Chapitre 6 : Verification of LOTOS specifications

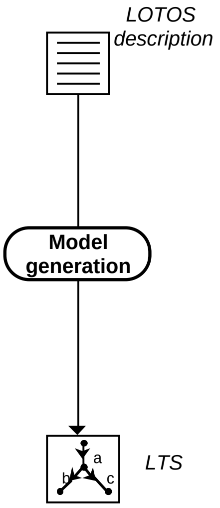
Verification is usually not performed at the source LOTOS level, but on some underlying **model** of the LOTOS specification.

The verification process is thus called **model-based** and is composed of two steps:

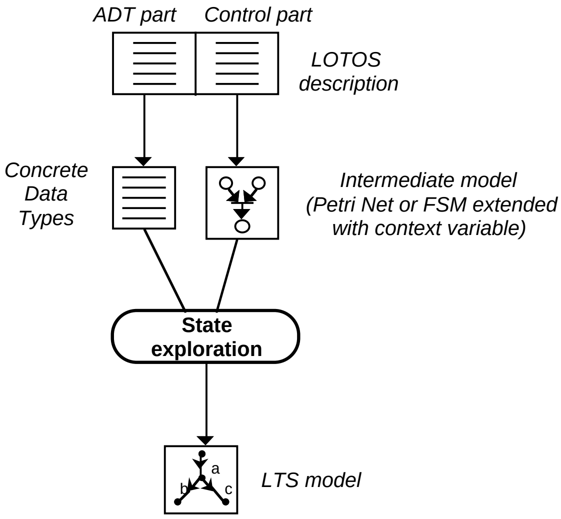
- the generation of a model
- the actual verification of the model by behavioural equivalence (or preorder) checking

Model generation

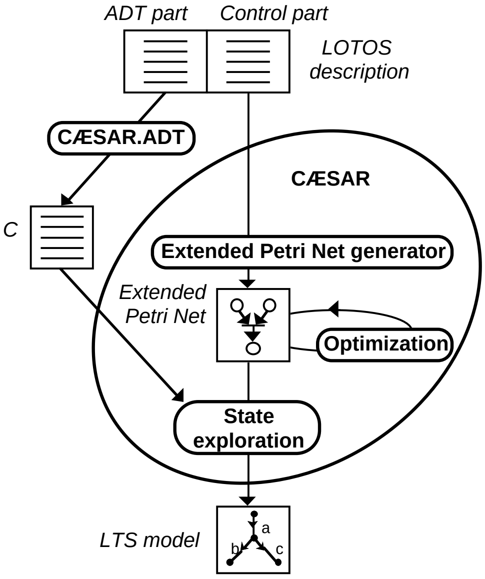
Abstract view



Detailed view of most tools



The CÆSAR LTS generator



Step 1: Generation of an extended Petri Net

The supported subset of LOTOS is workable

(i.e. finite control : no process recursion on the left and right part of |[…]|, nor on the left part of >> and [>.

Possible **explosion** due to the complexity of the control structure

Step 2: Optimization of the Petri Net

Step 3: **Generation of the LTS**

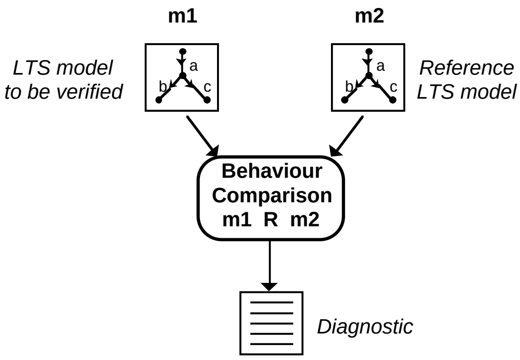
a) Generation of a C program (modelling the Petri Net)

b) Exhaustive simulation : execution of the C program (+ C code from CÆSAR.ADT) generates the LTS

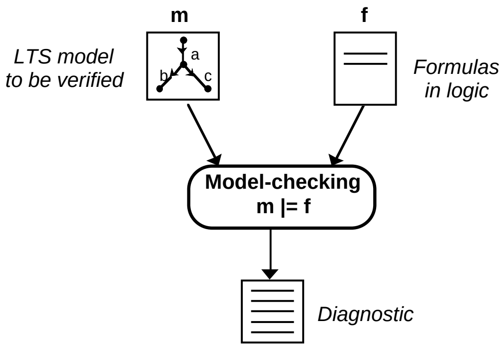
Possible state explosion due to large ranges of data values

