Compilation and Verification of LOTOS Specifications

Hubert Garavel

Centre d'Etudes Rhône-Alpes
VERILOG
Grenoble, France

Joseph Sifakis

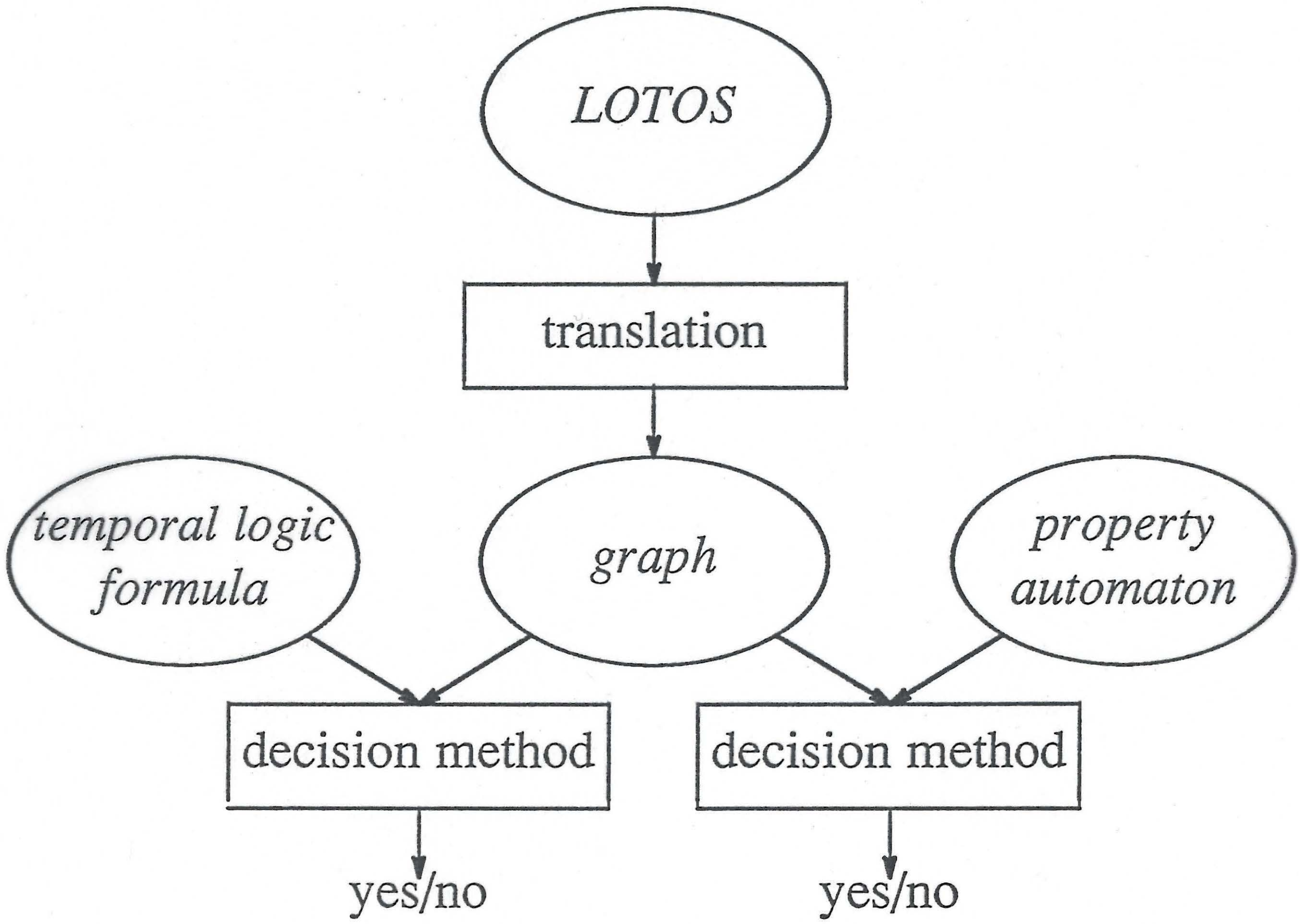
Laboratoire de Génie Informatique
Institut I.M.A.G.
Grenoble, France

Motivation

Verification: comparison of a LOTOS program against requirements.

