Model Checking of Action-Based Concurrent Systems

Radu Mateescu
INRIA Rhône-Alpes / VASY
http://www.inrialpes.fr/vasy
Why formal verification?

- Ariane-5 launch failure (1996)
- Mars climate orbiter failure (1999)

 Characteristics of these systems

- Errors due to software
- Complex, often involving parallelism
- Safety-critical

⇒ formal verification is useful for early error detection
informal requirements

requirements capture

modeling

formal specification

expected properties

verification (model checking, equivalence checking, visual checking)

model

testing

traces

system

implementation

rapid prototyping
Outline

- Communicating automata
- Process algebraic languages
- Action-based temporal logics
- On-the-fly verification
- Case study
- Discussion and perspectives
Asynchronous concurrent systems

Characteristics:
- Set of distributed processes
- Message-passing communication
- Nondeterminism

Applications:
- Hardware
- Software
- Telecommunications
CADP toolbox:
Construction and Analysis of Distributed Processes
(http://www.inrialpes.fr/vasy/cadp)

- Description languages:
  - ISO standards (LOTOS, E-LOTOS)
  - Networks of communicating automata

- Functionalities:
  - Compilation and rapid prototyping
  - Interactive and guided simulation
  - Equivalence checking and model checking
  - Test generation

- Case-studies and applications:
  - >100 industrial case-studies
  - >30 derived tools

- Distribution: over 400 sites (2008)
Communicating automata

- Basic notions
- Implicit and explicit representations
- Parallel composition and synchronization
- Hiding and renaming
- Behavioural equivalences
Transformational systems

- Work by computing a result in function of the entries
- Absence of termination undesirable
- Upon termination, the result is unique
- Sequential programming (sorting algorithms, graph traversals, syntax analysis, ...)

Reactive systems

- Work by reacting to the stimuli of the environment
- Absence of termination desirable
- Different occurrences of the same request may produce different results
- Parallel programming (operating systems, communication protocols, Web services, ...)

- Concurrent execution
- Communication + synchronization
Communicating automata

Simple formalism describing the behaviour of concurrent systems

**Black-box** approach:
- One cannot inspect directly the state of the system
- The behaviour of the system can be known only through its interactions with the environment

Synchronization on a gate requires the participation of the process and of its environment (**rendezvous**)

![Diagram of communicating automata with process/automaton (black box) and gate (communication channel)]
Automaton (LTS)

**Labeled Transition System** $M = \langle S, A, T, s_0 \rangle$

- $S$: set of *states* $(s_1, s_2, \ldots)$
- $A$: set of visible *actions* $(a_1, a_2, \ldots)$
- $T$: *transition* relation $(s_1 \xrightarrow{a} s_2 \in T)$
- $s_0 \in S$: *initial state*

**Example:**
process $\text{client}_1$

**Other kinds of automata:**
- Kripke strictures (information associated to states)
- Input/output automata [Lynch-Tuttle]
LTS representations in CADP
(http://www.inrialpes.fr/vasy/cadp)

**Explicit**
- List of transitions
- Allows forward and backward exploration
- Suitable for global verification

**BCG (Binary Coded Graphs) environment**
- API in C for reading/writing
- Tools and libraries for explicit graph manipulation (`bcg_io`, `bcg_draw`, `bcg_info`, `bcg_edit`, `bcg_labels`, ...)
- Global verification tools (XTL)

**Implicit**
- "Successor" function
- Allows forward exploration only
- Suitable for local (or on-the-fly) verification

**Open/Caesar environment [Garavel-98]**
- API in C for LTS exploration
- Libraries with data structures for implicit graph manipulation (stacks, tables, edge lists, hash functions, ...)
- On-the-fly verification tools (`Bisimulatort`, `Evaluator`, ...)

Garavel-98
Server example
(modeled using a single automaton)

- Server able to process two requests concurrently
- State variables $u_1$, $u_2$ storing the request status:
  - Empty (e)
  - Received (r)
  - Handled (h)
- A state: couple $<u_1, u_2>$
- Initial state: $<e, e>$ (ee for short)
- Gates (actions):
  - req1, req2: receive a request
  - res1, res2: send a response
  - i: internal action
LTS of the server
(9 states, 16 transitions)
Remarks

- All the theoretical states are reachable:
  \[| u_1 | \times | u_2 | = 3 \times 3 = 9\]
  (no synchronization between request processings)
- There is no sink state (the system is *deadlock-free*)
- From every state, it is possible to reach the initial state again (the server can be re-initialized)
- Shortcomings of modeling with a single automaton:
  - One must predict all the possible request arrival orders
  - For more complex systems, the LTS size grows rapidly

⇒ need of higher-level modeling features
Server example
(modeled using two concurrent automata)

- Decomposition of the system in two subsystems
  - Every type of request is handled by a subsystem
  - In the server example, subsystems are independent

- Simpler description w.r.t. single automaton:
  \[3 + 3 = 6\text{ states}\]
Decomposition in concurrent subsystems

Required at physical level
- Modeling of distributed activities
- Multiprocessor/multitasking execution platform

Chosen at logical level
- Simplified design of the system
- Well-structured programs

Communication and synchronization between subsystems may introduce behavioural errors (e.g., deadlocks)

In practice, even simple parallel programs may reveal difficult to analyze

➔ need of computer-assisted verification
Parallel composition ("product") of automata

Goals:
- Define internal composition laws
  \[ \otimes : \text{LTS} \times \cdots \times \text{LTS} \rightarrow \text{LTS} \]
  expressing the parallel composition of 2 (or more) LTSs
- Allow synchronizations on one or several actions (gates)
- Allow hierarchical decomposition of a system

Consequences:
- A product of automata can always be translated into a single (sequential) automaton
- The logical parallelism can be implemented sequentially (e.g., time-sharing OS)
Binary parallel composition
(syntax)

EXP language [Lang-05]
- Description of communicating automata
- Extensive set of operators
  - Parallel compositions (binary, general, ...)
  - Synchronization vectors
  - Hiding / renaming, cutting, priority, ...
- Exp.Open compiler $\rightarrow$ implicit LTS representation

Binary parallel composition:

```
“lts1.bcg” | [G1, ..., Gn] | “lts2.bcg”
```

with synchronization on $G_1$, $\ldots$, $G_n$

```
“lts1.bcg” ||| “lts2.bcg”
```

without synchronization (interleaving)
**Binary parallel composition**

*(semantics)*

Let $M_1 = \langle S_1, A_1, T_1, s_{01} \rangle$, $M_2 = \langle S_2, A_2, T_2, s_{02} \rangle$ and $L \subseteq A_1 \cap A_2$ a set of visible actions to be synchronized.