Model-based verification methods

Behavioural comparison



Model-checking



The behavioural comparison is based on formal relations **R** between LTS:

Equivalence relations

Preorder relations

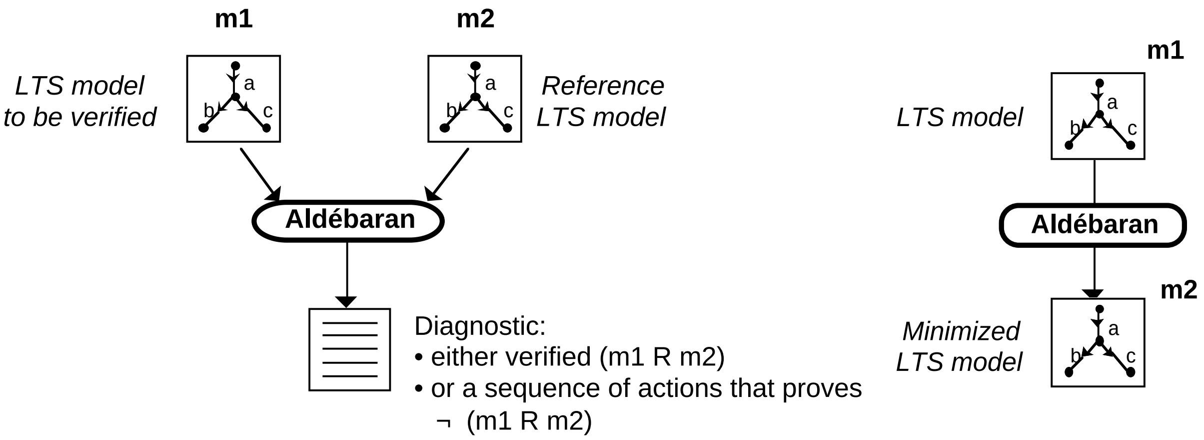
Various forms of logics may be used (e.g. temporal logic)

The verification method is based on a decision procedure for a **satisfaction** relation

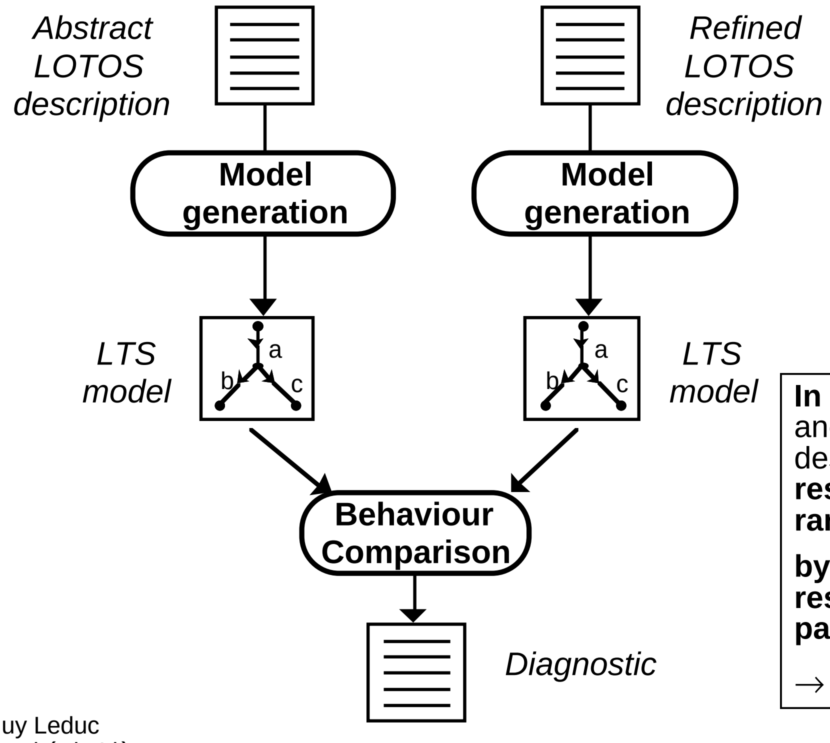
Model-based verification with Aldébaran

Behavioural comparison
Checks whether m1 R m2

Minimization modulo
an equivalence R
m1 R m2



Verification by behavioural comparison



In practice, both the abstract and the refined LOTOS descriptions have to be **restricted to limited data value ranges:**

by adding a new process (a restricted environment) in parallel

→ **non exhaustive verification**

Reference LTS as a property

A **reference LTS** has to be produced prior to verification.