Two approaches:

- **theorem proving:** (Boyer-Moore, LCF, ...)
- **model checking:**
  - step 1: translation LOTOS → finite state model (**graph**)
  - step 2: verification of requirements on the model



Theorem Proving vs. Model Checking

	theorem proving	model checking
analysis level	source-level	graph-level
symbolic evaluation	yes	no
full automation	no	yes
generality	yes	no
efficiency	no	yes

Model checking is less general but more efficient

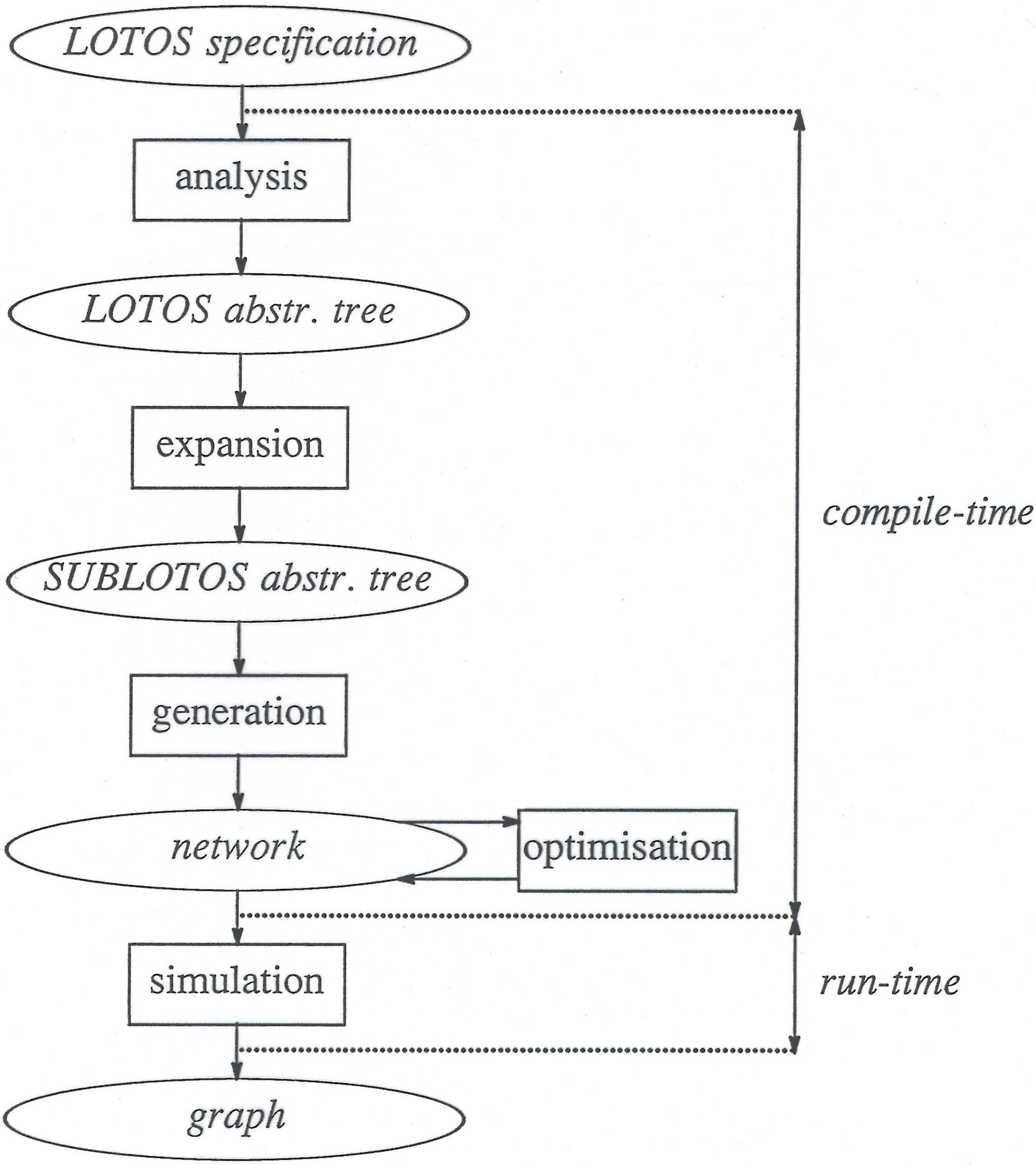
Given $\begin{cases} \text{a requirement } R \\ \text{a LOTOS specification represented by a graph } G \end{cases}$

	theorem proving	model checking
R undecidable	theoretically impossible	theoretically impossible
R decidable G infinite	theoretically possible practically not efficient	theoretically impossible
[R decidable] G finite $\|G\| > 10^6$–10^7 states	theoretically possible practically not efficient	practically impossible
[R decidable] G finite $\|G\| \leq 10^6$–10^7 states	theoretically possible practically not efficient	**possible and efficient**

Compilation vs. Interpretation

- **verification by model checking**
- problem: **efficient** translation LOTOS → graph
- two solutions:
  - **interpretation scheme** (LOTOS simulators)
    direct implementation of LOTOS dynamic semantics rules
  - **compilation scheme** (CÆSAR)
    implementation of an Extended Petri Net semantics

interpretation scheme	CÆSAR compilation scheme
direct translation LOTOS → graph	stepwise translation LOTOS → ... → ... → graph
no intermediate form	two intermediate forms: SUBLOTOS and networks
only a run-time phase	compile-time and run-time phases
computations performed several times, at each step	computations performed only once, at compile-time
states = LOTOS terms ⟹ high cost in memory	states = compact bit strings (position of control + values of variables)
transitions ← term rewriting ⟹ high cost in time	transitions ← Petri Net rules (use of a static control skeleton)


