ‘Lego’ principle)
  - this gives algebraic terms (≅ ‘programs’)

- Small example: subset of basic CCS
  - set of actions (or events): a, b, c, …
  - set of process behaviour expressions: P, P_0 , P_1 , P_2 , etc.

$P ::= nil$ -- inaction: does nothing

| $a . P_0$ -- prefix: does action a, then behave as P_0

| $P_1 + P_2$ -- choice: does either P_1 or P_2

| $P_1 \,||\, P_2$ -- parallel: does P_1 and P_2 concurrently

Algebraic/Axiomatic semantics

- A first approach to define the semantics
- A finite set of algebraic axioms

$P_1 + P_2 = P_2 + P_1$ -- commutativity of +

$(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$ -- associativity of +

$nil + P = P$ -- nil neutral for +

$P_1 \,||\, P_2 = P_2 \,||\, P_1$ -- commutativity of ||

$(P_1 \,||\, P_2) \,||\, P_3 = P_1 \,||\, (P_2 \,||\, P_3)$ -- associativity of ||

$(a \,.\, P_1) \,||\, (b \,.\, P_2) = a \,.\, (P_1 \,||\, b \,.\, P_2) + b \,.\, (a \,.\, P_1 \,||\, P_2)$

-- interleaving expansion law

  - goal: obtain a consistent and complete set of axioms
  - can be used to prove the equivalence of programs
  - mathematically interesting, but not really useful in practice

Operational semantics

- The mainstream approach to define the semantics of process calculi
- Main ideas:
  - operational semantics: describes the execution of a high-level program in terms of a low-level machine (or by translation to a low-level model)
  - here, the high-level ‘programs’ are algebraic terms
  - here, the low-level machine is a state/transition graph
  - therefore, operational semantics of process calculi is a translation of terms into graphs

Labelled Transition Systems (LTS)

- The standard model for process calculi semantics
- LTS = 4 components:
  - a (non-empty) set S of states
  - an initial state s_0 belonging to S
  - a (non-empty) set A of ‘visible’ actions (or labels), which contains a ‘hidden/internal’ action noted τ
  - a transition relation on S x A x S each transition is a triple: (source state, action, target state)
- States are opaque: no information attached to them
  - one can only distinguish the initial state from the other states
- Transition labels may contain ‘rich’ data
  - channel names
  - lists of typed values

Three uses of LTSs for verification (1/2)

- 1. Visual checking
  - to check a program P, generate LTS (P) and look if it is correct
  - caveat: only works if LTS (P) is small enough to be inspected
  - there exist funny tools for exploring very large graphs

- 2. Model checking
  - to check if LTS (P) satisfies a temporal logic formula
    e.g.: absence of deadlocks, absence of race condition, etc.
  - the model checker can diosplay counter-examples
  - caveat: only works if LTS (S) is small enough (< 10 billion states)

Three uses of LTSs for verification (2/2)

- 3. Equivalence checking
  - with axiomatic semantics, one compares terms algebraically
  - with operational semantics, one compares graphs
  - special equivalences for concurrency: ‘bisimulations’
  - special inclusion relations for concurrency: simulation preorders
  - one can reduce any LTS to a minimal LTS without loosing behaviourally important information
  - caveat: only works if LTS (S) is small enough (< 1 billion states)

Alternative models to LTS

- Action-based models vs state-based models
  - Labelled Transition Systems: information on labels only
  - Kripke Structures: information on states only
  - Kripke Transition Systems: information on states and labels
  - in theory: action-based and state-based are dual notions
  - in practice: action-based is more abstract and better resists evolutions because it only refers to system interfaces rather to system internal variables
- Branching-time models vs linear-time models
  - LTS are branching-time (= graphs)
  - traces are linear-time (= sequences of states/transitions)
  - branching-time models are more compact and adapted to concurrency

Structured Operational Semantics (SOS)

- The semantics of a language is described by a small set of semantic rules

$$\frac{true}{(\mathbf{i}\ ;\ B_0) \xrightarrow{\mathbf{i}} B_0}$$

$$\frac{(B_0 \xrightarrow{L} B_0') \wedge (V_0 = true)}{(\texttt{[}V_0\texttt{]}\ \texttt{->}\ B_0) \xrightarrow{L} B_0'}$$

- SOS rules have a mechanically checkable format
- Principles of translation
  - each state of the LTS is a process calculus algebraic term
  - the initial state is the source program itself
  - this program will be rewritten progressively as it executes
  - one advances step by step (each step ‘fires’ an action of A)

LOTOS

What is LOTOS?

- A international effort to standardize process calculi
  - defined between 1983 and 1989
  - ISO international standard (1989)
  - control part: unifies the best features of CCS and CSP
  - data part: based on abstract data types (ADT)
- Qualities
  - expressivity
  - applicable to many different systems
- Drawbacks
  - too different from usual languages (steep learning curve)
  - data types are cumbersome

LOTOS: lexical elements

- 7 classes of LOTOS identifiers:
  - T : type name
  - S : sort name
  - F : function name (official term: operation identifier)
  - X : variable name (official term: value identifier)
  - P : process name
  - G : gate name (two special gates: τ and δ)
  - λ : specification identifier (used only once after '**specification**')
- These 7 name spaces are disjoint
- identifier '**i**' is reserved for the hidden gate τ
- Comments are noted (* ... *)