Up to now we have more or less implicitly considered that this reference LTS is generated automatically from a reference LOTOS specification.

There is however **another way to use Aldebaran**.

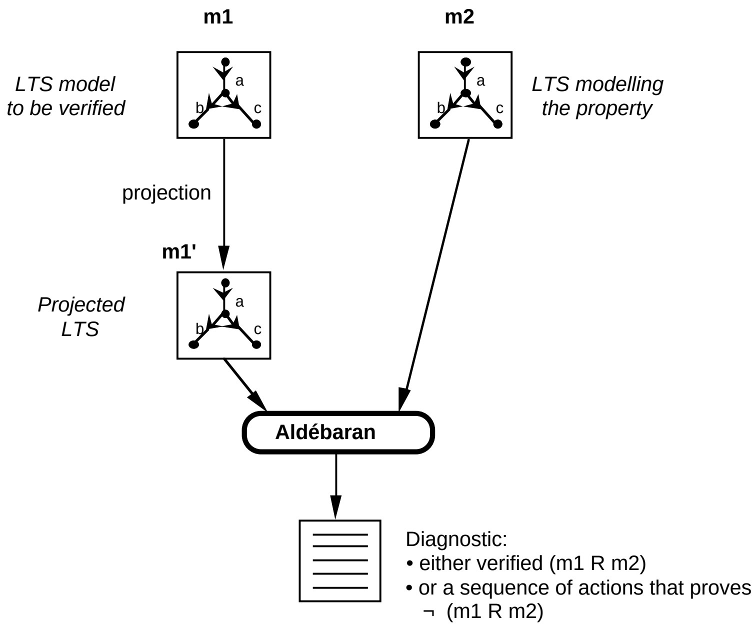
If the verifier is only interested in a **specific property** of its specification, it is usually easier to produce **a very simple LTS that models this property**, and then check whether the specification fulfills it.

However, as the problem is stated above, Aldebaran cannot be used because it is very unlikely that the generated LTS and the LTS modelling the property, be equivalent or related by any preorder.

This is because **the LTS modelling the property is limited to a very small subset of actions**: only those that are necessary to express the property.

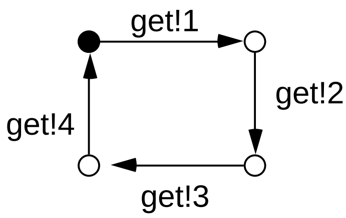
It is therefore necessary to use a **projection technique** (composed of hiding and/or renaming of actions) on the generated LTS prior to actual verification with respect to the property.

Projection of a LTS for verification

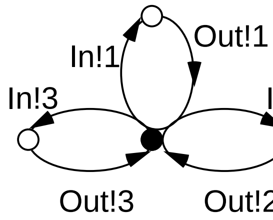


Modelling properties by LTS

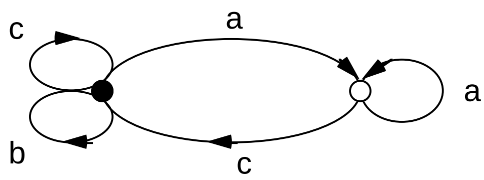
Event ordering : reception of messages in a given order
Suppose:
• four messages 1, 2, 3 and 4 have been sent, and
• get!i means reception of message i, and
• we only consider get!i actions