LOTOS specification
analysis
LOTOS abstr. tree
expansion
SUBLOTOS abstr. tree
generation
network
optimisation
simulation
graph
compile-time
run-time

Static Control Constraints

Restriction to a subset of LOTOS

- recursion is not allowed on the left or right side of "|[...]|"

```
process P [...]  ...
   ...  ||| P [...]
endproc
```

- recursion is not allowed on the left side of ">>" or "[>"

Also:

- process instantiation with identical gate parameters:

```
P [..., G, ..., G, ...]  (...)
```

  is handled differently than in the ISO semantics of LOTOS

- abstract data types must be implemented by concrete types

Reasons

In this subset of LOTOS:

- all specifications have a finite state control skeleton
- expressiveness is still sufficient for protocols

A good solution to the **expressiveness vs. efficiency** problem.

Expansion: from Lotos to SubLotos

SubLotos = subset of Lotos obtained by syntactic transformations

- elimination of Lotos "macro"-operators: >>, **exit**, **choice**, **par**

<table>
<tr><td><code>exit (V) >> accept X:S in B</code></td><td><code>hide δ in
  (δ !V; stop
  |[δ]|
  δ ?X:S; B)</code></td></tr>
<tr><td><code>choice G in [G1, G2] [] P [G]</code></td><td><code>P [G1] [] P [G2]</code></td></tr>
<tr><td><code>par G in [G1, G2] ||| P [G]</code></td><td><code>P [G1] ||| P [G2]</code></td></tr>
</table>

- recursion development to have "constant" gates

<table>
<tr><td><code>process P [G1, G2] ...
  G1; P [G2, G1]
endproc</code></td><td><code>process P [G1, G2] ...
  G1; G2; P [G1, G2]
endproc</code></td></tr>
</table>

- renaming of gates, variables and processes

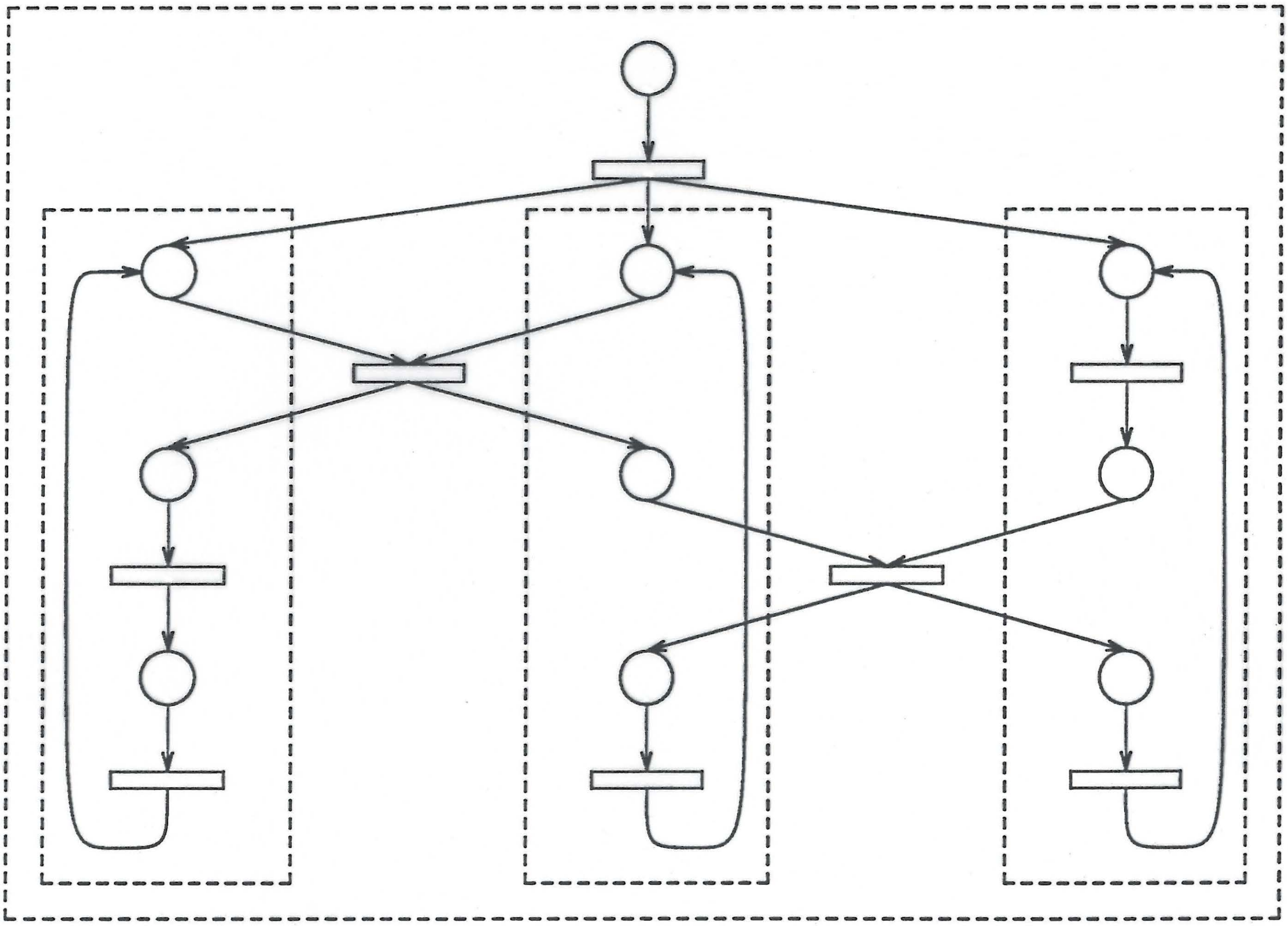
Static control constraints $\Longrightarrow$ SubLotos is an imperative language.