LOTOS specification (top-level)

$$
\begin{aligned}
\mathit{program} \quad \equiv \quad & \textbf{specification}\ \lambda\ [G_1, \ldots\ G_m]\ (\widehat{X}_1 : S_1, \ldots\ \widehat{X}_n : S_n)\ :\ \mathit{func} \\
& \mathit{type}_1, \ldots\ \mathit{type}_p \\
& \textbf{behaviour} \\
& \quad B \\
& \textbf{where}\ \mathit{block}_1, \ldots\ \mathit{block}_q \\
& \textbf{endspec}
\end{aligned}
$$

red means ‘unused’

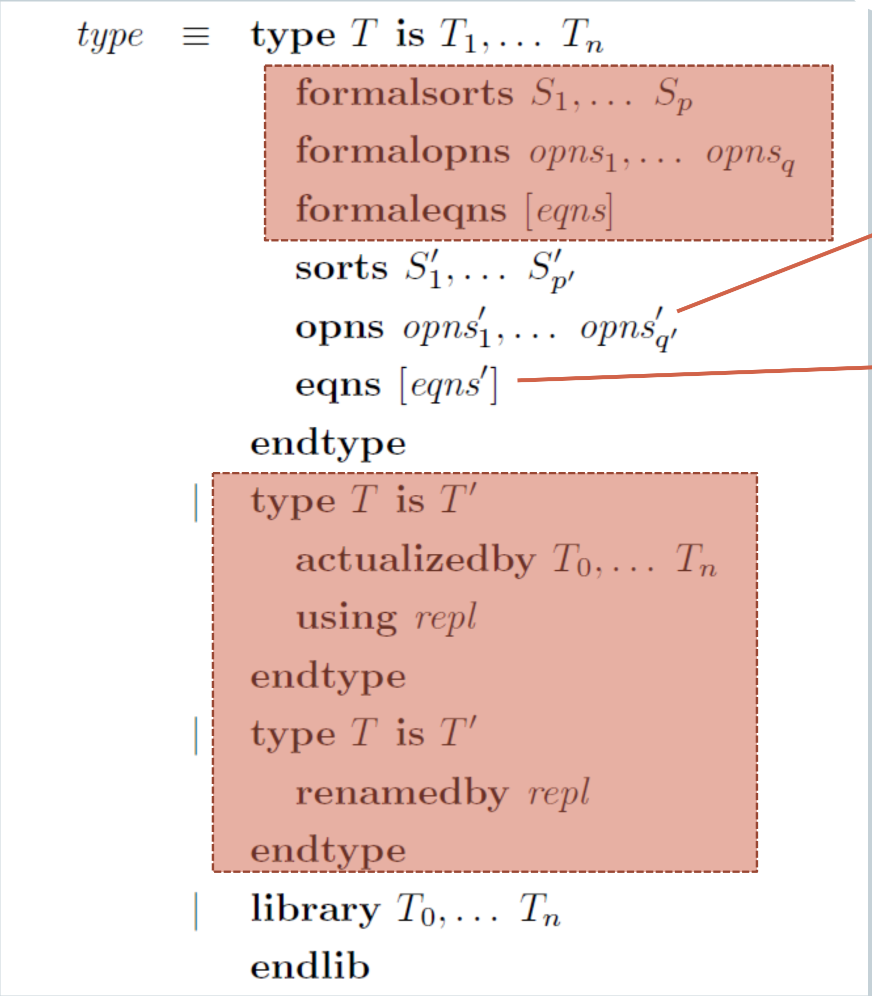
(B will be defined later)

$$
\begin{aligned}
\mathit{block} \quad \equiv \quad & \mathit{process} \\
\mid \quad & \mathit{type}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{func} \quad \equiv \quad & \textbf{noexit} \\
\mid \quad & \textbf{exit}\ (S_1, \ldots\ S_n)
\end{aligned}
$$

LOTOS data types

LOTOS type definitions



$$\begin{array}{lll}
\mathit{type} & \equiv & \textbf{type } T \textbf{ is } T_1, \ldots\ T_n \\
& & \quad \textbf{formalsorts } S_1, \ldots\ S_p \\
& & \quad \textbf{formalopns } \mathit{opns}_1, \ldots\ \mathit{opns}_q \\
& & \quad \textbf{formaleqns } [\mathit{eqns}] \\
& & \quad \textbf{sorts } S'_1, \ldots\ S'_{p'} \\
& & \quad \textbf{opns } \mathit{opns}'_1, \ldots\ \mathit{opns}'_{q'} \\
& & \quad \textbf{eqns } [\mathit{eqns}'] \\
& & \textbf{endtype} \\
& | & \textbf{type } T \textbf{ is } T' \\
& & \quad \textbf{actualizedby } T_0, \ldots\ T_n \\
& & \quad \textbf{using } \mathit{repl} \\
& & \textbf{endtype} \\
& | & \textbf{type } T \textbf{ is } T' \\
& & \quad \textbf{renamedby } \mathit{repl} \\
& & \textbf{endtype} \\
& | & \textbf{library } T_0, \ldots\ T_n \\
& & \textbf{endlib}
\end{array}$$

red means ‘unused’

$$\mathit{opns} \equiv F_0, \ldots\ F_m : S_1, \ldots\ S_n \texttt{ -> } S$$

$$\begin{array}{rcl}
\mathit{meq} & \equiv & V_1, \ldots\ V_n \texttt{ => } V \\
\mathit{ceq} & \equiv & \textbf{ofsort } S \textbf{ forall } \widehat{X_1}{:}S_1, \ldots\ \widehat{X_m}{:}S_m\ \mathit{meq}_0, \ldots\ \mathit{meq}_n \\
\mathit{eqns} & \equiv & \textbf{forall } \widehat{X_1}{:}S_1, \ldots\ \widehat{X_m}{:}S_m\ \mathit{ceq}_0, \ldots\ \mathit{ceq}_n
\end{array}$$

