Applied Concurrency Theory
Lecture 2: process calculi

Hubert Garavel
Alexander Graf-Brill
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

- **Theoretical CSP**
  - revised version of CSP (Brookes, Hoare, Roscoe, 1984)
  - multiway rendez-vous (more than two parties)
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)
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
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, P₀, P₁, P₂, etc.
    
    \[ P ::= \text{nil} \quad \text{-- inaction: does nothing} \]
    \[ \mid a \cdot P₀ \quad \text{-- prefix: does action a, then behave as } P₀ \]
    \[ \mid P₁ + P₂ \quad \text{-- choice: does either } P₁ \text{ or } P₂ \]
    \[ \mid P₁ || P₂ \quad \text{-- parallel: does } P₁ \text{ and } P₂ \text{ concurrently} \]
Algebraic/Axiomatic semantics

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

\[
\begin{align*}
P_1 + P_2 &= P_2 + P_1 & \text{-- commutativity of +} \\
(P_1 + P_2) + P_3 &= P_1 + (P_2 + P_3) & \text{-- associativity of +} \\
\text{nil} + P &= P & \text{-- nil neutral for +} \\
P_1 \parallel P_2 &= P_2 \parallel P_1 & \text{-- commutativity of } \parallel \\
(P_1 \parallel P_2) \parallel P_3 &= P_1 \parallel (P_2 \parallel P_3) & \text{-- associativity of } \parallel \\
(a \cdot P_1) \parallel (b \cdot P_2) &= a \cdot (P_1 \parallel b \cdot P_2) + b \cdot (a \cdot P_1 \parallel P_2) & \text{-- interleaving expansion law}
\end{align*}
\]

- 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
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 $\tau$
  - a transition relation on $S \times A \times S$
    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 display 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 losing 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.

\[ \begin{align*}
\text{true} & \quad (i; B_0) \xrightarrow{i} B_0 \\
(B_0 \xrightarrow{L} B'_0) \land (V_0 = \text{true}) & \quad ([V_0] \rightarrow B_0) \xrightarrow{L} B'_0
\end{align*} \]

- 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).
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: $\tau$ and $\delta$)
  - $\lambda$: specification identifier (used only once after ‘specification’)

- These 7 name spaces are disjoint

- Identifier ‘i’ is reserved for the hidden gate $\tau$

- Comments are noted (* ... *)
LOTOS specification (top-level)

\[
\text{program} \equiv \text{specification} \lambda [G_1, \ldots G_m] \\ \text{type}_1, \ldots \text{type}_p \\
\text{behaviour} \quad B \\
\text{where block}_1, \ldots \text{block}_q \\
\text{endspec}
\]

\[
\text{block} \equiv \text{process} \quad | \quad \text{type}
\]

\[
\text{func} \equiv \text{noexit} \quad | \quad \text{exit} (S_1, \ldots S_n)
\]

\text{red means ‘unused’}

(B will be defined later)
LOTOS data types
LOTOS type definitions

\[
\text{type} \equiv \text{type } T \text{ is } T_1, \ldots, T_n \\
\text{formalsorts } S_1, \ldots, S_p \\
\text{formalopns } opns_1, \ldots, opns_q \\
\text{formaleqns } [eqns] \\
\text{sorts } S'_1, \ldots, S'_p \\
\text{opns } opns'_1, \ldots, opns'_q \\
\text{eqns } [eqns'] \\
\text{endtype} \\
\text{type } T' \text{ is } T' \\
\text{actualized by } T_0, \ldots, T_n \\
\text{using } repl \\
\text{endtype} \\
\text{type } T' \text{ is } T' \\
\text{renamed by } repl \\
\text{endtype} \\
\text{library } T_0, \ldots, T_n \\
\text{endlib}
\]

red means ‘unused’

\[
\text{opns} \equiv F_0, \ldots, F_m : S_1, \ldots, S_n \rightarrow S
\]

\[
\begin{align*}
\text{meq} & \equiv V_1, \ldots, V_n \Rightarrow V \\
\text{ceq} & \equiv \text{ofsort } S \text{ forall } X_1:S_1, \ldots, X_m:S_m \text{ meq}_0, \ldots, \text{meq}_n \\
\text{eqns} & \equiv \text{forall } X_1:S_1, \ldots, X_m:S_m \text{ ceq}_0, \ldots, \text{ceq}_n
\end{align*}
\]

LOTOS vocabulary is non-standard:
- ‘type’ means ‘module’
- ‘sort’ means ‘type’
- ‘operation’ means ‘function’

library T, T’ endlib is interpreted as:
\[
\text{#include } "T.lib"
\]
\[
\text{#include } "T'.lib"
\]
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

\[
V \equiv X \\
| F (V_1, \ldots, V_n) \\
| V_1 = V_2
\]