Lotos	SubLotos (and networks)
dynamic architecture • dynamic creation/deletion of processes • dynamic creation/deletion of gates • dynamic creation/deletion of variables • gates with "variable" value	**static architecture** • static set of processes • static set of gates • static set of variables • gates with "constant" value
functional features • dynamic constants • single assignment • local scope	**imperative features** • static variables • multiple assignment • global scope

The Network Model

The Control Part

- a set of **places**
- a set of **transitions**, with the following attributes
  - a set of **input places**
  - a set of **output places**
  - a **gate** (visible, "τ", or "ε")
  - a list of **offers** ("$!V$" or "$?X:S$")
- a hierarchical refinement into **units** (sequential behaviors)



The Data Part

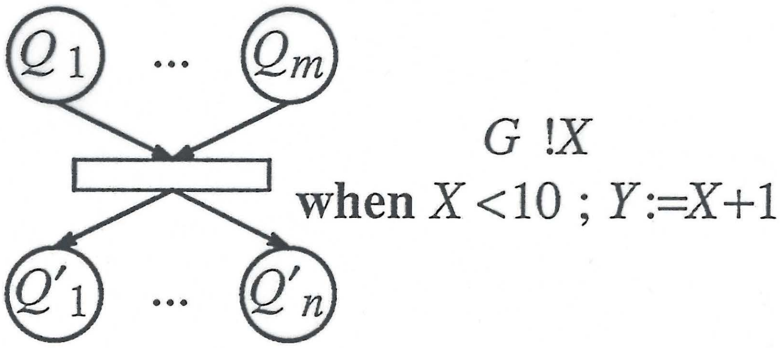
- a set of **variables**
- **actions** attached to transitions:
  - assignments: $X := X + 1$
  - conditions: **when** $X > 0$
  - iterations: **for** X **among** $BOOL$

The Network Model

Operational semantics

- **translation network → graph**
- **state = ⟨marking, context⟩**
  - marking = set of marked places (control part)
  - context = values of variables (data part)
- **transition relation:** $\textbf{state}_1 \xrightarrow{\textbf{gate offers}} \textbf{state}_2$
  - wrt to markings: Petri Net rules
  - wrt to contexts: execution of the action

Example:



$$\underbrace{\langle\{Q_1, ...Q_m\}, \{X=0, Y=0\}\rangle}_{\text{state}_1} \xrightarrow{G\ !0} \underbrace{\langle\{Q'_1, ...Q'_n\}, \{X=0, Y=1\}\rangle}_{\text{state}_2}$$