LOTOS vocabulary is non-standard:

- ‘type’ means ‘module’
- ‘sort’ means ‘type’
- ‘operation’ means ‘function’

library T, T’ endlib is interpreted as:

```
#include  "T.lib"
#include  "T’.lib"
```

LOTOS value expressions

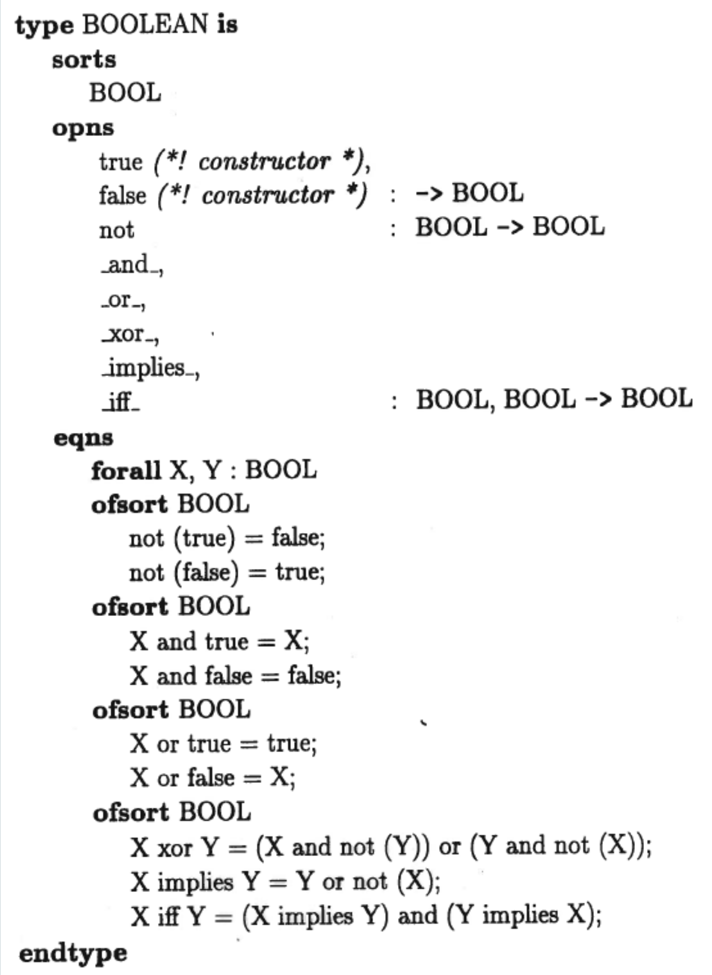
A value expression (non-terminal symbol: V) is either:

- a variable
- a function call with a (possibly empty) list of value expressions
- an equality test between two values

$$
\begin{array}{rcl}
V & \equiv & X \\
 & | & F\ (V_1, \ldots\ V_n) \\
 & | & V_1 = V_2
\end{array}
$$

- notation ‘V of S’ means that V has sort S (to resolve type ambiguities)

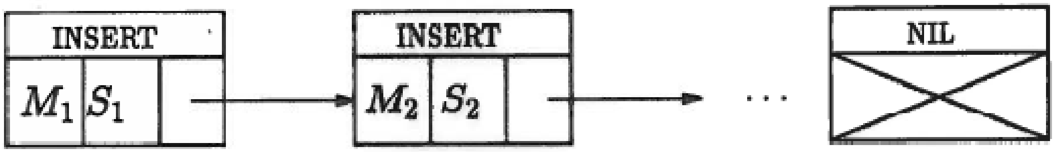
Abstract data types: example 1



```
type BOOLEAN is
   sorts
      BOOL
   opns
       true (*! constructor *),
       false (*! constructor *)   :  -> BOOL
       not                        :  BOOL -> BOOL
       _and_,
       _or_,
       _xor_,
       _implies_,
       _iff_                      :  BOOL, BOOL -> BOOL
   eqns
      forall X, Y : BOOL
      ofsort BOOL
         not (true) = false;
         not (false) = true;
      ofsort BOOL
         X and true = X;
         X and false = false;
      ofsort BOOL
         X or true = true;
         X or false = X;
      ofsort BOOL
         X xor Y = (X and not (Y)) or (Y and not (X));
         X implies Y = Y or not (X);
         X iff Y = (X implies Y) and (Y implies X);
endtype
```

Abstract data types: example 2

```
type RANDOM_ACCESS_QUEUE is BOOLEAN, MESSAGE, STATUS
  sorts
    QUEUE
  opns
     NIL (*! constructor *)       : -> QUEUE
     INSERT (*! constructor *)    : MSG, STAT, QUEUE -> QUEUE
     EMPTY                        : QUEUE -> BOOL
     HEAD_MESSAGE                 : QUEUE -> MSG
     HEAD_STATUS                  : QUEUE -> STAT
     TAIL                         : QUEUE -> QUEUE
     DELETE                       : QUEUE, MSG -> QUEUE
  eqns
    forall M,M1,M2:MSG, S:STAT, Q:QUEUE
    ofsort BOOL
      EMPTY (NIL) = true;
      EMPTY (INSERT (M, S, Q)) = false;
    ofsort MSG
      HEAD_MESSAGE (INSERT (M, S, Q)) = M;
    ofsort STAT
      HEAD_STATUS (INSERT (M, S, Q)) = S;
    ofsort QUEUE
      TAIL (NIL) = NIL;
      TAIL (INSERT (M, S, Q)) = Q;
    ofsort QUEUE
      DELETE (NIL, M) = NIL;
      DELETE (INSERT (M, S, Q), M) = Q;
      M1 <> M2 => DELETE (INSERT (M1, S, Q), M2) =
                      INSERT (M1, S, DELETE (Q, M2));
endtype
```



