CÆSAR Reference Manual

Hubert GARAVEL[*]

Laboratoire de Génie Informatique
Institut I.M.A.G.
GRENOBLE FRANCE

version 3.7

April 2016: This document is now obsolete. Its contents have been moved (and updated) to three manual pages of CADP, the CÆSAR, CÆSAR.ADT, and LOTOS manual pages, which are available from http://cadp.inria.fr/man and can also be found in each CADP distribution.

Abstract

This report describes a translator from LOTOS *to both interpreted Petri nets and finite state automata. This tool provides a way to verify* LOTOS *programs by model-checking. It applies to a large subset of* LOTOS *(including value expressions) and can be used in conjunction with other existing tools based on graph equivalences and/or temporal logics.*

[*]This work was also supported by I.N.R.I.A.. Author's current address is:

VERILOG Rhône-Alpes
Forum du Pré Milliet
MONTBONNOT
38330 SAINT-ISMIER
FRANCE
e-mail: `hubert@imag.imag.fr` | `hubert@imag.uucp`
tel: +(33) 76 52 18 36 fax: +(33) 76 52 12 39

Contents

1 What is CÆSAR? 4

2 How does CÆSAR work? 4
2.1 Lexical and syntactic analysis phase . . . 5
2.2 Semantic analysis phase . . . 5
2.3 Restriction phase . . . 5
2.4 Expansion phase . . . 7
2.5 Generation phase . . . 8
2.6 Optimization phase . . . 8
2.7 Simulation phase . . . 9

3 How to run CÆSAR? 9
3.1 Installation . . . 9
3.2 Command-line synopsis . . . 10
3.3 Inputs/Outputs . . . 10
3.4 Interface options . . . 10
3.4.1 Interfacing ALDEBARAN . . . 11
3.4.2 Interfacing XESAR . . . 12
3.5 Default options . . . 12
3.6 General options . . . 12

4 How to supply libraries? 13

5 How to supply concrete types? 13
5.1 Special comments . . . 13
5.2 Concrete types . . . 15
5.3 Concrete sort implementation . . . 15
5.4 Concrete sort comparison . . . 16
5.5 Concrete sort enumeration . . . 16
5.6 Concrete sort printing . . . 17
5.7 Concrete operation implementation . . . 18

6 How to understand CÆSAR's diagnostics? 19
6.1 Standard output and error file . . . 19
6.2 Messages, warnings and errors . . . 19
6.3 Errors during the simulation phase . . . 19

A Known bugs 21
A.1 Lexical and syntactic analysis phase . . . 21
A.2 Libraries . . . 21
A.3 Special comments . . . 21
A.4 Semantic analysis phase . . . 22
A.5 Expansion phase . . . 23
A.6 Generation phase . . . 23
A.7 Simulation phase . . . 23
A.8 Error messages . . . 24

B A complete example 24
B.1 Preparing the ".lotos" and ".lib" files . . . 24

B.2 Preparing the ".h" file 26
B.3 Generating the ".h" file with CÆSAR.ADT 27
B.4 Generating the graph 29
B.5 Reducing the graph 29
B.6 Comparing two graphs 30
B.7 Generating a reduced graph 31
B.8 Designing verification tools 31

1 What is CÆSAR?

CÆSAR belongs to the CESAR software family, which contains several verification tools (QUASAR [FSS83] [FRV85], XESAR [RRSV87] [Rod88], CÆSAR [Gar89a] [GS90]) intended for formal verification of distributed systems. All these tools share common principles:

- the system to verify is described in some high-level parallel language (FDT), such as a CSP-like for QUASAR, a variant of ESTELLE for XESAR, LOTOS for CÆSAR.
- this description is automatically compiled into a *state graph* (i.e., a finite state automaton, also called a labelled transition system).
- the translator proceeds in several steps. First the source description is translated into an intermediate model which constitutes a compact, structured and user-readable representation of both parallel control flow and data flow. XESAR uses *communicating automata* (also *called abstract machines*) whereas CÆSAR is based on *networks*, an extension of interpreted Petri nets.
- then, this intermediate form is translated into a graph, by applying *reachability analysis*, i.e., *exhaustive simulation* of all possible evolutions of the system. For an ordinary Petri Net, this would be nothing but marking graph computation. The translation is efficient because of the static control skeleton provided by the intermediate model.
- the properties to verify are evaluated on the graph. They can be expressed as formulas of some temporal logic (CTL for QUASAR, LTAC for XESAR). They can also consist in the comparison of the graph with another graph, for a given equivalence relation (such as strong equivalence, observational (i.e., weak) equivalence, testing equivalence, language (i.e., trace) equivalence, ...)

2 How does CÆSAR work?

CÆSAR takes as input a LOTOS program [ISO89] and produces both a corresponding network and a strongly-equivalent graph. CÆSAR itself does not embody verification tools, such as graph reductors or temporal logic evaluators. However CÆSAR smoothly interfaces with a variety of verification systems which operate on graphs.