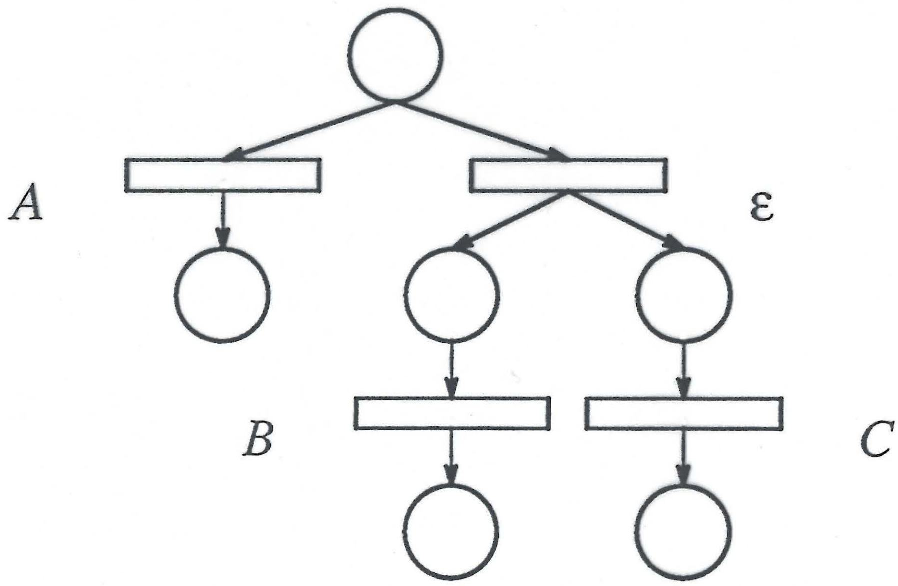
ε-transitions

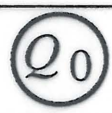
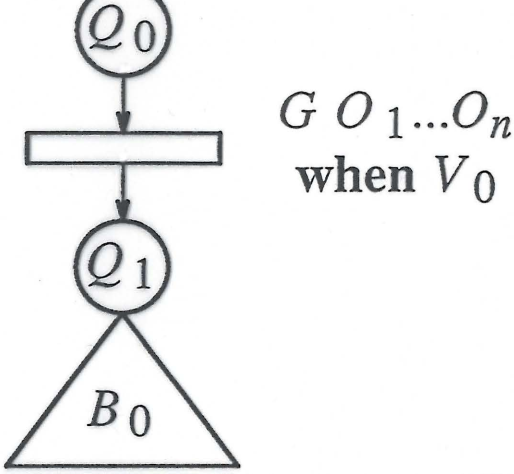
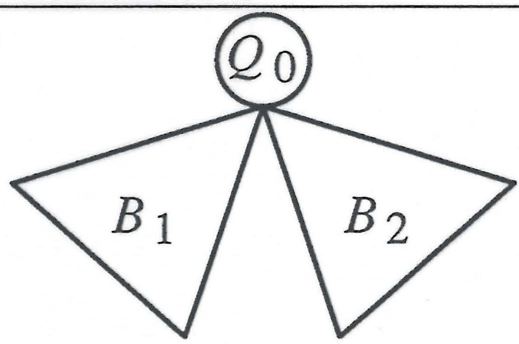
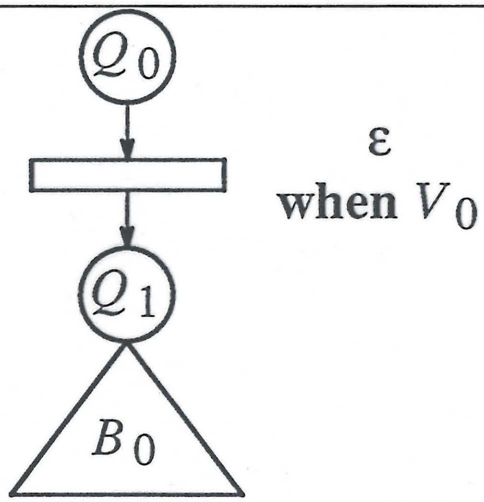
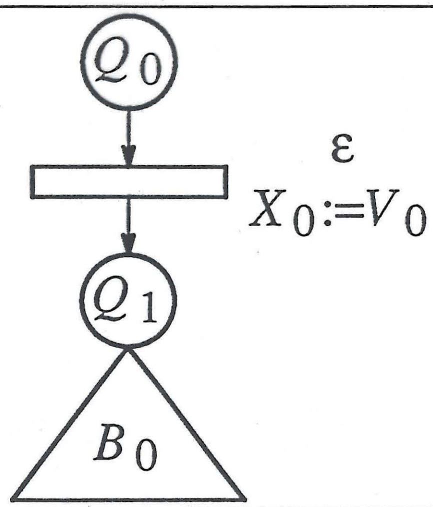
- representation of instantaneous silent events
- compositional construction of the network
- semantics:

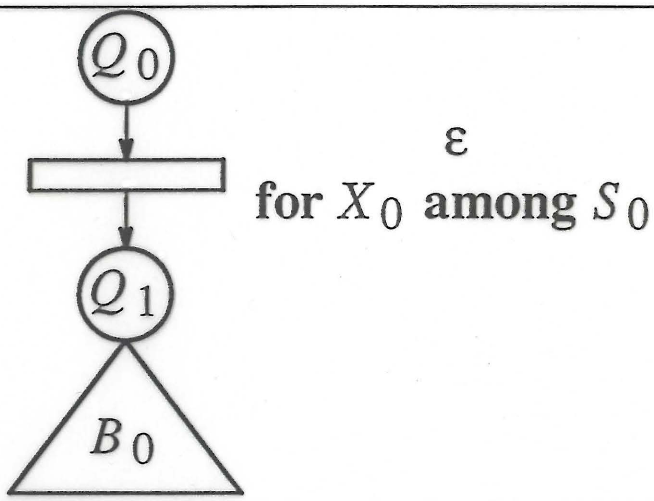
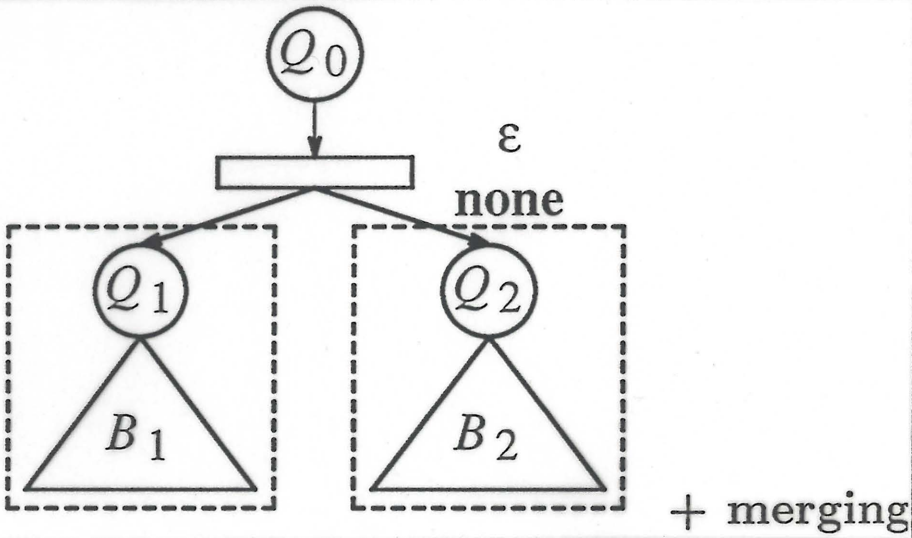
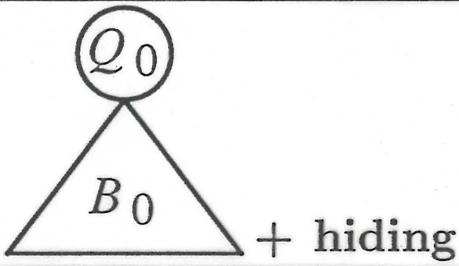
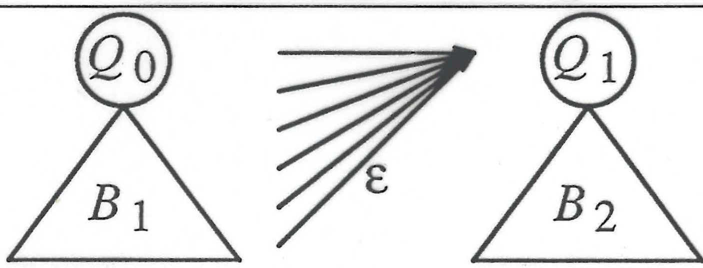
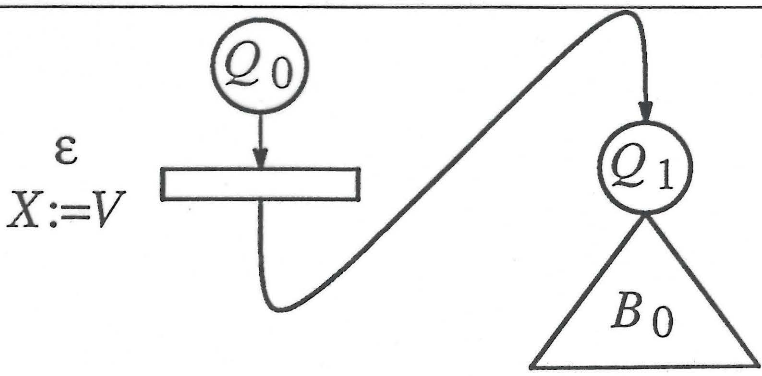
$$\begin{cases} \text{closure algorithm } (\sim \text{ automata theory}) \\ + \text{ atomicity rule} \end{cases}$$

Example:

$$A;\ \mathbf{stop}\ [\,]\ (B;\ \mathbf{stop}\ |||\ C;\ \mathbf{stop})$$

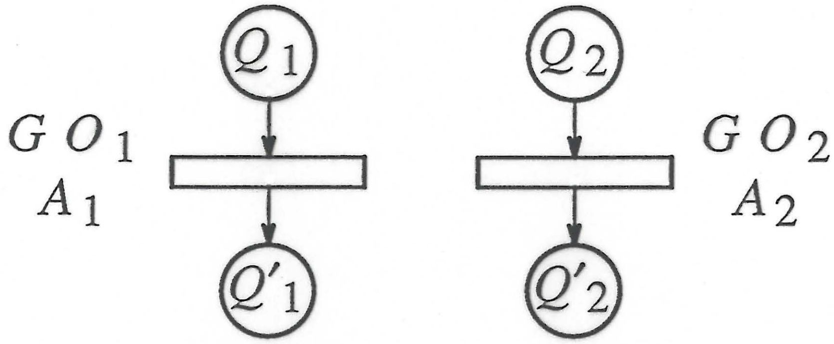


<table>
<tr><td>stop</td><td>
</td></tr>
<tr><td>$G\ O_1, ... O_n$ [V_0] ; B_0</td><td>
</td></tr>
<tr><td>B_1 [] B_2</td><td>
</td></tr>
<tr><td>[V_0] -> B_0</td><td>
</td></tr>
<tr><td>let $X_0 : S_0 = V_0$ in B_0</td><td>
</td></tr>
</table>