$M_1 \mid [L] \mid M_2 = \langle S, A, T, s_0 \rangle$

- $S = S_1 \times S_2$
- $A = A_1 \cup A_2$
- $s_0 = \langle s_{01}, s_{02} \rangle$
- $T \subseteq S \times A \times S$

defined by $R_1$-$R_3$

\[\begin{align*}
\text{(R1)} & \quad s_1 \xrightarrow{a} s_1' \land a \notin L \\
& \quad \langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1', s_2 \rangle \\
\text{(R2)} & \quad s_2 \xrightarrow{a} s_2' \land a \notin L \\
& \quad \langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1, s_2' \rangle \\
\text{(R3)} & \quad s_1 \xrightarrow{a} s_1' \land s_2 \xrightarrow{a} s_2' \land a \in L \\
& \quad \langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1', s_2' \rangle
\end{align*}\]
Example

\[
\begin{array}{c}
\langle 1 \rangle \\
\langle 2 \rangle \\
\langle 3 \rangle \\
\langle 4 \rangle \\
\langle 5 \rangle \\
\langle 6 \rangle \\
\end{array}
\]

\[
\begin{array}{c}
a \\
b \\
[ b ] \\
c \\
= \\
\end{array}
\]

\[
\begin{array}{c}
\langle 1, 4 \rangle \\
\langle 2, 4 \rangle \\
\langle 3, 5 \rangle \\
\langle 1, 6 \rangle \\
\langle 2, 6 \rangle \\
\end{array}
\]

(a R_1) (b R_2) (c R_3)
Interleaving semantics

Hypothesis:
- Every action is atomic
- One can observe at most one action at a time

suitable paradigm for distributed systems

Parallelism can be expressed in terms of choice and sequence (expansion theorem [Milner-89])
Internal and external choice

- **External** choice (the environment decides which branch of the choice will be executed)
  - the environment can force the execution of a and b by synchronizing on that action

- **Internal** choice (the system decides)
  - the environment may synchronize on a, but this will not remove the nondeterminism
Example of modeling with communicating automata

Mutual exclusion problem:

Given two parallel processes $P_0$ and $P_1$ competing for a shared resource, guarantee that at most one process accesses the resource at a given time.

Several solutions were proposed at software level:
- In centralized setting (Peterson, Dekker, Knuth, ...)
- In distributed setting (Lamport, ...)

Peterson's algorithm [1968]

\[
\begin{align*}
\text{var } & \quad \text{d0 : bool := false} & \text{read by P1, written by P0} \\
\text{var } & \quad \text{d1 : bool := false} & \text{read by P0, written by P1} \\
\text{var } & \quad \text{t } \in \{0, 1\} := 0 & \text{read/written by P0 and P1}
\end{align*}
\]

\[
\begin{align*}
\text{loop forever } & \{ \text{P0} \} \\
1 & : \{ \text{ncs0} \} \\
2 & : \text{d0 := true} \\
3 & : \text{t := 0} \\
4 & : \text{wait (d1 = false or t = 1)} \\
5 & : \{ \text{b_cs0} \} \\
6 & : \{ \text{e_cs0} \} \\
7 & : \text{d0 := false}
\end{align*}
\]

\[
\begin{align*}
\text{endloop}
\end{align*}
\]

\[
\begin{align*}
\text{loop forever } & \{ \text{P1} \} \\
1 & : \{ \text{ncs1} \} \\
2 & : \text{d1 := true} \\
3 & : \text{t := 1} \\
4 & : \text{wait (d0 = false or t = 0)} \\
5 & : \{ \text{b_cs1} \} \\
6 & : \{ \text{e_cs1} \} \\
7 & : \text{d1 := false}
\end{align*}
\]

\[
\begin{align*}
\text{endloop}
\end{align*}
\]
Automata of $P_0$ and $P_1$
Automata of $d_0$, $d_1$, and $t$

1. **$d_0$**
   - Transition: $d0 := false$
   - Transition: $d0 := true$

2. **$d_1$**
   - Transition: $d1 := false$
   - Transition: $d1 := true$

3. **$t$**
   - Transition: $t := 1$
   - Transition: $t := 0$
   - Transition: $t = 0$?
   - Transition: $t = 1$?
Architecture of the system (graphical)

- Synchronized actions: \( \text{«d0:=false», «d0:=true», ...} \)
- Non synchronized actions: \( \text{ncs0, b_cs0, e_cs0, ...} \)
Architecture of the system

(textual)

- Using binary parallel composition:
  \[(P_0 \ ||| P_1)\]
  \[| [ "d_0:=false", "d_0:=true", ... ]|\]
  \[(d_0 \ ||| d_1 \ ||| t)\]

- Using general parallel composition:
  \[\text{par}\]
  \[\text{par}
  "d_0:=false", "d_0:=true", ... \rightarrow P_0
  || "d_1:=false", "d_1:=true", ... \rightarrow P_1
  || "d_0:=false", "d_0:=true", "d_0=false?" \rightarrow d_0
  || "d_1:=false", "d_1:=true", "d_1=false?" \rightarrow d_1
  || "t:=0", "t:=1", "t=0?", "t=1?" \rightarrow t\]
  \[\text{end par}\]
Construction of the LTS
(“product automaton”)

Explicit-state method:
- LTS construction by exploring forward the transition relation, starting at the initial state
- Transitions are generated by using the $R_1$, $R_2$, $R_3$ rules
- Detect already visited states in order to avoid cycling

Several possible exploration strategies:
- Breadth-first, depth-first
- Guided by a criterion / property, ...

Several types of algorithms:
- Sequential, parallel, distributed, ...
Construction of the LTS

\[ S = \{ F, V \} \times \{ F, V \} \times \{ 0, 1 \} \times \{ 1..7 \} \times \{ 1..7 \} \]

\[ A = \{ \text{ncs0, ncs1, \ldots, “d0:=true”, \ldots } \} \]

\[ s_0 = \langle F, F, 0, 1, 1 \rangle = \text{FF011} \]

\[ T = \]

![LTS diagram]

\[ \text{FF011} \]

\[ \text{FF012} \]

\[ \text{FF022} \]

\[ \text{FF023} \]

\[ \text{VF032} \]

\[ \text{VF031} \]

\[ \text{VF041} \]

\[ \text{VF013} \]

\[ \text{VF114} \]
Remarks

The LTS of Peterson’s algorithm is finite:

\[ |S| \approx 50 \leq 2 \times 2 \times 2 \times 7 \times 7 = 392 \]

In the presence of synchronizations, the number of reachable states is (much) smaller than the size of the cartesian product of the variable domains.

Some tools of CADP for LTS manipulation:
- OCIS (step-by-step and guided simulation)
- Executor (random exploration)
- Exhibitor (search for regular sequences)
- Terminator (search for deadlocks)

\[ \rightarrow \text{can be used in conjunction with Exp.Open} \]
Verification

Once the LTS is generated, one can formulate and verify automatically the desired properties of the system.