LOTOS processes

LOTOS process definitions

process P $[G_1, \ldots\ G_m]$ $(\widehat{X_1}:S_1, \ldots\ \widehat{X_n}:S_n)$: func :=
  B
where $\mathit{block}_1, \ldots\ \mathit{block}_p$
endproc

lists of variables

- P is the process identifier, whereas B is a behaviour expression defining the ‘body’ of P
- LOTOS processes have two lists of parameters
- betwen brackets: a list of (untyped) gates
- between parentheses: a list of (typed) variables

LOTOS non-terminal symbols

- Five symbols to be defined:
  - B : behaviour expression
  - O : offer (official term: experiment offer)
  - op : parallel operator
  - R : result
  - V : value expression (see above)

- Note:
  - the ISO concrete grammar has many more non-terminals
  - this presentation is much simpler, but equivalent

LOTOS behaviour expressions

$$
\begin{array}{lcl}
B & \equiv & \textbf{stop} \\
 & | & \textbf{i}\ ;\ B_0 \\
 & | & G\ O_1,\ldots\ O_n\ [[V_0]]\ ;\ B_0 \\
 & | & B_1\ [\,]\ B_2 \\
 & | & \textbf{choice}\ \widehat{G_0}\ \textbf{in}\ [\widehat{G'_0}],\ldots\ \widehat{G_n}\ \textbf{in}\ [\widehat{G'_n}]\ [\,]\ B_0 \\
 & | & B_1\ op\ B_2 \\
 & | & \textbf{par}\ \widehat{G_0}\ \textbf{in}\ [\widehat{G'_0}],\ldots\ \widehat{G_n}\ \textbf{in}\ [\widehat{G'_n}]\ op\ B_0 \\
 & | & \textbf{hide}\ G_0,\ldots\ G_n\ \textbf{in}\ B_0 \\
 & | & [V_0]\ \texttt{->}\ B_0 \\
 & | & \textbf{let}\ \widehat{X_0}:S_0\texttt{=}V_0,\ldots\ \widehat{X_n}:S_n\texttt{=}V_n\ \textbf{in}\ B_0 \\
 & | & \textbf{choice}\ \widehat{X_0}:S_0,\ldots\ \widehat{X_n}:S_n\ [\,]\ B_0 \\
 & | & \textbf{exit}\ (R_1,\ldots\ R_n) \\
 & | & B_1\ \texttt{>>}\ \textbf{accept}\ \widehat{X_1}:S_1,\ldots\ \widehat{X_n}:S_n\ \textbf{in}\ B_2 \\
 & | & B_1\ [\texttt{>}\ B_2 \\
 & | & P\ [G_1,\ldots\ G_n]\ (V_1,\ldots\ V_m)
\end{array}
$$

$$
\begin{array}{lcl}
O & \equiv & !V \\
 & | & ?X_0,\ldots\ X_n:S
\end{array}
$$

$$
\begin{array}{lcl}
op & \equiv & || \\
 & | & ||| \\
 & | & |[G_0,\ldots\ G_n]|
\end{array}
$$

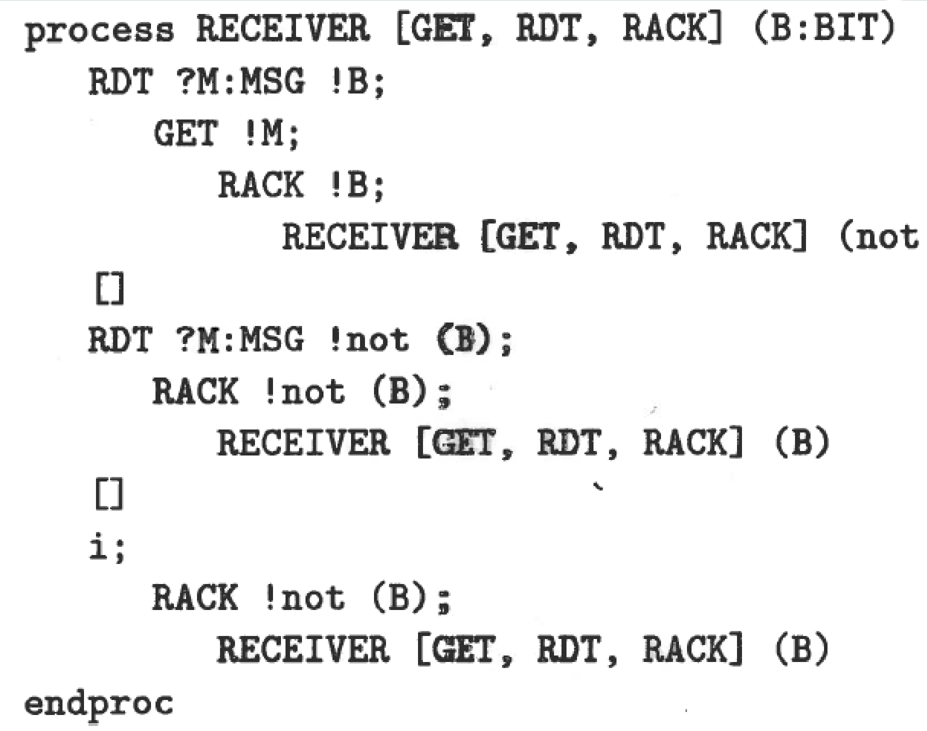
$$
\begin{array}{lcl}
R & \equiv & V \\
 & | & \textbf{any}\ S
\end{array}
$$

red means ‘unused’