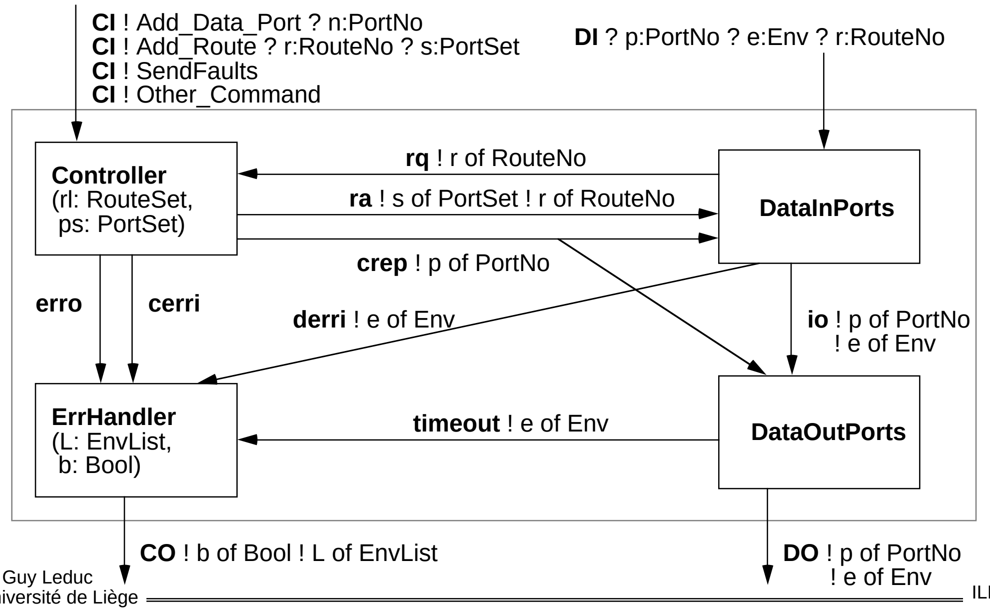
Mutual exclusion
Suppose:
• we have 3 processes, and
• In!i means process i enters the critical section, and
• Out!i means process i exits the critical section, and
• we only consider In!i and Out!i actions



Not a to b unless c
That is: action c must occur between action a and b
If we only consider actions a, b and c



Case-study: a Transit-Node



Data messages (called **Envelopes**) enter the node at **DI** and exit the node at **DO** if they do not become faulty. Faulty data messages exit the node at **CO**.

Every data message is input at a specific **port** and associated with a specific **route**. A route consists of a set of output ports. For every message received, the node will select nondeterministically an output port associated with the route, and output the message at this port. Routes and their associated output ports are stored in the controller.

Control messages enter the node at **CI.**

Control messages at **CI** are used to add a port or a route in the node. When a new port is created, the controller sends at **crep** (create port) the new port number to the processes responsible for the data transfer.

When a data message is received on a route, the DataInPorts sends a request at **rq** to the controller which replies by sending at **ra** the set of output ports associated with this route. One output port is selected nondeterministically if the set is not empty, and the message will follow the path **io** and then **DO**. If the set is empty, a faulty data message is sent to the error handler via **derri**. Also, if a buffered message remains in DataOutPorts more than T time units, it will become faulty and sent to the error handler via **timeout**.

When the controller receives a Send_Faults command at **CI**, it sends a message at **erro** to inform the error handler to output all the buffered faulty messages at **CO**.

When an erroneous command is received at **CI** (modelled as other_command), the controller sends a faulty control message at **cerri**.

The error handler buffers the faulty data messages from **derri** and **timeout** in a list, and registers the reception of faulty control messages from **cerri** by setting a boolean value to **true**. When the error handler receives a message at **erro**, it outputs at **CO** the list of faulty data messages and the boolean value. It also sets back the boolean value to false.

Modelling a suitable environment to control state explosion

We must first specify **an environment process** that will:

- feed the Transit-Node with control and data messages
- ensure the **finiteness** (and acceptable size) of the LTS model **while keeping as much as possible of the interesting behaviour of the node**

This requires that:

1) the data domain associated with each message field must be **finite**
2) the number of copies of each data message in the node must be **finite**, due to the storage in the node.

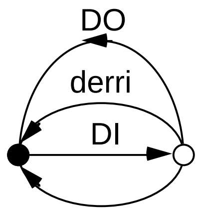
Concretely, **requirement 1** is fulfilled if we have:

- Finite number of ports (already bounded by N in the spec)
- Finite number of routes (implied by the finiteness of the number of ports)
- Finite number of distinct sets of ports and routes (also implied as above)
- **Finite number of distinct messages**

Requirement 2 is fulfilled if for example we write an environment that ensures that there is a single copy of each message in the node. An easy way to do this is to write an environment that keeps track of the messages that have been input and not yet output.