For Peterson’s algorithm:

- **Deadlock freedom**: each state has at least one successor
- **Mutual exclusion**: at most one process can be in the critical section at a given time
- **Liveness**: no process can indefinitely overtake the other when accessing its critical section

[see the chapter on temporal logics]
Limitations of binary parallel composition

Several ways of modeling a process network:
- Absence of *canonical form*
- Difficult to determine whether two composition expressions denote the same process network
- Difficult to retrieve the process network from a composition expression

The semantics of “| [G₁, ..., Gₙ] |” (rule R₃) does not prevent that other processes synchronize on G₁, ..., Gₙ (*maximal cooperation*)

Some networks cannot be modeled using “| [] |”:
Example
(ring network [Garavel-Sighireanu-99])

Description using binary parallel composition:

\[(P_1 \mid [G_1] \mid P_2 \mid [G_2] \mid P_3 \mid [G_3] \mid P_4) \mid [G_4, G_5] \mid P_5\]

*the composition expression does not reflect the symmetry of the process network*
General parallel composition
[Garavel-Sighireanu-99]

“Graphical” parallel composition operator allowing the composition of several automata and their $m$ among $n$ synchronization:

\[
\text{par} \left[ g_1 \# m_1, \ldots, g_p \# m_p \right. \left. \text{ in } \right] \quad \begin{align*}
G_1 & \rightarrow B_1 \\
\| \quad G_2 & \rightarrow B_2 \\
\cdots & \\
\| \quad G_n & \rightarrow B_n \\
\end{align*}
\text{ end par}
\]

- gates with their associated synchronization degrees
- automata (processes)
- communication interfaces (gate lists)
General parallel composition  
( semantics - rules without synchronization degrees )

\[ \exists a, i . B_i \rightarrow a \rightarrow B_i' \land a \not\in G_i \land \forall j \neq i . B_j' = B_j \]  
\[ \text{par } G_1 \rightarrow B_1, ..., G_n \rightarrow B_n -a\rightarrow \text{par } G_1 \rightarrow B_1', ..., G_n \rightarrow B_n', \]  
\[ \text{(GR1)} \]

mandatory interleaved execution of non-synchronized actions

\[ \exists a . \forall i . \text{if } a \in G_i \text{ then } B_i \rightarrow a \rightarrow B_i' \text{ else } B_j' = B_j \]  
\[ \text{par } G_1 \rightarrow B_1, ..., G_n \rightarrow B_n -a\rightarrow \text{par } G_1 \rightarrow B_1', ..., G_n \rightarrow B_n', \]  
\[ \text{(GR2)} \]

execution in maximal cooperation of synchronized actions
Example (1/3)

Process network unexpressible using “| [] |”:

Description using general parallel composition:

\[
\text{par } G#2 \text{ in } G \rightarrow P_1 \\
\quad | | G \rightarrow P_2 \\
\quad | | G \rightarrow P_3 \\
\text{end par}
\]

maximal cooperation avoided by means of synchronization degrees
Example (2/3)
(ring network [Garavel-Sighireanu-99])

Description using general parallel composition:

\[
\begin{align*}
\text{par} & \quad G_1, G_5 \rightarrow P_1 \\
& \quad \mid \mid \quad G_2, G_1 \rightarrow P_2 \\
& \quad \mid \mid \quad G_3, G_2 \rightarrow P_3 \\
& \quad \mid \mid \quad G_4, G_3 \rightarrow P_4 \\
& \quad \mid \mid \quad G_5, G_4 \rightarrow P_5 \\
\text{end par}
\end{align*}
\]

the symmetry of the process network is also present in the composition expression
Example (3/3)

Definition of “[]” in terms of “par”:
\[ B_1 \parallel [G_1, \ldots, G_n] \parallel B_2 = \text{par} \quad G_1, \ldots, G_n \rightarrow B_1 \]
\[ \parallel \quad G_1, \ldots, G_n \rightarrow B_2 \]
\[ \text{end par} \]

CREW (Concurrent Read / Exclusive Write):
\[ \text{par} \quad W\#2 \text{ in} \]
\[ \quad R, W \rightarrow P_1 \]
\[ \quad || \quad R, W \rightarrow P_2 \]
\[ \quad || \quad R, W \rightarrow P_3 \]
\[ \quad || \quad R, W \rightarrow \text{VAR} \]
\[ \text{end par} \]
Parallel composition using synchronization vectors

- Primitive form of n-ary parallel composition
- Proposed in various networks of automata: MEC [Arnold-Nivat], FC2 [deSimone-Bouali-Madelaine]
- Synchronizations are made explicit by means of *synchronization vectors*

Syntax in the EXP language [Lang-05]:

\[
\text{par } V_1, \ldots, V_m \text{ in } B_1 \ | \ldots \ | \ B_n \text{ end par}
\]

\[
V ::= (G_1 \ | \ _) \ast \ldots \ast (G_n \ | \ _) \Rightarrow G_0
\]
Example
(client-server with gate multiplexing)

Description using synchronization vectors:
par req * _ * req → req, rep * _ * rep → rep,
_ * req * req → req, _ * rep * rep → rep
in
Client₁ || Client₂ || Server
end par

binary synchronization on gates req and res
Behavioural equivalence

Useful for determining whether two LTSs denote the same behaviour

Allows to:

- Understand the semantics of languages (communicating automata, process algebras) having LTS models
- Define and assess translations between languages
- Refine specifications whilst preserving the equivalence of their corresponding LTSs
- Replace certain system components by other, equivalent ones (maintenance)
- Exploit identities between behaviour expressions (e.g., $B_1 \parallel \lnot[G] \parallel B_2 = B_2 \parallel \lnot[G] \parallel B_1$) in analysis tools
A large spectrum of equivalence relations proposed:

- *Trace* equivalence (\(\cong\) language equivalence)
- *Strong* bisimulation [Park-81]
- *Weak* bisimulation [Milner-89]
- *Branching* bisimulation [Bergstra-Klop-84]
- Safety equivalence [Bouajjani-et-al-90]
- ...
Trace equivalence

Trace: sequence of visible actions (e.g., $\sigma = \text{req}_1 \text{res}_1 \text{req}_2 \text{res}_2$)

Notations ($a =$ visible action):

- $s = a =>$: there exists a transition sequence $s \rightarrow s_1 \rightarrow s_2 \ldots \rightarrow a \rightarrow s_k$

- $s = \sigma =>$: there exists a transition sequence $s = a_1 => s_1 \ldots = a_n => s_n$ such that $\sigma = a_1 \ldots a_n$