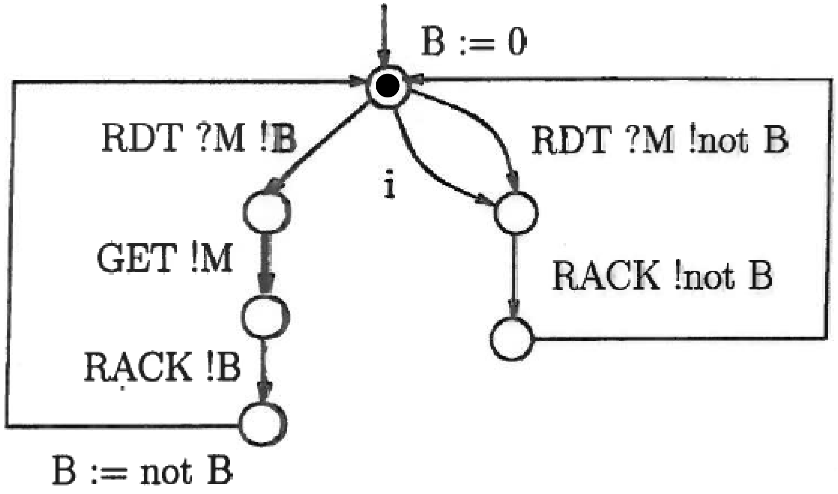
Sequential processes in a nutshell

- Trees of actions are easy to obtain by combining
  - stop (deadlock state)
  - ; (action prefix)
  - [] (choice)
- To create loops, one must use a recursive process
- LOTOS variables are ‘dynamic constants’
  - they are assigned only once when declared (i.e., ‘X:S’)
  - they cannot be modified afterwards
  - except by a recursive process call: P [...](X) calls P [...](X+1)
  - this is a way LOTOS ensures that variables are assigned before used
- Parentheses rules are cumbersome, but essential

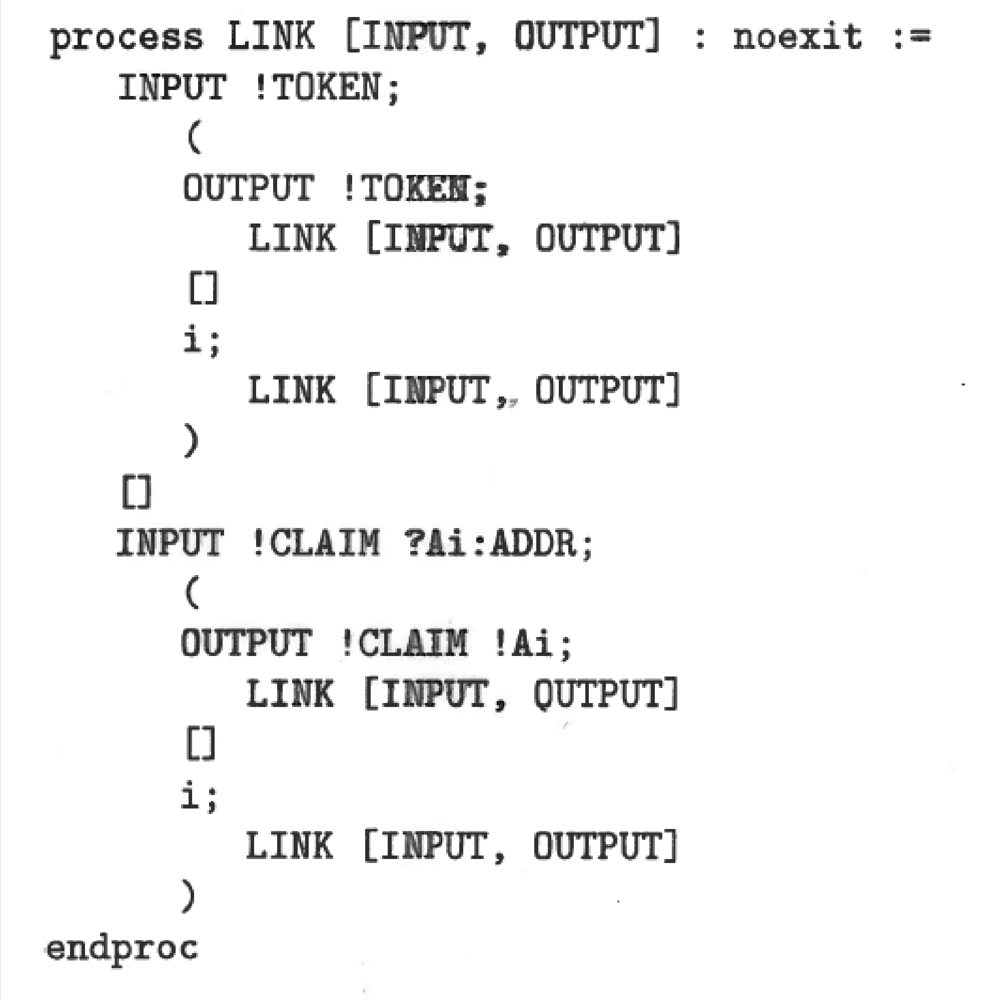
Sequential processes: example 1



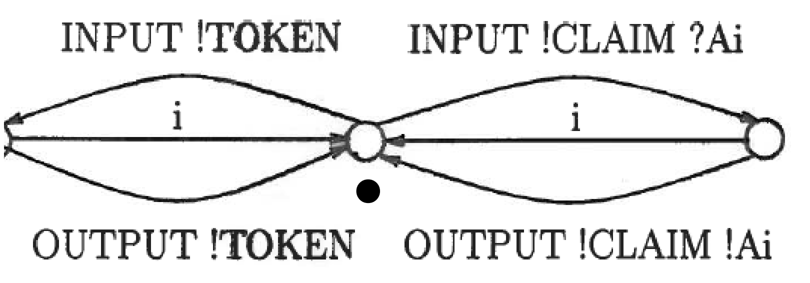
```
process RECEIVER [GET, RDT, RACK] (B:BIT) : noexit :=
   RDT ?M:MSG !B;
      GET !M;
         RACK !B;
            RECEIVER [GET, RDT, RACK] (not (B))
   []
   RDT ?M:MSG !not (B);
      RACK !not (B);
         RECEIVER [GET, RDT, RACK] (B)
   []
   i;
      RACK !not (B);
         RECEIVER [GET, RDT, RACK] (B)
endproc
```