The translation process is divided into 7 successive phases:

- lexical and syntactic analysis
- semantic analysis
- restriction
- expansion
- generation
- optimization
- simulation

2.1 Lexical and syntactic analysis phase

The LOTOS program is scanned and parsed, according to the lexical and syntactical definition of LOTOS. During this phase, an abstract syntax tree is built.

2.2 Semantic analysis phase

The abstract tree is explored to enforce static semantic rules of LOTOS definition. This phase is divided into 8 successive sub-phases:

- gates binding
- processes binding
- types binding
- signature analysis
- sorts binding
- variables binding
- operations binding
- functionality analysis

The analysis phase is defined to be the succession of both syntactic and semantic phases. Any program which successfully goes through analysis phase should be correct with respect to [ISO87]. A serious attempt was made to fully implement the DIS standard. Deviations are reported in appendix A.

The analysis phase is quite fast: the average speed is close to 60 lines per second. It can handle large programs, at least 6 300 lines and 220 000 bytes.

This part of CÆSAR constitutes the front-end part of other tools, especially CÆSAR.ADT [Bar88] [Gar89b].

2.3 Restriction phase

To be translated into finite state graphs, LOTOS programs shall define regular behaviors. But the translation technique also requires additional constraints for efficiency reasons (in order to have a Petri Net with at most a single token per place). For any LOTOS program, the following constraints (*static control property*) must be enforced:

- there shall be no recursive process-instantiation[1] on the left (respectively right) of a parallel operator.
- there shall be no recursive process-instantiation through a "**par**" operator.
- there shall be no recursive process-instantiation on the left of a *disable* operator.
- there shall be no recursive process-instantiation on the left of an *enable* operator.

[1]Note that any process-instantiation of a recursive process is not necessarily a recursive process-instantiation

Example 1

For instance, the behavior shown below violates these constraints:

```
      BufferChain [Get, Put] (0, 3)
   where
      process BufferChain [Input, Output] (Low, High:Nat) : noexit :=
         [Low = High] ->
            Buffer [Input, Output]
         []
         [Low < High] ->
            hide Middle in
               (
               Buffer [Input, Middle]
               |[Middle]|
               BufferChain [Middle, Out] (Low + 1, High)
               )
      endproc
```

because it involves recursion on the right part of a parallel operator. The user has to develop recursion by hand, to obtain a fixed number of concurrent processes. The previous behavior should be replaced by:

```
hide Middle0, Middle1, Middle2 in
   (
   Buffer [Input, Middle0]
   |[Middle0]|
   Buffer [Middle0, Middle1]
   |[Middle1]|
   Buffer [Middle1, Middle2]
   |[Middle2]|
   Buffer [Middle2, Output]
   )
```

■

The restriction phase rejects programs which do not satisfy the static control property.

Example 2

```
In specification TOKEN_RING_PROTOCOL [1]
   - process STATION [51] recursively instanciates process ELECTION [68]
     on the left of some ``[>'' operator
     in the strongly-connected component which contains processes
        STATION [51], PRIVILEDGE [58], ELECTION [68]
```

■

The class of programs accepted by CÆSAR is expected to be very close to the whole class of regular behaviors: no operator is discarded; value expressions are handled; in many cases unbounded value domains will be also accepted (provided that no attempt is done to enumerate them exhaustively).

It is not possible to decide whether the behavior defined by a program is regular or not. In fact, programs that do not satisfy the above constraints are often non-regular.

Furthermore the restriction phase statically detects processes which cannot be reached from the entry of the specification, even if some instantiations of this process exist.

Example 3

```
In specification TOKEN_RING_PROTOCOL [1]
   - process LINK [82] is never reached
```

■

2.4 Expansion phase

It was found convenient to divide network generation in two phases, in order to simplify the translation scheme, leading to less complex and more reliable algorithms.

The first phase (*expansion*) expands the LOTOS abstract tree into a so-called SUBLOTOS abstract tree. SUBLOTOS is a process algebra similar to LOTOS, with the following differences:

- a SUBLOTOS program describes a *finite and statically fixed* set of concurrent processes. These processes interact by means of a *statically fixed* set of communication gates.
- the behavior of each process is both determined by a SUBLOTOS algebraic term and by the values of *finite and statically fixed* set of state variables (related to the variables occurring in the source LOTOS program). Each SUBLOTOS variable has a global scope and can be assigned more than one time: SUBLOTOS is not a functional language.
- only LOTOS specifications that satisfy static control constraints can be translated in SUBLOTOS.

From a given LOTOS behavior expression, CÆSAR generates an equivalent SUBLOTOS behavior expression. The translation is defined by induction rules which map LOTOS behavior operators to SUBLOTOS ones.