Example of a safety property

Property : possible history of a given message



Any received data message (**DI**) will have the ability to exit the node (**DO**) or to become faulty (**derri, timeout**)

Note that only four gates are concerned by this property.

And that this property refers to some **internal** gates.

Verification procedure:

1. Encode the LTS modelling the property (either directly or via a LOTOS process)
2. Project the Transit-Node spec. on the limited visible alphabet DI, DO, derri, timeout.

   More precisely, hide all actions except actions DI!…, DO!…, derri!…, timeout!… for a **particular** message, say 0. Then rename those actions by removing the useless attributes.
3. Minimize the obtained LTS modulo the **safety equivalence** (in practice, one first minimizes with the strong bisimulation equivalence)
4. Compare the minimized LTS with the LTS modelling the property w.r.t. the safety equivalence

Example of a more complex property

A property like:

> On a Send_Faults message reception at **CI**, some of the faulty messages buffered so far must exit the node, through **CO**.

can be split into a liveness and a safety property.

Liveness: Each Send_Faults request is always followed by an emission at CO.

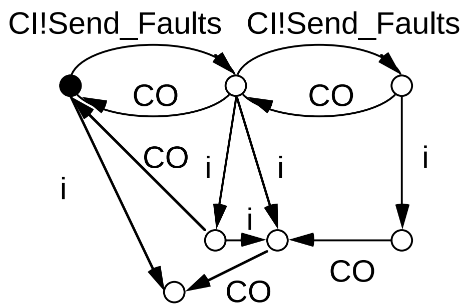
Safety: Messages emitted at CO are exactly messages previously buffered as faulty.

Liveness part of the property

Property: Each Send_Faults request is always followed by an emission at CO

Verification procedure:

1. Compute the minimization of the Transit-Node modulo **branching bisimulation** when all actions are hidden except CI!Send_Faults and CO (without atttributes). We get the LTS:



This quotient LTS is small enough to check that, for all execution sequence, each occurrence of a CI!Send_Faults action is eventually followed (later in the sequence) by an occurrence of a CO action.

The sink state is due to the chosen environment.

2. We must check the absence of divergence in the original LTS, because they are not preserved by the minimization modulo branching bisimulation, but may compromise liveness properties (unfairness of divergences)

Safety part of the property

Property: Messages emitted at CO are exactly messages previously buffered as faulty.

Again this property may be split into two simpler properties: one on control messages and one on data messages.

Verification procedure for the control messages:

1. **Rephrase** the property in terms of execution sequences in the LTS:

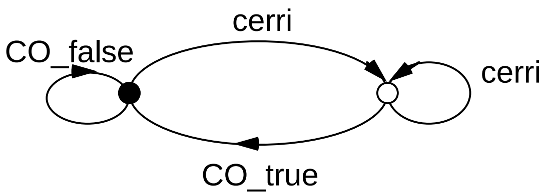
   • a **cerri** action cannot be followed by a **CO!false!L** action (i.e.some faulty control messages buffered as faulty must leave the node on request)

   • two successive occurrences of a **CO!true!L** action must be separated by an occurrence of a **cerri** action (i.e. no faulty control message can still be buffered after a CO!true!L action)

2. **Encode** the LTS modelling the property:

   where CO_false = any CO!false!L

   and CO_true = any CO!true!L



3. In the LTS of the Transit_Node, **hide** all actions except actions **cerri** and **CO!…** and **rename** all CO!false!L actions as CO_false and all CO!true!L actions as CO_true.

4. **Minimize** this LTS modulo the safety equivalence, and **compare** the result with the LTS modelling the property w.r.t. the **safety preorder**

Improvement of model-based verification

Problem: the LTS is too large to be generated

Solution : **compositional verification**

1. Find a good partition of the system into subprocesses
   - Rule: keep together strongly synchronized processes, otherwise the state space of a subprocess is larger than the state space of the combination
2. Generate the LTS of those subprocesses
3. Minimize each LTS using the strong bisimulation equivalence (~)
4. Minimize them further depending on the properties to be verified
   - branching bisimulation if liveness properties
   - safety equivalence if safety properties
5. Recombine the minimized LTS to generate the global LTS
   - This works because all the above equivalences are congruences
   - Aldebaran allows the generation of a LTS from other LTS provided that the combination rules are specified in a special file.