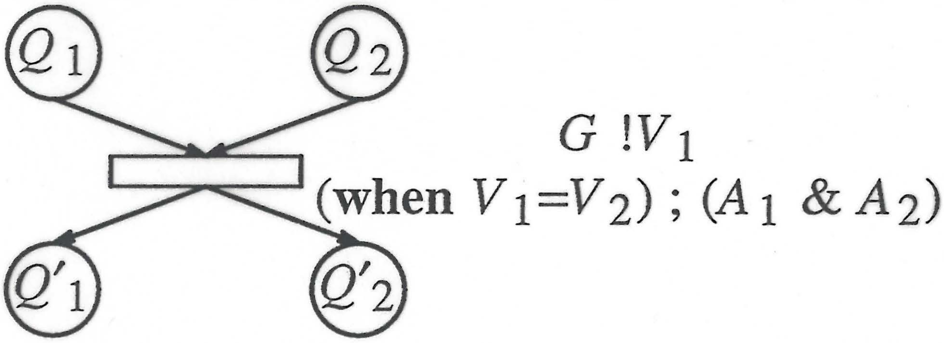
choice $X_0 : S_0$ [] B_0	
B_1 |[$G_0, \ldots G_n$]| B_2	
hide $G_0, \ldots G_n$ in B_0	
B_1 [> B_2	
P [...] (V) where process P [...] ($X : S$) B_0 endproc	

Generation: from SubLotos to Network

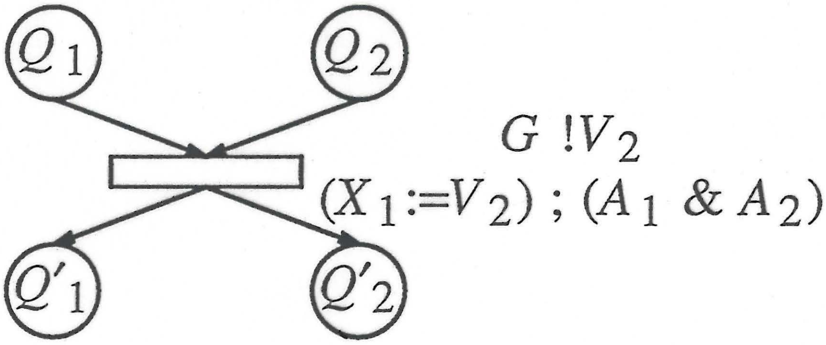
Parallel composition: rules for transition merging



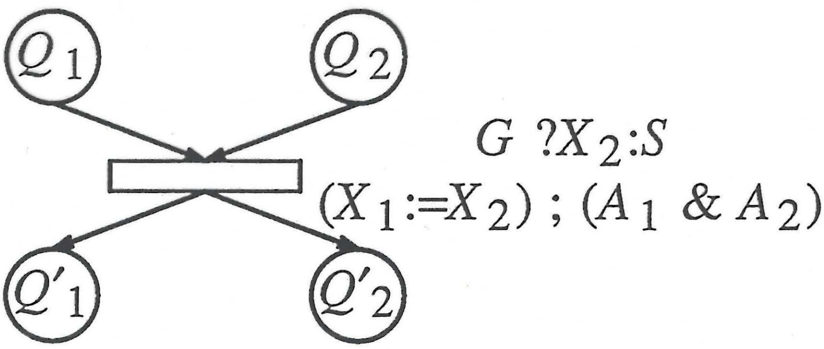
- **value matching:** $O_1 = !V_1$ and $O_2 = !V_2$



- **value passing:** $O_1 = ?X_1 : S_1$ and $O_2 = !V_2$



- **value generation:** $O_1 = ?X_1 : S_1$ and $O_2 = ?X_2 : S_2$



Optimization: Reductions of Networks

Reducing networks improves the efficiency of the simulation phase.