- the following LOTOS behavior operators: "**par**", "**choice**" on gates, "**exit**", "`>>`" and "`|||`" do not appear in SUBLOTOS. They are replaced by semantically equivalent forms.
- all occurrences of termination gate "δ" are made explicit.
- SUBLOTOS semantics is free from gate relabelling. It is therefore necessary to remove first-order gates which appear in LOTOS programs. This implies that process instantiation should be developed until actual gate parameters are found to be identical to formal gate parameters.

  Example 4
  For instance, the following LOTOS behavior:

```
   Cell [Get, Put] (0)
where
   process Cell [Input, Output] (N:Nat) : noexit :=
      Input ?M:Nat [M gt N];
         Output !M;
            Cell [Output, Input] (M)
   endproc
```

  is translated into the following SUBLOTOS behavior:

```
   Cell [Get, Put] (0)
where
   process Cell [Get, Put] (N:Nat) : noexit :=
      Get ?M:Nat [M gt N];
         Put !M;
            (
            let N':Nat = M in
               Put ?M':Nat [M' gt N'];
                  Get !M';
                     Cell [Get, Put] (M')
            )
   endproc
```

■

It is proven that recursion development always remains finite. Expansion creates new gate, variable, and process identifiers to ensure that each object is given a unique name.

2.5 Generation phase

The second phase (*generation*) translates a SubLotos abstract tree into a semantically equivalent network.

Networks are a generalization of interpreted Petri nets. A network is composed of *places* and *transitions*. This control part reflects the structure of behavior-expressions occurring in the corresponding Lotos program.

Transitions are labelled by *gates-identifiers* and *experiment-offers* (related to those which occur in the source Lotos program). Some transitions are labelled by a special gate, "ϵ", which means they are spontaneous and non-visible[2].

Transitions are also labelled by *actions*, *conditions*, and *iterations*. Actions modify the content of state variables. Conditions prevent transitions from being fired, depending on the value of state variables. Iterations make given state variables enumerate all possible values in their domains (which must be finite).

The generation makes a recursive descent through the syntax tree and synthetizes the network in a bottom-up way. The algorithm has some analogies with those for producing a finite state automaton given a regular expression. No attempt is made to interpret value-expressions: conditions and actions are generated as symbolic expressions.

During the generation phase, static analysis of synchronized *rendez-vous* is performed and possible deadlocks are reported to the user. Cæsar detects the situations where an action-prefix can not be derived because there is no compatible action-prefix (i.e., same gate and offer) to synchronize with.

Example 5

```
In specification SLIDING_WINDOW_PROTOCOL [1]
   - a deadlock exists for rendez-vous:
        RACK [2]  !NAT [NATURALNUMBER:6]
     synchronized by parallel operator ``|[exit, SDT, RDT, RACK, SACK]|''
     this gate/offer combination is used in the left operand:
        TRANSMITTER [exit, PUT, SDT, SACK] (...)
        |[exit]|
        RECEIVER [exit, GET, RDT, RACK] (...)
     but not in the right operand:
        MEDIUM [exit, SDT, RDT]
        |[exit]|
        MEDIUM [exit, RACK, SACK]
```

■

2.6 Optimization phase

The network produced by the generation phase is simplified, according to several reduction rules. The transformations affect both control and data flow. Optimization preserves strong equivalence.

[2]ϵ-transitions are different from CCS τ-transitions; they behave as ϵ-edges in non-deterministic automata theory, but they are also *atomic*

Another optimization, called *safety reduction*, replaces all τ-transitions by ϵ-transitions. It is not performed by default unless explicitly requested by the user. It does not preserve strong equivalence nor observational equivalence. It is believed, however, that it preserves safety equivalence [Rod88] (a relation which is weaker than observational equivalence, but stronger than trace equivalence). This conjecture has not be proven formally, but seems to hold on a large number of examples. It can be useful for large programs when trace equivalence is sufficient for the properties to prove. This optimization usually produces graphs that are much smaller than those produced for strong equivalence (since the generated graphs are partially reduced according to trace equivalence).

2.7 Simulation phase

This phase exhaustively explores all possible behaviors defined by the network to produce a state graph. All states that can be reached from the initial state are stored in a table. Whenever a state is generated, it is compared with existing states in the table. If present, a cyclic edge appears in the graph. Otherwise the state is entered in the table. The state table is extensible since its size can not be predicted until simulation terminates. To allow fast search, hashing techniques are used.

Each *state* of the graph can be seen as a pair constituted by a marking (the set of places which are marked) and a context (the current values of state variables). Each *edge* is labelled by a gate and an offer, which is a list of value expressions without variable. The number of states, as well as the number of edges, is only limited by the amount of memory available.