Two state are trace equivalents iff they are the source of the same traces:

$$s \approx_{\text{tr}} s' \iff \forall \sigma . (s = \sigma => \iff s = \sigma =>)$$
Example
(coffee machine)

The two LTSs below are trace equivalent:

\[
\text{Traces (} M_1 \text{)} = \text{Traces (} M_2 \text{)} = \{ \varepsilon, \text{money, money coffee, money tea} \}
\]

\( M_1 \approx_{\text{tr}} M_2 \)

\( M_1 \): risk of deadlock

Have the two coffee machines the same behaviour w.r.t. a user?
Bisimulation

- Trace equivalence is not sufficiently precise to characterize the behaviour of a system w.r.t. its interaction with its environment

⇒ stronger relations (bisimulations) are necessary

- Two states $s_1$ et $s_2$ are bisimilar iff they are the origin of the same behaviour (execution tree):

$$\forall s_1 \xrightarrow{a} s_1' \ . \ \exists s_2 \xrightarrow{a} s_2' \ . \ s_1' \approx s_2'$$

$$\forall s_2 \xrightarrow{a} s_2' \ . \ \exists s_1 \xrightarrow{a} s_1' \ . \ s_2' \approx s_1'$$

- Bisimulation is an equivalence relation (reflexive, symmetric, and transitive) on states

- Two LTSs are bisimilar iff $s_{01} \approx s_{02}$
Strong bisimulation: the largest bisimulation

\[ M_1 \approx_{st} M_2 \]

\( \Rightarrow \) to show that two LTSs are strongly bisimilar, it is sufficient to find a bisimulation between them
Is strong bisimulation sufficient?

*Trace equivalence* ignores internal actions (\(i\)) and does not capture the branching of transitions

\[\Rightarrow \text{does not distinguish the LTSs below}\]

*Strong bisimulation* captures the branching, but handles internal and visible actions in the same way

\[\Rightarrow \text{does not abstract away the internal behaviour}\]
Weak bisimulation
(or observational equivalence)

In practice, it is necessary to compare LTSs
- By abstracting away internal actions
- By distinguishing the branching

Weak bisimulation
[Milner-89]:

\[
\begin{align*}
\text{every } a\text{-transition corresponds to an } a\text{-transition preceded and followed by 0 or more } \tau\text{-transitions} \\
\text{every } \tau\text{-transition corresponds to 0 or more } \tau\text{-transitions}
\end{align*}
\]
Weak bisimulation
(formal definition)

- Let $M_1 = <S_1, A, T_1, s_{01}>$ and $M_2 = <S_2, A, T_2, s_{02}>$

- A weak bisimulation is a relation $\approx \subseteq S_1 \times S_2$ such that $s_1 \approx s_2$ iff:

  $\forall s_1 -a\rightarrow s_1' . \exists s_2 -\tau^*.a.\tau^*\rightarrow s_2' . s_1' \text{ eq } s_2'$

  $\forall s_1 -\tau\rightarrow s_1' . \exists s_2 -\tau^*\rightarrow s_2' . s_1' \text{ eq } s_2'$

  and

  $\forall s_2 -a\rightarrow s_2' . \exists s_1 -\tau^*.a.\tau^*\rightarrow s_1' . s_1' \text{ eq } s_2'$

  $\forall s_2 -\tau\rightarrow s_2' . \exists s_1 -\tau^*\rightarrow s_1' . s_1' \text{ eq } s_2'$

- $\approx_{obs}$ is the largest weak bisimulation

- $M_1 \approx_{obs} M_2$ iff $s_{01} \approx_{obs} s_{02}$
Example

To show that two LTSs are weakly bisimilar, it is sufficient to find a weak bisimulation between them.

\[ \text{Example} \]

\[ \text{To show that two LTSs are weakly bisimilar, it is sufficient to find a weak bisimulation between them.} \]
Communicating automata
(summary)

**Advantages:**
- Simple model for describing concurrency
- Powerful tools for manipulation
  - MEC (University of Bordeaux)
  - Auto/Autograph/FC2 (INRIA, Sophia-Antipolis)
  - CADP (INRIA, Grenoble)
- Some industrial applications

**Shortcomings:**
- Limited expressiveness
  - No dynamic creation and destruction of automata
  - Impossible to express: A then (B || C) then D
  - No handling of data (each variable = an automaton), unacceptable for complex types (numbers, lists, structures, ...)
- Maintenance difficult and error-prone (large automata)
Process algebraic languages

- Basic notions
- Parallel composition and hiding
- Sequential composition and choice
- Value-passing and guards
- Process definition and instantiation
Process algebras

PAs: theoretical formalisms for describing and studying concurrency and communication

Examples of PAs for asynchronous systems:
- CCS (*Calculus of Communicating Systems*) [Milner-89]
- CSP (*Communicating Sequential Processes*) [Hoare-85]
- ACP (*Algebra of Communicating Processes*) [Bergstra-Klop-84]

Basic idea of PAs:
- Provide a small number of operators
- Construct behaviours by freely combining operators (lego)

Standardized specification languages:
LOTOS
(Language Of Temporal Ordering Specification)

- International standard [ISO 8807] for the formal specification of telecommunication protocols and distributed systems

  http://www.inrialpes.fr/vasy/cadp/tutorial

- Enhanced LOTOS (E-LOTOS): revised standard [2001]

- LOTOS contains two “orthogonal” sublanguages:
  - data part (for data structures)
  - process part (for behaviours)

- Handling data is necessary for describing realistic systems. “Basic LOTOS” (the dataless fragment of LOTOS) is useful only for small examples.
LOTOS - data part

Based on algebraic abstract data types (ActOne):

```
type Natural is
  sorts Nat
  opns 0 : -> Nat
        succ : Nat -> Nat
        + : Nat, Nat -> Nat
  eqns forall M, N : Nat
    ofsort Nat
    0 + N = N;
    succ(M) + N = succ(M + N);
endtype
```

Caesar.Adt compiler of CADP [Garavel-Turlier-92]

ADTs tend to become cumbersome for complex data manipulations (removed in E-LOTOS).
LOTOS - process part

Combines the best features of the process algebras CCS [Milner-89] and CSP [Hoare-85]

Terminal symbols (identifiers):
- Variables: $X_1$, ..., $X_n$
- Gates: $G_1$, ..., $G_n$
- Processes: $P_1$, ..., $P_n$
- Sorts (≈ types): $S_1$, ..., $S_n$
- Functions: $F_1$, ..., $F_n$
- Comments: (* ... *)

Caesar compiler of CADP [Garavel-Sifakis-90]
Value expressions and offers

Value expressions: \( V_1, \ldots, V_n \)
\[ V ::= X \]
\[ | \quad F (V_1, \ldots, V_n) \]
\[ | \quad V_1 \cdot F \cdot V_2 \]