Sequential processes: example 2



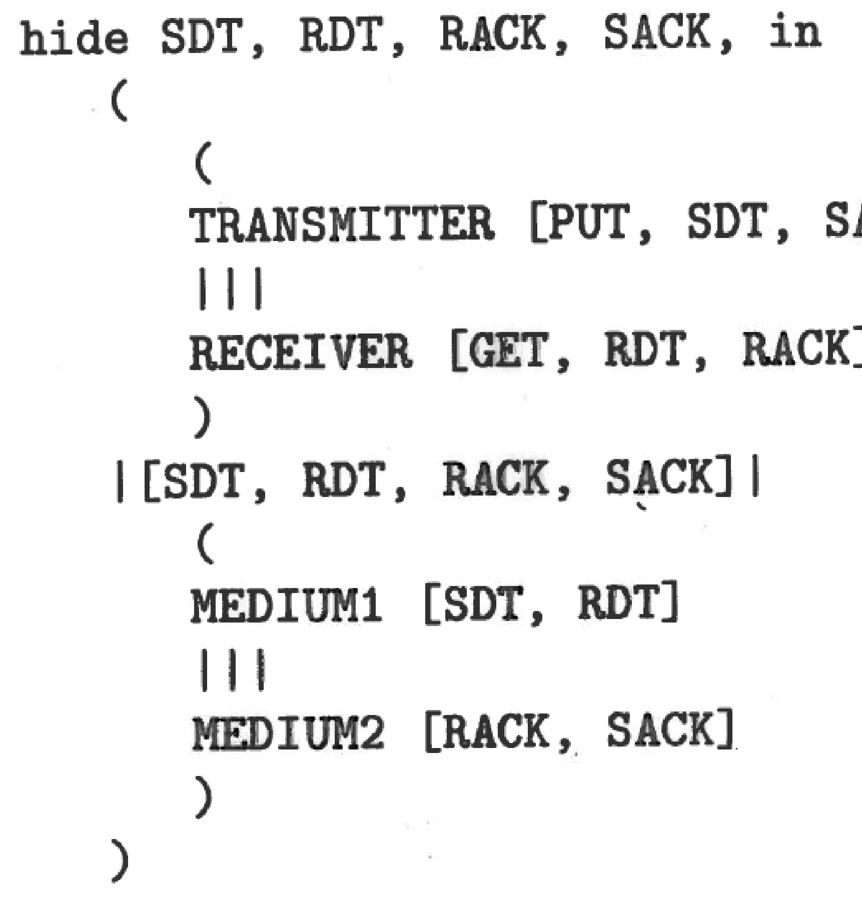
```
process LINK [INPUT, OUTPUT] : noexit :=
    INPUT !TOKEN;
        (
        OUTPUT !TOKEN;
            LINK [INPUT, OUTPUT]
        []
        i;
            LINK [INPUT, OUTPUT]
        )
    []
    INPUT !CLAIM ?Ai:ADDR;
        (
        OUTPUT !CLAIM !Ai;
            LINK [INPUT, OUTPUT]
        []
        i;
            LINK [INPUT, OUTPUT]
        )
endproc
```



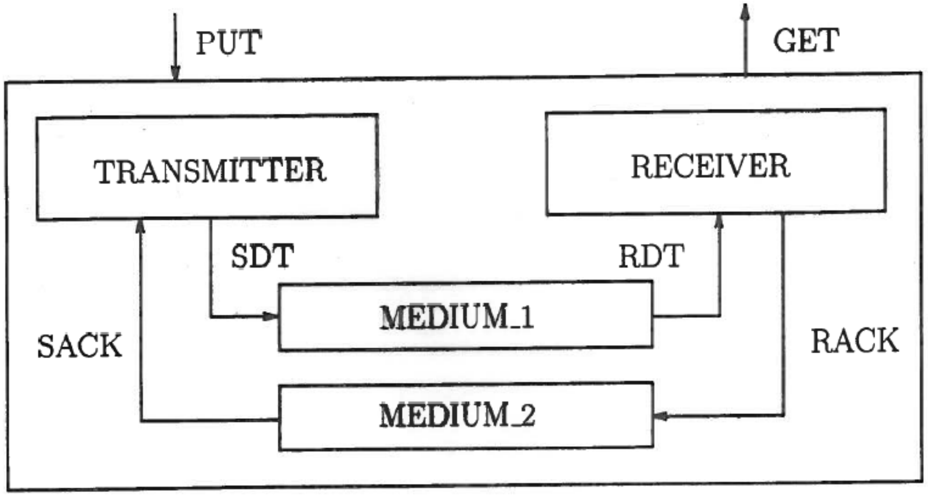
Parallel processes in a nutshell

- The rules of the game:
  - one must describe sets of boxes (= processes)
  - boxes can be nested one into another (= nested processes)
  - boxes are connected by links (= gates)
  - more than two boxes can connect on the same link (= multiway rendezvous)
  - links can be hidden to avoid third-party interference and to make internal details unobservable
  - all of this must be described using only the (binary) parallel operators and the (unary) hiding operator
- Three parallel operators
  - || : synchronize on all visible gates (includes δ, excludes τ)
  - ||| : don't synchronize on any gate (excepted δ)
  - |[G_0, ... G_n]| : synchronize on gates G_0, ... G_n and δ
  - the 1st and 2nd operators are particular cases of the 3rd one