A non-regular LOTOS program requires the production of an infinite graph. Since the available memory space and the number of states allowed are bounded, the simulation phase always terminates. When simulation stops because of memory exhaustion, CÆSAR can not decide whether the program is non-regular or regular with too much states; the user has to find it by himself.

To deal with values, CÆSAR builds a C program, called *simulator*. This program contains rules to determine which network transitions can be fired from the current state, an algorithm to explore exhaustively all states of the graph, and routines for state table management. The simulator program also imports concrete types and and functions definitions supplied by the user (see § 5). Then CÆSAR calls the C compiler to translate the simulator program and the resulting executable code is run, generating the graph and writing it to a file.

For efficiency reasons, CÆSAR does not handle the abstract data types of the source LOTOS programs by term-rewriting techniques. The user has to translate LOTOS sorts and operations into equivalent C types and functions that must be supplied to the C compiler when the simulator program is compiled.

3 How to run CÆSAR?

3.1 Installation

CÆSAR is not a public-domain software, but is is free for non-commercial use, under license agreement. To order it, please send a request to `hubert@imag.imag.fr`.

CÆSAR is written in C (85 files, 25 000 lines of code). Lexical-syntactic analysis and abstract tree construction are performed by a powerful compiler-generator, SYNTAX[3].

CÆSAR should be portable to any UNIX[4] operating system; however the current version only runs

[3]SYNTAX is a trademark of I.N.R.I.A.

[4]UNIX is a trademark of AT&T Bell Laboratories

on the SUN 3 workstations family.

CÆSAR binary code size is about 250 k-bytes. The executable file "`caesar`" is ready-to-use. There is no environment to build, nor special file to install, nor shell variable to set.

3.2 Command-line synopsis

CÆSAR can be called from any UNIX shell. The command line obeys the usual UNIX conventions:

`caesar` [-*option*$_1$] `...` [-*option*$_n$] *name*[`.lotos`]

CÆSAR takes as input the LOTOS specification contained in file "*name*`.lotos`". As a shorthand the "`.lotos`" suffix can be omitted in the command line.

An option is said to be *set* either when it appears in the command line or when it is a default option which is not cancelled by another option.

3.3 Inputs/Outputs

An execution of CÆSAR takes 4 different kinds of inputs:

- the command-line options
- the LOTOS program ("*name*`.lotos`" file)
- the LOTOS libraries ("*type*`.lib`" files)
- the C implementation of abstract data types ("*name*`.h`" file)

and generates 4 kinds of outputs:

- the graph ("*name*`.aut`", "*name*`.gph`", ... files)
- the network ("*name*`.net`" file)
- the abstract/concrete mappings ("*name*`.map`" file)
- the diagnostics (standard output and "*name*`.err`" file)

The meaning of these input and output data is detailed below.

3.4 Interface options

CÆSAR provides the suitable interfaces for 7 existing verification systems: ALDEBARAN, AUTO, MEC, PIPN, SCAN, SQUIGGLES and XESAR. It can generate graphs in the right format expected by each tool. To interface one of these tools, an appropriate option must be set. Several options can be set simultaneously. Whatever tool(s) the user chooses, the semantics of the graph remains the same, but the number, the suffix and the format of the generated files change.

tool	*institution*	*option*	*generated file(s)*
ALDEBARAN	LGI-IMAG (Grenoble)	`aldebaran`	*name*`.aut`
AUTO	INRIA (Sophia)	`auto`	*name*`.m0`
MEC	University of Bordeaux I	`mec`	*name*`.mec`
PIPN	LAAS (Toulouse)	`pipn`	*name*`.auto.pro`
SCAN	BULL	`scan`	*name*`.scan`
SQUIGGLES	CNUCE (Pisa)	`squiggles`	*name*`.graph`
XESAR	LGI-IMAG (Grenoble)	`xesar`	*name*`.gra`, *name*`.dp3`, *name*`.ge3`, *name*`.tai`

In all graph formats, visible gates are only those which appear as parameters of the specification. According to LOTOS semantics, all other gates are relabelled as "**i**". The termination gate "δ" is displayed as "**exit**".

When the simulation phase stops (because there are too many markings, or contexts, or states, or because no more memory is available) the uncompleted graph files are not removed, so that the user can consult the beginning of the graph. In that case, all data which can not be known before the end of simulation phase (e.g., the number of states and the number of edges) are replaced by character "?".

3.4.1 Interfacing ALDEBARAN

The ALDÉBARAN [Fer88] [Fer90] graph format seems to be the simplest one to understand. Each state is numbered with a natural number. The first line of the "`.aut`" file, called *descriptor*, has the following structure:

`des (`*first_state*`, `*number_of_transitions*`, `*number_of_states*`)`

The first state is always equal to zero. Each of the remaining lines of the file represents an edge; these lines have the following structure:

`(`*from_state*`, "`*gate_name* `!`$value_1$ `...` `!`$value_n$`", `*to_state*`)`

Each value is the ASCII representation of an algebraic term, according to printing conventions that are to be defined by the user (see § 5.6). There may be no value occurring after the gate name.