Offers: \( O_1, \ldots, O_n \)
\[ O ::= ! V \quad \text{emission of a value} \ V \]
\[ | \quad ? X : S \quad \text{reception of a value to be stored in a variable} \ X \ \text{of sort} \ S \]
Behaviour expressions
(Lots Of Terribly Obscure Symbols :-) )

Behaviours: $B_1, \ldots, B_n$

$B ::= \text{stop}$

- $G_0 O_1 \ldots O_n [V] ; B_0$
- $B_1 [] B_2$
- $B_1 [G_1, \ldots, G_n] B_2$
- $B_1 [ ] [ ] B_2$
- hide $G_1, \ldots, G_n$ in $B_0$
- $[V] \rightarrow B_0$
- let $X : S = V$ in $B_0$
- choice $X : S [] B_0$
- $P [G_1, \ldots, G_n] (V_1, \ldots, V_n)$

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop</td>
<td>inaction</td>
</tr>
<tr>
<td>$G_0 O_1 \ldots O_n [V] ; B_0$</td>
<td>action prefix</td>
</tr>
<tr>
<td>$B_1 [] B_2$</td>
<td>choice</td>
</tr>
<tr>
<td>$B_1 [G_1, \ldots, G_n] B_2$</td>
<td>parallel with synchronization on $G_1, \ldots, G_n$</td>
</tr>
<tr>
<td>$B_1 [ ] [ ] B_2$</td>
<td>interleaving</td>
</tr>
<tr>
<td>hide $G_1, \ldots, G_n$ in $B_0$</td>
<td>hiding</td>
</tr>
<tr>
<td>$[V] \rightarrow B_0$</td>
<td>guard</td>
</tr>
<tr>
<td>let $X : S = V$ in $B_0$</td>
<td>variable definition</td>
</tr>
<tr>
<td>choice $X : S [] B_0$</td>
<td>choice over values</td>
</tr>
<tr>
<td>$P [G_1, \ldots, G_n] (V_1, \ldots, V_n)$</td>
<td>process call</td>
</tr>
</tbody>
</table>
Process definitions

\[
\text{process } P \left[ G_1, \ldots, G_n \right] (X_1:S_1, \ldots, X_n:S_n) := \\
B \\
\text{endproc}
\]

where:
- \( P \) = process name
- \( G_1, \ldots, G_n \) = formal gate parameters of \( P \)
- \( X_1, \ldots, X_n \) = formal value parameters of \( P \), of sorts \( S_1, \ldots, S_n \)
- \( B \) = body (behaviour) of \( P \)
Remarks

- LOTOS process: “black box” equipped with communication points (gates) with the outside

\[
\text{process } P \ [G_1, G_2, G_3] (...) := \\
\ldots \\
\text{endproc}
\]

- Each process has its own local (private) variables, which are not accessible from the outside

\[\Rightarrow \text{communication by rendezvous and not by shared variables}\]

- Parallel composition and encapsulation of boxes: described using the |[...]|, |||, and hide operators
Example

(Sender \[PUT, A, D\] ||| Receiver \[GET, B, C\])
\[A, B, C, D\]

or

(Medium1 \[A, B\] ||| Medium2 \[C, D\])

or

(Sender \[PUT, A, D\] \[A\] Medium1 \[A, B\])
\[B, D\]

(Receiver \[GET, B, C\] \[C\] Medium2 \[C, D\])
Multiple rendezvous

LOTOS parallel operators allow to specify the synchronization of \( n \geq 2 \) processes on the same gate.

Example (client-server):

\[
\begin{align*}
\text{C1} [A] & \mid [A] \mid \text{C2} [A] \mid [A] \mid \text{C3} [A] \\
& \mid [A] \mid \\
& \text{S} [A]
\end{align*}
\]

the three client processes synchronize with the server on gate A (4-way rendezvous)
Binary rendezvous

The `|||` operator allows to specify binary rendezvous (2 among $n$) on the same gate.

Example (client-server):

$$(C1 \ [A] \ |\ |\ |\ C2 \ [A] \ |\ |\ |\ C3 \ [A])$$

$$(\ |\ |\ A)$$

$$(S \ [A])$$

The three client processes are competing to access the server on gate A but only one can get access at a given moment.
Abstraction
(hiding)

In LOTOS, when a synchronization takes place on a gate $G$ between two processes, another one can also synchronize on $G$ (*maximal cooperation*)

If this is undesirable, it can be forbidden by hiding the gate (renaming it into $i$) using the `hide` operator:

$$\text{hide } G_1, \ldots, G_n \text{ in } B$$

which means that all actions performed by $B$ on gates $G_1, \ldots, G_n$ are hidden

The gates $G_1, \ldots, G_n$ are “abstracted away” (hidden from the outside world)
Example

process Network [PUT, GET] :=
    hide A, B, C, D in
    (Sender [PUT, A, D] ||| Receiver [GET, B, C])
    || [A, B, C, D]
    (Medium1 [A, B] ||| Medium2 [C, D])
endproc
Operational semantics

Notations:

- $G$: gate list (or set)
- $L$: action (transition label), of the form $G V_1, ..., V_n$
  where $G$ is a gate and $V_1, ..., V_n$ is the list of values exchanged on $G$ during the rendezvous
- $\text{gate} (L) = G$
- $B [ v / X ]$: syntactic substitution of all free occurrences of $X$ inside $B$ by a value $v$ (having the same sort as $X$)
- $V [ v / X ]$: idem, substitution of $X$ by $v$ in $V$
Semantics of “[[…]]”

\[
B_1 \rightarrow_{L} B_1' \land gate (L) \not\in G \\
\frac{B_1 \mid [G] \mid B_2 \rightarrow_{L} B_1' \mid [G] \mid B_2}{B_1 \text{ evolves}}
\]

\[
B_2 \rightarrow_{L} B_2' \land gate (L) \not\in G \\
\frac{B_1 \mid [G] \mid B_2 \rightarrow_{L} B_1 \mid [G] \mid B_2'}{B_2 \text{ evolves}}
\]

\[
B_1 \rightarrow_{L} B_1' \land B_2 \rightarrow_{L} B_2' \land gate (L) \in G \\
\frac{B_1 \mid [G] \mid B_2 \rightarrow_{L} B_1' \mid [G] \mid B_2'}{B_1 \text{ and } B_2 \text{ evolve}}
\]

Gates have no direction of communication
Semantics of “hide”

\[ B \rightarrow_L B' \land \text{gate} (L) \notin G \]

hide \( G \) in \( B \rightarrow_L \) hide \( G \) in \( B' \)

\[ B \rightarrow_L B' \land \text{gate} (L) \in G \]

hide \( G \) in \( B \rightarrow_i \) hide \( G \) in \( B' \)

In LOTOS, \( i \) is a keyword: use with care
Sequential behaviours

LOTOS allows to encode sequential automata by means of the choice (“[]”) and sequence operators (“;” and “stop”), and recursive processes.

```
process P [A, B, C, D, E]:
  noexit :=
  A; (B; stop [])
  C; (D; stop [])
  E; P [A, B, C, D, E]
endproc
```
Remarks

The description of automata in LOTOS is not far from regular expressions (operators “.”, “|”, “*”), except that:

- The “;” operator of LOTOS is asymmetric (≠ from “.”)
  \[ G \, O_1 \ldots \, O_n \, ; \, B \quad \text{but not} \quad B_1 \, ; \, B_2 \]
- There is no iteration operator “*”, one must use a recursive process call instead

LOTOS allows to describe automata with data values (≈ functions in sequential languages) by using processes with value parameters
Semantics of “stop”

The “stop” operator (inaction) has no associated semantic rule, because no transition can be derived from it.