- notation ‘V of S’ means that V has sort S (to resolve type ambiguities)
Abstract data types: example 1

\begin{verbatim}
\textbf{type} BOOLEAN \textbf{is}
\begin{align*}
\textbf{sorts} & \quad \text{BOOL} \\
\textbf{ops} & \quad \text{true (}{^*! \textbf{constructor}}^*\text{)}, \\
           & \quad \text{false (}{^*! \textbf{constructor}}^*\text{)} : \rightarrow \text{BOOL} \\
           & \quad \text{not} : \text{BOOL} \rightarrow \text{BOOL} \\
           & \quad \text{and}_\rightarrow \\
           & \quad \text{or}_\rightarrow \\
           & \quad \text{xor}_\rightarrow \\
           & \quad \text{implies}_\rightarrow \\
           & \quad \text{iff}_\rightarrow : \text{BOOL, BOOL} \rightarrow \text{BOOL}
\end{align*}
\textbf{eqns}
\begin{align*}
\text{forall } X, Y : \text{BOOL} \\
\text{ofsort BOOLEAN} \\
\quad \text{not (true)} = \text{false;}
\quad \text{not (false)} = \text{true;}
\text{ofsort BOOLEAN} \\
\quad X \text{ and true} = X;
\quad X \text{ and false} = \text{false;}
\text{ofsort BOOLEAN} \\
\quad X \text{ or true} = \text{true;}
\quad X \text{ or false} = X;
\text{ofsort BOOLEAN} \\
\quad X \text{ xor Y} = (X \text{ and not (Y)}) \text{ or (Y and not (X));}
\quad X \text{ implies Y} = Y \text{ or not (X);}
\quad X \text{ iff Y} = (X \text{ implies Y}) \text{ and (Y implies X;)}
\end{align*}
\textbf{endtype}
\end{verbatim}
Abstract data types: example 2

```haskell
module AbstractDataTypes where

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

- P is the process identifier, whereas B is a behaviour expression defining the ‘body’ of P
- LOTOS processes have two lists of parameters
  - between 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

\[ B \equiv \text{stop} \]
\[ i \; ; \; B_0 \]
\[ G \; O_1, \ldots, O_n \; [[V_0]] \; ; \; B_0 \]
\[ B_1 \; \parallel \; B_2 \]
\[ \text{choice } \widehat{G}_0 \; \text{in } [[G_0^\prime]] \ldots \ldots \widehat{G}_n \; \text{in } [[G_n^\prime]] \; \parallel \; B_0 \]
\[ B_1 \; \text{op} \; B_2 \]
\[ \text{par } \widehat{G}_0 \; \text{in } [[G_0^\prime]] \ldots \ldots \widehat{G}_n \; \text{in } [[G_n^\prime]] \; \text{op} \; B_0 \]
\[ \text{hide } G_0, \ldots, G_n \; \text{in } B_0 \]
\[ [V_0] \rightarrow B_0 \]
\[ \text{let } \widehat{X}_0 : S_0 = V_0, \ldots, \widehat{X}_n : S_n = V_n \; \text{in } B_0 \]
\[ \text{choice } \widehat{X}_0 : S_0, \ldots, \widehat{X}_n : S_n \; \parallel \; B_0 \]
\[ \text{exit } (R_1, \ldots, R_n) \]
\[ B_1 \; \gg \; \text{accept } \widehat{X}_1 : S_1, \ldots, \widehat{X}_n : S_n \; \text{in } B_2 \]
\[ B_1 \; [\rightarrow \; B_2 \]
\[ P \; [G_1, \ldots, G_n] \; (V_1, \ldots, V_m) \]

\[ O \equiv \; \! V \]
\[ \; \mid \; ?X_0, \ldots, X_n : S \]

\[ \text{op} \equiv \mid \mid \mid \]
\[ \mid \mid \mid \mid \mid [G_0, \ldots, G_n] \]

\[ R \equiv \; V \]
\[ \; \mid \; \text{any } S \]

red means ‘unused’
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
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
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 $\delta$, excludes $\tau$
- $|||$: don’t synchronize on any gate (excepted $\delta$)
- $|[G_0, \ldots, G_n]|$: synchronize on gates $G_0, \ldots, G_n$ and $\delta$
- the 1$^{st}$ and 2$^{nd}$ operators are particular cases of the 3$^{rd}$ one
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]  
)
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
Historical papers on CSP and CCS

Tutorials on LOTOS


- More LOTOS tutorials: [http://cadp.inria.fr/tutorial](http://cadp.inria.fr/tutorial)
Tutorial on CADP tools (optional)


- More CADP info: http://cadp.inria.fr/tutorial