Example 6
Examples of graphs in ALDÉBARAN format are, for instance:

```
des (0, 599, 555)
(0, "PUT  !0", 1)
(0, "SACK  !0 !false", 2)
(0, "SACK  !1 !false", 3)
(0, i, 4)
(0, i, 5)
(1, "SDT  !0", 6)
(2, "PUT  !1", 7)
...
```

or:

```
des (0, 13274, 10165)
...
(9265, "Y  !‘W1*X3+W2*X4+W3*X5’", 10160)
(9266, i, 10161)
(9267, "Y  !‘W1*X3+W2*X4+W3*X5’", 10161)
(9268, i, 10162)
(9269, "Y  !‘W1*X3+W2*X4+W3*X5’", 10162)
(9270, i, 10163)
(9271, i, 10164)
(9272, "Y  !‘W1*X3+W2*X4+W3*X5’", 10164)
```

■

3.4.2 Interfacing XESAR

The connection to XESAR [Rod88] temporal formula evaluator is only possible for graphs whose edges are labelled only by gates, and not by values.

This can be done by using the shell-script "`caesar.xesar`" which appears in the distribution package.

Temporal logic formulas are presented in [RRSV87]. Basic predicates have the form "`enable (`*G*`)`" and "`after (`*G*`)`" where *G* is either one of the gates following keyword "**specification**" or "**exit**" which denotes the termination gate "δ" of the specification.

Example 7
Here are some temporal formula of current use:

```
not sink
pot enable (exit)
after (SEND) => inev [not enable (SEND)] after (RECEIVE)
...
```

■

3.5 Default options

The "`network`" option prints the network corresponding to program "*name*`.lotos`" on the "*name*`.net`" file. Undocumented.

The "`graph`" option prints the graph corresponding to program "*name*`.lotos`" on the "*name*`.gph`" file. Yet another graph format, mainly intended for debugging purpose. It also contains interesting information on edges, especially the original names and definition lines that gates have before they are renamed as "`i`" (when they occur in the scope of a *hiding* operator). Undocumented.

When no tool option is set, the "`network`" and "`graph`" options are set by default. These options can also be used together with other interfaces options.

3.6 General options

CÆSAR is bilingual. Two options are available "`english`" and "`french`". The default option is "`french`".

Various versions of UNIX do not agree on the length of file names. CÆSAR provides two options, "`systemV`" and "`berkeley`", to solve this problem. When given the "`systemV`" option, CÆSAR truncates file names to ensure that they fill into 14 characters. The user is warned when truncation is necessary. On the opposite hand, the "`berkeley`" option never truncates. The default option is "`berkeley`". The files produced by the "`xesar`" option are always generated as if the "`systemV`" option were set.

The "`simulator`" option prevents CÆSAR from compiling and running the simulator program. When this option is given, CÆSAR stops just after the "*name*`.c`" file is produced. This C file may be compiled and run on another machine, for which a version of CÆSAR does not exist yet. This can be useful if the simulator program is too large for the current compiler, or if it compiles and executes too slowly on the current machine. This would allow for instance to use a Cray to perform formal verification!

The "`safety`" option implements the safety reduction described above (see § 2.6). As a consequence of the network transformation, the generated graph is also modified.

4 How to supply libraries?

CÆSAR handles libraries by textual inclusion. When a library declaration of the following form is encountered:

library *name* **endlib**

the string *name* is converted into upper-case letters (since LOTOS is case unsensitive) and the file "*name*`.lib`" is searched in the current directory. The library declaration is replaced by the content of this file.

Example 8
For instance, the following LOTOS declarations:

```
library Boolean endlib
library NaturalNumber endlib
library Bit endlib
```

are replaced by the contents of files "`BOOLEAN.h`", "`NATURALNUMBER.h`" and "`BIT.h`". ■

CÆSAR does not handle multiple search rules. Use symbolic links to access shared libraries.

Cyclic inclusion, direct and transitive, is detected and rejected.

5 How to supply concrete types?

To accept a significant class of LOTOS programs, a verification tool should handle value-expressions, variables, sorts, operations, types, ... However, current techniques for interpretation of algebraic data types are not very efficient. These methods based on rewriting or symbolic evaluation would be too slow.

That's why the user has to provide concrete implementations of the abstract data types, which are used during CÆSAR's simulation phase. Each LOTOS sort is mapped to a C type. Each LOTOS operation is mapped to a C function or macro-definition. This translation from LOTOS to C can be done by hand or automatically, using the CÆSAR.ADT system [Bar88] [Gar89b].