A set of optimizing transformations:

- based on static analysis techniques
- preserving strong equivalence
- fast and effective

Optimization of the control part

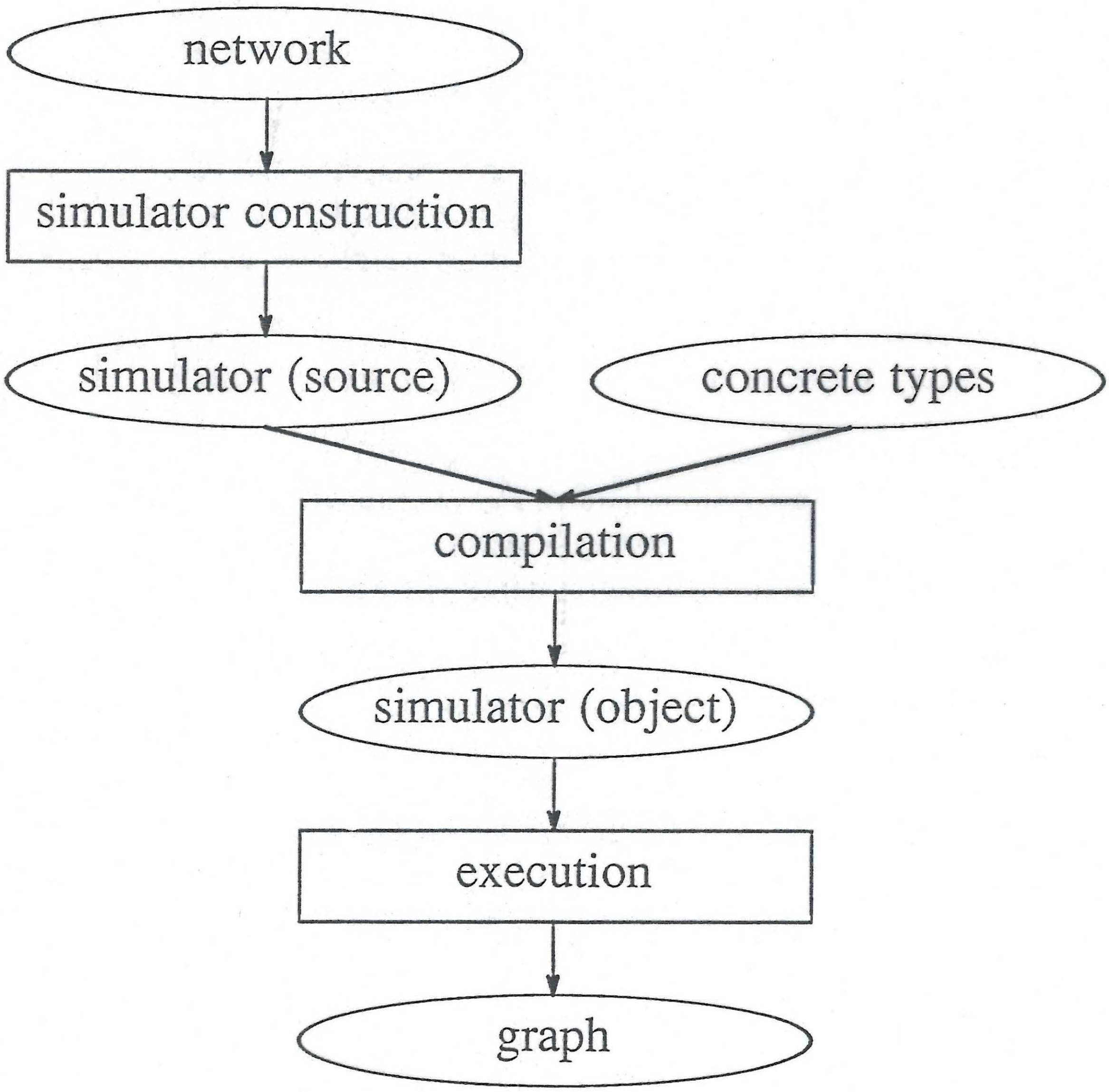
- **based on (local) Petri Net analysis techniques**
  - removing non reachable places/transitions
  - removing non productive places/transitions
  - removing places Q' such that $(\exists Q)\ Q\ \mathit{marked} \iff Q'\ \mathit{marked}$
  - eliminating many ε-transitions

Optimization of the data part

- **based on (global) data-flow analysis techniques**
  - removing variables never used
  - removing assignments of the form $X := X$
  - removing variables X' such that $(\exists X)\ X = X'$
  - discovering variables with constant values
  - evaluating constant boolean guards

Simulation: from Network to Graph

- breadth-first graph exploration (~ marking graph construction)
  - all encountered states are stored in a table
  - all edges are written on a file
- LOTOS abstract data types are implemented by C concrete types
- three successive steps:
  1. construction of a C program (*simulator*)
  2. compilation of this program
  3. execution of this program



A new approach for compiling and verifying LOTOS

Initial goal: verification by model checking of LOTOS specifications.

Derived goal: efficient translation of LOTOS programs into graphs.

The proposed compilation method:

- accepts a large subset of LOTOS
- uses Petri Nets (extended with data) as an intermediate form
- could be easily adapted for:
  - interactive simulation
  - test generation
  - sequential code generation

A tool for LOTOS: CÆSAR

- full implementation of the translation method
  (25 000 lines of C code, SYNTAX compiler-generator)
- graphs up to 800 000 states and 3 500 000 edges
- 40–540 states per second (on a SUN4 with 8 Mbytes)
- connection with 7 verification tools: ALDÉBARAN, PIPN, AUTO