Parallel processes: example 1

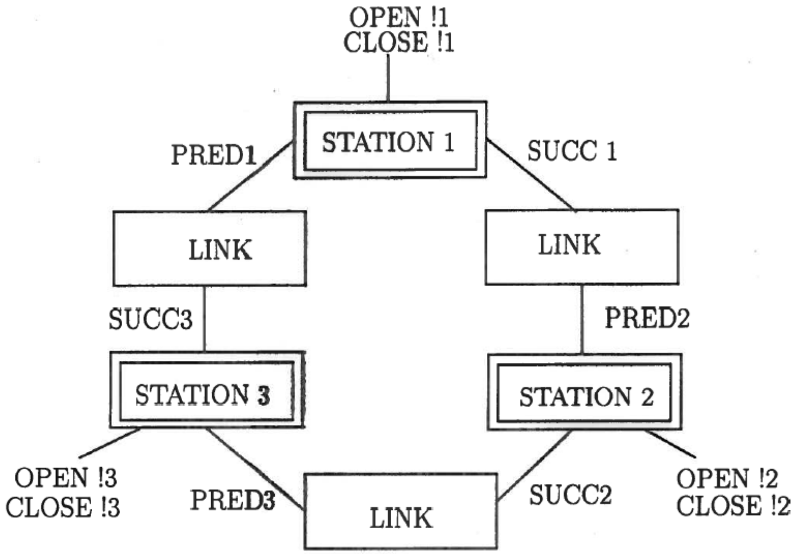


```
hide SDT, RDT, RACK, SACK, in
   (
      (
      TRANSMITTER [PUT, SDT, SACK] (0)
      |||
      RECEIVER [GET, RDT, RACK] (0)
      )
   |[SDT, RDT, RACK, SACK]|
      (
      MEDIUM1 [SDT, RDT]
      |||
      MEDIUM2 [RACK, SACK]
      )
   )
```