5.1 Special comments

How to make the correspondence between abstract LOTOS sorts (respectively operations) identifiers and concrete C types (respectively functions)? The problem is not simple because lexical conventions of LOTOS and C differ and because LOTOS allows overloading whereas C does not.

For each LOTOS sort and operation name the user has to supply a corresponding C name. This solution allows interfacing with user-defined type and functions libraries. To do that *special comments* are used. These special comments are a subset of LOTOS comments: "`(*! ... *)`". Unlike ordinary comments, their content is meaningful and parsed. They should only appear immediately after the declaration of sort and operation identifiers.

Example 9

```
type BasicNaturalNumber is
   sorts
      Nat  (*! implementedby NAT comparedby CMP_NAT
               enumeratedby ENUM_NAT printedby PRINT_NAT *)
   opns
      0    (*! implementedby ZERO constructor *) : -> Nat
      Succ (*! implementedby SUCC constructor *) : Nat -> Nat
      _+_  (*! implementedby PLUS *),
      _*_  (*! implementedby MULT *),
      _**_ (*! implementedby POWER *) : Nat, Nat  -> Nat
   eqns
      ...
endtype
```

■

Lexical definition of special comments uses the following definitions:

$sep = (space \mid carriage\_return \mid line\_feed \mid form\_feed \mid horizontal\_tabulation)^{+}$
$C\_name = (letter \mid underscore)\ (letter \mid underscore \mid digit)^{*}$

The non-terminal *sep* denotes a non-empty sequence of separator characters. The non-terminal *C_name* denotes a C identifier. Upper-case and lower-case are different. CÆSAR does not accept names which are identical to some C keyword or which are prefixed by the reserved string "`CAESAR_`".

Special comments associated with sorts have the following syntax:

(*! *sep*
  [**implementedby** *sep* C_name_1 *sep*]
  [**comparedby** *sep* C_name_2 *sep*]
  [**enumeratedby** *sep* C_name_3 *sep*]
  [**printedby** *sep* C_name_4 *sep*]
*)

Letters in words "**implementedby**", "**comparedby**", "**enumeratedby**" and "**printedby**" can be either uppercase or lowercase. Four *attributes* are attached to each sort S:

- C_name_1 denotes a C type which implements S.
- C_name_2 denotes a C function (or macro) which implements equality operation between values of S, since guards "V_1=V_2" must be evaluated even if S has no "`eq`" operation associated with.
- C_name_3 denotes a C macro which iterates over the domain of S, in order to implement LOTOS constructions: "**any** S", "?X:S", "**choice** X:S".
- C_name_4 denotes a C function which prints values of S on a file with a suitable ASCII format.

Special comments associated with operations have the following syntax:

```
(*! sep
   [ implementedby sep C_name sep ]
   [ constructor sep ]
*)
```

Letters in words "**implementedby**" and "**constructor**" can be either lowercase or uppercase. Two *attributes* are attached to each operation F:

- *C_name* denotes a C function (or macro) which implements F.
- "**constructor**" indicates that F is a *constructor* operation. This information is not used by CÆSAR; it is only relevant for CÆSAR.ADT.

Whenever an attribute is not supplied, CÆSAR automatically generates a C name, prefixed by string "`CAESAR_`", and different from all previously existing C names.

The "`map`" option, when set, generates a "*name*`.map`" file containing the list of types, sorts and operations defined in program "*name*`.lotos`" and, for each LOTOS name, the corresponding C identifier. Undocumented.

5.2 Concrete types

The concrete implementation of the abstract data types defined in "*name*`.lotos`" file must be put in a "*name*`.h`" file (located in the current directory). CÆSAR generates a simulator program (the "*name*`.c`" file) whose first lines are:

```
#include <stdio.h>
#include "name.h"
```

When the simulator program is compiled by the UNIX C compiler, the "*name*`.h`" file supplied by the user is consequently included.

Following sections define the contents of the "`.h`" file. This file can be:

- either written by hand, according to the rules given in the next sections. Many other examples of abstract/concrete data types can be found in the "`lib`" directory of the CÆSAR/ALDÉBARAN Distribution Package. A comprehensive list of common errors is given in section 6.3
- or automatically generated by CÆSAR.ADT. A practical example combining CÆSAR.ADT with CÆSAR is shown in appendix B.3

5.3 Concrete sort implementation

Each LOTOS sort is implemented by a C type. The most simple way to do this is to provide a "**typedef**" definition in the "*name*`.h`" file. Macro-definitions could also be used as an alternative approach.

Example 10 The "`Nat`" sort of type "`BasicNaturalNumber`" (see § 5.1) can be implemented as follows:

```
typedef unsigned char NAT;
```

■

In the current version of CÆSAR, any C type can be used to implement abstract sorts, provided that the assignment operator ("`=`" in C) applies to values of this type. In other words, it is necessary that concrete values can be copied by using assignments.