A call of a “pathological” recursive process like:

```plaintext
process P [A] : noexit :=
    P [A]
endproc
```

has a behaviour equivalent to stop (unguarded recursion)
Prefix operator (";")

Allows to describe:
- Sequential composition of actions
- Communication (emission / reception) of data values

Simplest variant: actions on gates, without value-passing (basic LOTOS)

\[ a ; b ; c ; d ; \text{stop} \]
Semantics of “;”

Case 1: action without reception offers (?X:S)

\[
(\forall 1 \leq i \leq n . \ O_i \equiv ! V_i ) \land V = true \\
G \ O_1 \ldots O_n \ [ V ]; \ B \rightarrow_{G} V_1 \ldots V_n \ B
\]

- The boolean guard and the offers are optional
- If the guard \( V \) is false, the rendezvous does not happen (deadlock):

\[
G \ O_1 \ldots O_n \ [ V ]; \ B \approx \ stop
\]
Example (1/2)

Sequential composition:

A \text{!true}; B \text{!4}; \text{stop}

\begin{itemize}
\item A \text{!true}
\item B \text{!4}
\item \text{stop}
\end{itemize}
Example (2/2)

Synchronization by *value matching*: two processes send to each other the same values on a gate.

- \( G \downarrow 1; B_1 \| G \| G \downarrow 1; B_2 \)
- \( G \downarrow 1; B_1 \| G \| G \downarrow 2; B_2 \)
  - deadlock (different values)
- \( G \downarrow 1; B_1 \| G \| G \downarrow \text{true}; B_2 \)
  - deadlock (different types)
**Semantics of “;”**

**Case 2:** action containing reception offer(s) (?X:S)

\[
(v \in S) \land (V[v/X] = \text{true})
\]

\[
G ?X:S [ V ] ; B \rightarrow_{Gv} B [ v/X ]
\]

- The variables defined in the offers ?X:S are visible in the boolean guard V and inside B
- An action can freely mix emission and reception offers
Example (1/3)

\[ G \ ?X: \text{Bool}; \]
\[ \text{stop} \]

\[ G \ ?X: \text{Nat}[X < 4]; \]
\[ H ! X; \]
\[ \text{stop} \]

- The semantics handles the reception by branching on all possible values that can be received.
Example (2/3)

Emission of a value = guarded reception:
\[ G !V \equiv G ?X:S [ X = V ] \]
where \( S = \text{type} (V) \)

Synchronization by *value generation*: two processes receive values of the same type on a gate
\[ G ?n_1:Nat [ n_1 \leq 5 ]; B_1 \]
\[ | [ G ] | \]
\[ G ?n_2:Nat [ n_2 > 2 ]; B_2 \]
Example (3/3)

Synchronization by *value-passing*:

\[ G \ ?X: \text{Bool} \ ; \ \text{stop} \quad | [ G ] | \quad G \ !\text{true} \ ; \ \text{stop} \]

\[ G \ ?X: \text{Bool} \ ; \ \text{stop} \quad | [ G ] | \quad G \ !3 \ ; \ \text{stop} \]

---

deadlock: the semantics of the “| [...] |” operator requires that the two labels be identical (same type for the emitted value and the reception offer)
Rendezvous
(summary)

- General form:

\[ G \; O_1 \ldots \; O_m [V_1]; \; B_1 \mid [ \; G \; ] \mid \; G' \; O_1' \ldots \; O_n'[V_2]; \; B_2 \]

- Conditions for the rendezvous:
  - \( G = G' \) and \( G \in G \)
  - \( m = n \)
  - \( V_1 \) and \( V_2 \) are true in the context of \( O_1, \ldots, O_n' \)
  - \( \forall 1 \leq i \leq n. \; \text{type} \,(O_i) = \text{type} \,(O_i') \)
  - \( \forall 1 \leq i \leq n. \; \text{prop} \,(O_i) \cap \text{prop} \,(O_i') \neq \emptyset \)

where \( \text{prop}(O) = \text{set of values accepted by offer } O \)

- \( \text{prop} \,(!V) = \{ \; V \; \} \)
- \( \text{prop} \,(?X:S) = S \)
Choice operator (“[]”)

”[]”: notation inherited from the programs with guarded commands [Dijkstra]

Allows to specify the choice between several alternatives:

\((B_1 [] B_2 [] B_3)\)

can execute either \(B_1\), or \(B_2\), or \(B_3\)

Example:

\[
a ; \\
(b ; \text{stop} \\
[] \\
c ; \text{stop})
\]
Semantics of “[]”

\[ B_1 \rightarrow_L B_1' \]
\[ B_1 [] B_2 \rightarrow_L B_1' \]
\[ B_2 \rightarrow_L B_2' \]
\[ B_1 [] B_2 \rightarrow_L B_2' \]

After the choice, one of the two behaviours disappears (the execution was engaged on a branch of the choice and the other one is abandoned)
Internal / external choice

\((G_1 ; B_1 \ [\ ] \ G_2 ; B_2 )\)
- External choice: the environment can decide which branch will be executed
- Internal choice: the program decides

Example (coffee machine):

\[
\begin{align*}
\text{money} & \rightarrow \text{coffee} \rightarrow \text{money} \\
\text{coffee} & \rightarrow \text{tea} \\
\text{external choice (user)} \\
\text{money} & \rightarrow \text{coffee} \rightarrow \text{money} \\
\text{coffee} & \rightarrow \text{tea} \\
\text{internal choice (machine)}
\end{align*}
\]
Internal action ("i")

In LOTOS, the special gate $i$ denotes an internal event on which the environment cannot act:

$$(i; G_1; \text{stop}[]; i; G_2; \text{stop})$$

$$((i; G_1; \text{stop}[]; i; G_2; \text{stop})$$

$$((G_1; \text{stop}[]; i; G_2; \text{stop}))$$

$$GD_1$$

$$GD_2$$

$${\text{internal choice}}$$

$${\text{still internal choice}}$$
Guard operator ("[...] ->")