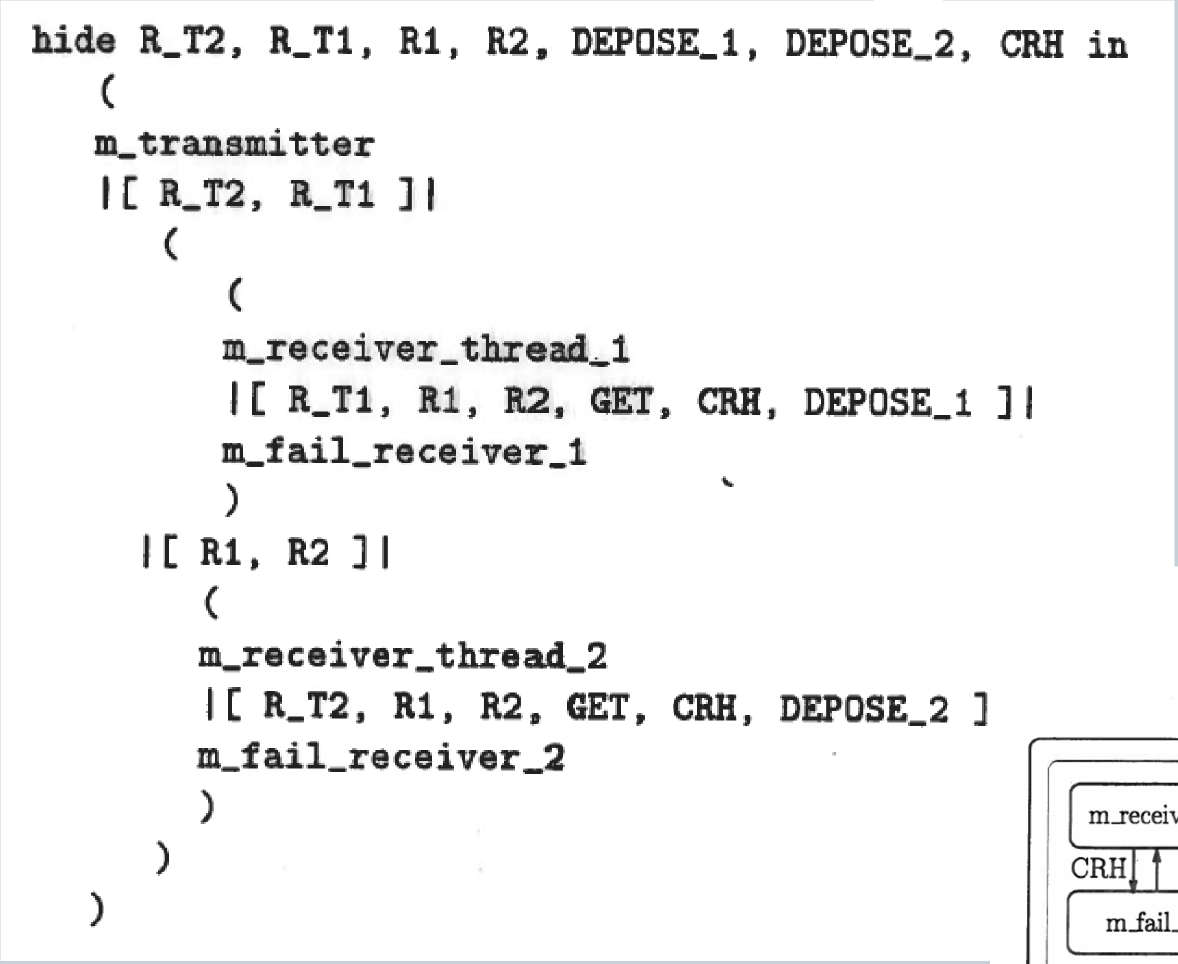


Parallel processes: example 2

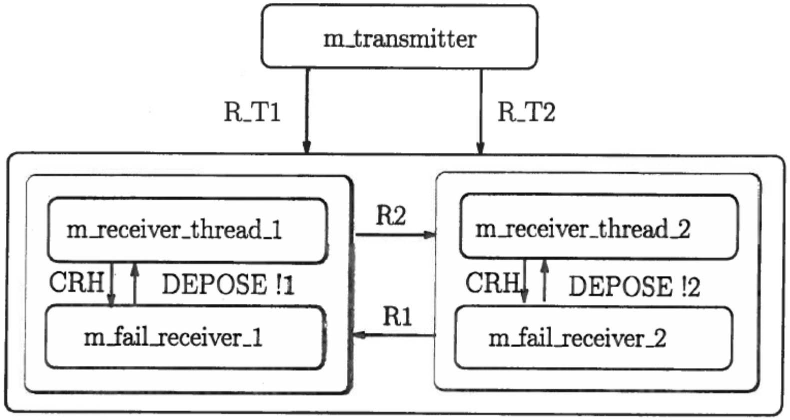
```
   (
   STATION [OPEN, CLOSE, PRED1, SUCC1] (A1)
   |||
   STATION [OPEN, CLOSE, PRED2, SUCC2] (A2)
   |||
   STATION [OPEN, CLOSE, PRED3, SUCC3] (A3)
   )
|[PRED1, SUCC1, PRED2, SUCC2, PRED3, SUCC3]|
   (
   LINK [SUCC1, PRED2]
   |||
   LINK [SUCC2, PRED3]
   |||
   LINK [SUCC3, PRED1]
   )
```



Parallel processes: example 3



```
hide R_T2, R_T1, R1, R2, DEPOSE_1, DEPOSE_2, CRH in
   (
   m_transmitter
   |[ R_T2, R_T1 ]|
      (
         (
         m_receiver_thread_1
         |[ R_T1, R1, R2, GET, CRH, DEPOSE_1 ]|
         m_fail_receiver_1
         )
     |[ R1, R2 ]|
        (
        m_receiver_thread_2
        |[ R_T2, R1, R2, GET, CRH, DEPOSE_2 ]
        m_fail_receiver_2
        )
      )
   )
```



Today's challenge

Today’s challenge (1/2)

- Get the LOTOS tutorial by Bolognesi & Brinksma
- Copy the LOTOS example ‘Max3’ in ‘max.lotos’
  - from ‘specification’ to ‘endspec’
  - beware of a dozen copy-paste errors! (this is a scanned PDF)
  - insert ‘(*! constructor *)’ between ‘opns zero’ and before ‘: -> nat’
  - insert ‘(*! constructor *)’ between ‘succ’ and before ‘: nat -> nat’
  - replace equation ‘largest(x, y) = largest(y, x);’ with ‘largest(x, zero) = x;’
  - create (in the same directory) a text file named ‘max.t’ containing only two lines:

```
#define CAESAR_ADT_EXPERT_T  5.3
#define CAESAR_ADT_ITR_NEXT_NAT(CAESAR_ADT_0) ((CAESAR_ADT_0)++ < 5)
```

(this restricts NAT values to the range 0..5)

Today's challenge (2/2)

- Compile the data types of your LOTOS specification:
  - $ caesar.adt max.lotos
  - fix the remaining syntax errors that escaped your attention
- Compile the processes of your LOTOS specification:
  - $ caesar max.lotos
  - this generates an LTS stored in file max.bcg
- Minimize this file using strong bisimulation:
  - $ bcg_min max.bcg
- Display this file:
  - $ bcg_edit max.bcg
  - send the PostScript drawing of this LTS to Alexander

References

Historical papers on CSP and CCS

- C. A. R. Hoare. *Communicating sequential processes*. Communications of the ACM, 21 (8), 1978.
- S. Brookes, C. A. R. Hoare, A. W. Roscoe. *A Theory of Communicating Sequential Processes*. Journal of the ACM, 31 (3), 1984.
- C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- R. Milner. *A Calculus of Communicating Systems*. Springer Verlag. 1980.
- R. Milner. *Communication and Concurrency*. Prentice Hall. 1989.

Tutorials on LOTOS

- T. Bolognesi and E. Brinksma. *Introduction to the ISO specification language LOTOS*. Computer Networks and ISDN Systems, vol. 14, num. 1, 1987.
  http://doc.utwente.nl/69857/1/Bolognesi87introduction.pdf

- L. Logrippo, M Faci, and M. Haj-Hussein. *An introduction to LOTOS: learning by examples*. Computer Networks and ISDN Systems, vol. 23, num. 5, 1992.
  http://lotos.site.uottawa.ca/ftp/pub/Lotos/Papers/tutorial.pdf

- More LOTOS tutorials: http://cadp.inria.fr/tutorial

Tutorial on CADP tools (optional)

- H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. TACAS 2011 http://cadp.inria.fr/vasy/publications/Garavel-Lang-Mateescu-Serwe-11.html
- More CADP info: http://cadp.inria.fr/tutorial