The following types can be used : "**char**", "**int**", "**enum**", "**struct**" and "**union**".

Pointer-types (e.g., arrays, linked lists) are also allowed, provided that no function modifies the data structure pointed by its arguments: each function must first make a copy of this data structure, and modify only this copy. More generally, the C functions given to CÆSAR should never perform side effects. Under this condition, structural sharing (i.e., several pointers referring to the same object) is correct.

5.4 Concrete sort comparison

The comparison function associated to a LOTOS sort S has two arguments, the type of which is the concrete type that implements S, and an integer result (0 means *False*, and any other value means *True*, according to C language conventions). The definition of this function is often a mere equality test between both arguments.

Example 11
The comparison function for sort "`Nat`" can be defined as follows:

```
int CMP_NAT (X1, X2)
   NAT X1, X2;
   {
   return X1 == X2;
   }
```

or, more simply:

```
#define CMP_NAT(X1,X2) ((X1) == (X2))
```

■

If S is implemented by a pointer-type, the comparison function should first check whether each of its arguments is equal to "`NULL`" or not. This is true even if no value in the domain of sort S is concretely represented by "`NULL`" (because CÆSAR initializes all simulator variables to a bit string of zeros).

5.5 Concrete sort enumeration

The iteration macro associated to a LOTOS sort S has a single argument which is a C variable X, the type of which is the concrete type that implements S. The purpose of the iteration macro is to define a loop control structure which allows X to enumerate all possible values in the domain of S. If this domain is infinite, the iteration macro defines the finite subset to be enumerated.

Example 12
To enumerate the range 0..10, the user should supply the following macro:

```
#define ENUM_NAT(X) for ((X) = 0; (X) <= 10; ++(X))
```

■

More often than not, an iteration macro is the header of a "**for**" loop. But other schemes might be used.

Example 13

- enumerating range 0..10 can be done with a “**while**” loop:

```
#define ENUM_INT(X) (X) = 1; while ((X)++ <= 0)
```

- the “**for**” loop header can be followed by a test; for instance the following iteration macro enumerates all integers in range 0..99 that cannot be divided (exactly) by 3:

```
#define ENUM_NAT(X) for ((X) = 0; (X) < 100; ++(X)) if ((X) % 3 != 0)
```

- it is also possible to limit the enumeration to a single value, randomly chosen in the domain of the sort:

```
#define ENUM_NAT(X) (X)=(NAT) rand();
```

  A “`#define`” statement should never be followed by a “`;`” symbol; in the previous example, however this symbol is mandatory to express sequential composition of instructions.

■

5.6 Concrete sort printing

The printing function associated to a LOTOS sort S has two arguments and a result of “`void`” type. The first one is a file descriptor (which has the type “`(FILE *)`” defined in the standard UNIX input/output library “`stdio.h`”). The second argument is a concrete value, the type of which is the concrete type that implements S. The printing function is used to write the values attached to the edges of the graphs.

Example 14
The printing function for sort “`Nat`” can be defined as follows:

```
void PRINT_NAT (F, X)
   FILE *F;
   NAT X;
   {
   fprintf (F, "%u", (NAT) X);
   }
```

■

More often than not, the printing function is simply a macro-definition:

Example 15

```
#define PRINT_NAT(F,X) fprintf (F, "%u", (NAT) (X))
```

■

If S is implemented by a pointer-type, the printing function should first check whether its arguments is equal to “`NULL`” or not. This is true even if no value in the domain of sort S is concretely represented by “`NULL`” (because CÆSAR initializes all simulator variables to a bit string of zeros).

For existing output graph formats produced by CÆSAR, any value should be printed as an ASCII string without control characters. Depending on the graph format, some meaningful characters should be avoided or escaped. For instance, in the ALDÉBARAN format, double quote character “`"`” should be escaped by a backslash character: “`\"`”.

5.7 Concrete operation implementation

Each LOTOS operation is implemented by a C function or a C macro-definition. The parameters and result of the concrete operation must be compatible with those of the abstract operation.

Example 16
The operations of type "`BasicNaturalNumber`" can be implemented either as plain C functions (the definitions of which reflect the algebraic equations):

```
NAT ZERO ()
   {
   return 0;
   }

NAT SUCC (X)
   NAT X;
   {
   return X;
   }

NAT PLUS (X1, X2)
   NAT X1, X2;
   {
   if (X2 == ZERO) return X1;
   else return PLUS (SUCC (X1), X2 - 1);
   }

NAT MULT (X1, X2)
   NAT X1, X2;
   {
   if (X2 == ZERO) return ZERO;
   else return PLUS (X1, MULT (X1, X2 - 1));
   }

NAT POWER (X1, X2)
   NAT X1, X2;
   {
   if (X2 == ZERO) return SUCC (ZERO);
   else return MULT (X1, POWER (X1, X2 - 1));
   }
```

or as C macro-definitions:

```
#define ZERO() 0

#define SUCC(X) ((X) + 1)

#define PLUS(X1,X2) ((X1)+(X2))

#define MULT(X1,X2) ((X1)*(X2))

#include <math.h>
#define POWER(X1,X2) ((NAT) pow ((X1), (X2)))
```

■

Caution! When a LOTOS operation with no operand (i.e., an operation of arity zero) is implemented by a C macro-definition the macro name must always be followed by parentheses (as shown above for "`ZERO`").