- LOTOS does not possess an “if-then-else” construct
- Guards (boolean conditions) can be used instead
- Informal semantics:

\[
[ V ] \rightarrow B \; \approx \; \text{if } V \text{ then } B \text{ else stop}
\]

- Frequent usage in conjunction with “[]”:

READ ?m,n:Nat ;

( [ m >= n ] -> PRINT !m; stop

[]

[ m < n ] -> PRINT !n; stop )

emission of max (m,n) on gate PRINT
Semantics of “[…] ->”

\[(V = \text{true}) \land B \rightarrow_L B'\]

\[\text{[ } V \text{ ] -> } B \rightarrow_L B'\]

If the boolean expression \(V\) evaluates to false, no semantic rule applies (deadlock):

\[\text{[ false ] -> } B \approx \text{ stop}\]
Examples

“if-then-else”:

\[
\begin{align*}
[ V ] & \rightarrow B_1 \\
[\ ] & \rightarrow B_2 \\
[ \text{not} (V) ] & \rightarrow B_2
\end{align*}
\]

“case”:

\[
\begin{align*}
[ X < 0 ] & \rightarrow B_1 \\
[\ ] & \rightarrow B_2 \\
[ X = 0 ] & \rightarrow B_2 \\
[ X > 0 ] & \rightarrow B_3
\end{align*}
\]

Beware of overlapping guards:

\[
\begin{align*}
[ X \leq 0 ] & \rightarrow B_1 \\
[\ ] & \rightarrow B_2 \\
[ X \geq 0 ] & \rightarrow B_2
\end{align*}
\]

If \( X = 0 \) then this is equivalent to an unguarded choice \( B_1 [\ ] B_2 \)
Operator “let”

- LOTOS allows to define variables for storing the results of expressions.

- Variable definition:
  
  ```
  let X:S = V in B
  
  declares variable X and initializes it with the value of V. X is visible in B.
  ```

- Write-once variables (no multiple assignments):
  
  ```
  let X:Bool = true  in G !X ; (* first X *)
  let X:Bool = false in G !X ; (* second X *)
  ```

stop
Semantics of “let”

\[ B [ V / X ] \rightarrow_{L} B' \]

\[ \text{let } X:S = V \text{ in } B \rightarrow_{L} B' \]

Example:

\[ \text{let } X:\text{NatList} = \text{cons} \,(0, \text{nil}) \text{ in } \]
\[ G !X; \]
\[ H !\text{cons} \,(1, X ); \]
\[ \text{stop} \]
Remarks

LOTOS is a *functional* language:

- No uninitialized variable (forbidden by the syntax)
- No assignment operator (":="), the value of a variable does not change after its initialization
- No "global" or "shared" variables between functions or processes
- Each process has its own local variables
- Communication by rendezvous only
- No side-effects
Operator “choice”

Operator “choice”: similar to “let”, except that variable $X$ takes a nondeterministic value in the domain of its sort $S$

Semantics:

$$\frac{(v \in S) \land B \ [ \ v / X \ ] \rightarrow L B'}{\text{choice } X : S [ ] B \rightarrow L B'}$$

Example:

```
choice X:Bool [ ]

G !X; stop
```

G false  G true
Examples

- Reception of a value = particular case of “choice”:
  \[ G ?X:S ; B \ = \ \text{choice } X:S [] B \]

- Iteration over the values of an enumerated type:
  \[
  \text{choice } A:\text{Addr} [] \\
  \text{SEND } !m !A \ ; \text{stop}
  \]

- Generation of a random value:
  \[
  \text{choice } rand:\text{Nat} [] \\
  [ \text{rand } \leq 10 ] \rightarrow \text{PRINT } !\text{rand} \ ; \text{stop}
  \]
Operator “exit”

LOTOS allows to express *normal termination* of a behaviour, possibly with the return of one or several values:

\[
\text{exit } (V_1, \ldots, V_n)
\]

denotes a behaviour that terminates and produces the values \(V_1, \ldots, V_n\).

Example:

\[
\text{REC } ?x : \text{Nat } [ x < 2 ] ; \\
\text{exit } (x + 1)
\]
Semantics of “exit”

true

\[
\text{exit } (V_1, \ldots, V_n) \rightarrow \text{exit } V_1 \ldots V_n \text{ stop}
\]

- \textit{exit} = special gate, synchronized by the “| [... ] |” operator (see later)
- The values \( V_1, \ldots, V_n \) are optional (“\textit{exit}” means normal termination without producing any value)
Operator “\( \gg \)”

LOTOS allows to express the sequential composition between a behaviour \( B_1 \) that terminates and a behaviour \( B_2 \) that begins:

\[
B_1 \gg \text{accept } X_1:S_1, \ldots, X_n:S_n \text{ in } B_2
\]

means that when \( B_1 \) terminates by producing values \( V_1, \ldots, V_n \), the execution continues with \( B_2 \) in which \( X_1, \ldots, X_n \) are replaced by the values \( V_1, \ldots, V_n \).

Example:

\[
\text{exit (1) } \gg \text{accept } n:\text{Nat in }
\]

\[
\text{PRINT } !n ; \text{ stop}
\]
Semantics of “>>”

\[
(B_1 \rightarrow_L B_1') \land \neg \text{gate (L)} \implies \neg \text{exit }
\]

\[
(B_1 >> \text{accept X:S in } B_2) \rightarrow_L (B_1' >> \text{accept X:S in } B_2)
\]

\[
B_1 \rightarrow_{\text{exit}} V B_1' \\
(B_1 >> \text{accept X:S in } B_2) \rightarrow_i B_2 [V/X]
\]

- The \_V\_ values must belong pairwise to the \_S\_ sorts
- The exit gate is hidden (renamed into \_i\_) when sequential composition takes place
- The “>>” operator is also called enabling (\_B_2’s execution is made possible by \_B_1’s termination)
Example (1/4)

Sequential composition without value-passing:

\[(\text{In1; In2; exit}
\quad[]
\quad\text{In2; In1; exit})
\]

\[
\gg
\]

\[(\text{Access; exit})
\]

\[
\gg
\]

\[(\text{Out1; Out2; stop}
\quad[]
\quad\text{Out2; Out1; stop})
\]
Example (2/4)

Sequential composition with value-passing:

```plaintext
READ ?m,n:Nat ;
( [ m >= n ] -> exit (m)
  []
  [ m < n ] -> exit (n) )

>>

accept max:Nat in
PRINT !max ; stop
```
Example (3/4)

Definition of terminating process:

```
process Login [LogReq, LogConf, LogAbort] : exit :=
    LogReq;
    ( i ; LogConf ; exit
        []
            i ; LogAbort ; Login [LogReq, LogConf, LogAbort])
endproc
```

Example of call:

```
Login [Req,Conf,Abort] >> Transfer ; Logout ; stop
```
Example (4/4)

Combination of “exit” and parallel composition: the two behaviours are synchronized on the exit gate (they terminate simultaneously)

( a ; b ; exit ) ||| ( c ; exit )
In LOTOS, difference between
“;” (asymmetric)
and
“$$\gg$$” (symmetric):
Process call

Let a process $P$ defined by:

\[
\text{process } P \left[ G_1, \ldots, G_n \right] \ (X_1:S_1, \ldots, X_n:S_n) := B \\
\text{endproc}
\]

Semantics of a call to $P$:

\[
B \left[ g_1 / G_1, \ldots, g_n / G_n \right] \left[ v_1 / X_1, \ldots, v_n / X_n \right] \rightarrow_L B'
\]

\[
P \left[ g_1, \ldots, g_n \right] \left( v_1, \ldots, v_n \right) \rightarrow_L B'
\]

This semantics explains why a call to

\[
\text{process } P[G] : \text{noexit} := P[G] \text{ endproc}
\]

is equivalent to $\text{stop}$. 
Example

Boolean variable:

```
VAR [READ, WRITE] (b:Bool) :
  noexit :=
  READ !b;
  VAR [READ, WRITE] (b)
  []
  WRITE ?b2:Bool;
  VAR [READ, WRITE] (b2)
endproc
```
Static semantics
(summary)

Scope of variables inside behaviours:

\[ B ::= G !V_0 ?X:S \ldots [ V ] ; B_0 \]

- \( p (X) = \{ V, B_0 \} \)
- \( p (G) = \{ B_0 \} \)
- \( p (X) = \{ B_0 \} \)
- \( p (X) = \{ B_0 \} \)
- \( p (X) = \{ B_0 \} \)

Scope of process parameters:

process P [G] (X:S) :=

\[ B_0 \]

endproc

\[ p (G) = \{ B_0 \} \]

\[ p (X) = \{ B_0 \} \]
LOTOS specification

A LOTOS specification is similar to a process definition:

```
specification Protocol [ SEND, RECEIVE ] : noexit :=
   (* ... type definitions *)

   behaviour
      (* ... behaviour = body of the specification *)

   where
      (* ... process definitions *)

endspec
```
Example: Peterson’s mutual exclusion algorithm

\[
\begin{align*}
&\text{var } d_0 : \text{bool} := \text{false} \quad \{ \text{read by P1, written by P0} \} \\
&\text{var } d_1 : \text{bool} := \text{false} \quad \{ \text{read by P0, written by P1} \} \\
&\text{var } t \in \{0, 1\} := 0 \quad \{ \text{read/written by P0 and P1} \}
\end{align*}
\]

\[
\begin{align*}
&\text{loop forever} \{ \text{P0} \} \\
1 : &\{ \text{ncs0} \} \\
2 : &d_0 := \text{true} \\
3 : &t := 0 \\
4 : &\textbf{wait} (d_1 = \text{false} \text{ or } t = 1) \\
5 : &\{ \text{cs0} \} \\
6 : &d_0 := \text{false} \\
\text{endloop}
\end{align*}
\]

\[
\begin{align*}
&\text{loop forever} \{ \text{P1} \} \\
1 : &\{ \text{ncs1} \} \\
2 : &d_1 := \text{true} \\
3 : &t := 1 \\
4 : &\textbf{wait} (d_0 = \text{false} \text{ or } t = 0) \\
5 : &\{ \text{cs1} \} \\
6 : &d_1 := \text{false} \\
\text{endloop}
\end{align*}
\]
Description of variables d0, d1

- Each variable: instance of the same process D
- Current value of the variable: parameter of D
- Reading and writing: RdV on gates R et W

\[
\begin{align*}
\text{process } & D[R, W] (b: \text{Bool}) : \text{noexit} := \\
& R !b ; D[R, W] (b) \\
& [] \\
& W ?b2: \text{Bool} ; D[R, W] (b2) \\
\text{endproc}
\end{align*}
\]

\[d0 \equiv D[R0, W0] (\text{false}), \ d1 \equiv D[R1, W1] (\text{false})\]
Description of variable t

- Variable t: instance of process T
- Current value of the variable: parameter of T
- Reading and writing: RdV on gates R et W

```plaintext
process T [R, W] (n:Nat) : noexit :=
R !n ; T [R, W] (n)
[]
W ?n2:Bool ; T [R, W] (n2)
endproc
```

\[ t \equiv T [RT, WT] (0) \]
Description of processes P0 and P1

- Process $P_m$: instance of the same process $P$
- Index $m$ of the process: parameter of $P$

```
process P [Rm, Wm, Rn, Wn, RT, WT, NCS, CS]  
  (m: Nat) : noexit :=  
  NCS !m ; Wm !true ; WT !m ;  
  P2 [Rm, Wm, Rn, Wn, RT, WT, NCS, CS] (m)  
endproc
```

- $P0 \equiv P [R0, W0, R1, W1, RT, WT, NCS, CS] (0)$
- $P1 \equiv P [R1, W1, R0, W0, RT, WT, NCS, CS] (1)$
Auxiliary process to describe busy waiting:

```markdown
process P2 [Rm, Wm, Rn, Wn, RT, WT, NCS, CS] (m:Nat) : noexit :=

Rn ?dn:Bool ; RT ?t:Nat ;
( [ dn and (t eq m) ] ->
  P2 [Rm, Wm, Rn, Wn, RT, WT, NCS, CS] (m)
  []
  [ not (dn) or (t eq ((m + 1) mod 2)) ] ->
    CS !m ; Wn !false ;
    P [Rm, Wm, Rn, Wn, RT, WT, NCS, CS] (m) )
endproc
```
Architecture of the system
(graphical)
Architecture of the system
(textual)

hide R0, W0, R1, W1, RT, WT in

( ( P [R0, W0, R1, W1, RT, WT, NCS, CS] (0) ||| P [R1, W1, R0, W0, RT, WT, NCS, CS] (1) )
  ||| [ R0, W0, R1, W1, RT, WT ]
  ( T [RT, WT] (0) ||| D [R0, W0] (false) ||| D [R1, W1] (false) ) )
LTS model

- 55 states
- 110 transitions
Process algebraic languages
(summary)

More concise than communicating automata: process parameterization, value-passing communication (Exercise: model variables d0, d1, t using a single gate allowing both reading / writing)

In general, there are several ways of describing the parallel composition of processes (Exercise: write a different expression for the architecture of Peterson’s algorithm)

Modeling of nested loops: mutually recursive LOTOS processes (Exercise: model processes P0, P1 using a single LOTOS process)

But: E-LOTOS process part is much more convenient