6 How to understand CÆSAR's diagnostics?

6.1 Standard output and error file

According to UNIX conventions, CÆSAR exits with status 0 (success) or 1 (failure). The textual diagnostics are divided up among two files:

- the UNIX "`stdout`" stream, which is generally assigned to the user's terminal. CÆSAR unbuffers this stream. For simplicity, the "`stderr`" stream is never used.
- the *error file*, which is "*name*`.err`" for program "*name*`.lotos`". This file is created at the beginning of execution and removed, if empty, when CÆSAR terminates.

When execution stops, the content of the error file is displayed on the standard output, using the "`/usr/ucb/more`" UNIX command, unless the "`error`" option is set.

6.2 Messages, warnings and errors

There are three classes of diagnostics:

- the *messages*, displayed on the standard output, which report to the user the advancement state of the translation (i.e., the name of each phase, as they are executed).

  The "`silent`" option, when set, suppresses these messages, making it possible to use CÆSAR in a quiet way, like other UNIX compilers.
- the *short errors and warnings*, displayed on the standard output, which indicate some failure. They are composed of 2 or 3 lines. The first line gives the internal error code (which follows the sharp character "`#`"), the error class (error, warning, unexpected bug, ...) and the phase in which the problem occurred. The second line gives a short indication related to the cause of the error. If recovery is impossible, CÆSAR stops.

  The "`warning`" option, when set, suppresses only warnings, but not other classes of errors.
- the *long errors and warnings*, printed on the error file, provide the user with detailed explanations. It is worth noting that SYNTAX issues pertinent diagnostics and does valuable lexical and syntactic error repair.

  Diagnostics for static semantics errors are well-studied also. They give, for each identifier I, the line N where it is defined; this is the meaning of notation "I `[`N`]`". If the identifier I is not defined in the main program but at line N of library L, the notation "I `[`L`:`N`]`" is used.

All identifiers occurring in error diagnostics are displayed in upper-case letters (since LOTOS is not case-sensitive).

6.3 Errors during the simulation phase

If the C compiler reports errors while compiling the simulator, it is very likely that the problem comes from the concrete types supplied by the user, and not from the C code generated by CÆSAR. Some common errors are listed below:

1. a C type or a C function defined in the "`.h`" file does not have the same name as that given in the corresponding special comment
2. there is a missing semi-colon "`;`" at the end of a "**typedef**" declaration
3. there is an extra semi-colon "`;`" at the end of a "`#define`" macro-definition
4. there is an extra space immediately after the macro name in a "`#define`" construction. For instance:
   ```
   #define SUCC (X) ((X)+1)
   ```
   is not equivalent to:
   ```
   #define SUCC(X) ((X)+1)
   ```
   (the latter solution being the right one)
5. parentheses "`()`" are missing immediately after the macro name in a "`#define`" construction. For example, one must write:
   ```
   #define ZERO() 0
   ```
   instead of:
   ```
   #define ZERO 0
   ```
   In such case, the C compiler often reports an "illegal function declaration"
6. an enumeration macro is not properly defined. In such case, the C compiler reports a syntax error (usually "too many {")

Similarly, if the simulator "core dumps", it is (often) wiser to examine the "`.h`" file, rather than suspecting the "`.c`" file automatically generated by CÆSAR. Here are some errors which often occur:

1. a C function has no "**return**" instruction
2. a C function implementing "**comparedby**" has pointer-type arguments and does not check whether its arguments have the "`NULL`" value
3. a C function implementing "**printedby**" has a pointer-type argument and does not check whether its argument has the "`NULL`" value
4. a C function implementing some LOTOS operation makes side effects on its (pointer-type) arguments

Acknowledgements

The first prototype of CÆSAR was developed in the SEDOS project of the European ESPRIT program. The author would like to thank his colleagues for their suggestions.

Next versions of the tool were realized in collaboration between the C.E.D.I.A. project at I.N.R.I.A. and the SPECTRE project at L.G.I.-I.M.A.G. Members of both teams are acknowledged, particularly Jacky Barré, Ahmed Serhrouchni who developed the early version of the LOTOS compiler, Pascal Bouchon and Jean-Michel Houdouin who realized the next version of CÆSAR's front-end.

The author is especially grateful to his research director, Joseph Sifakis, to his thesis adviser, Jacques Voiron.

Many thanks to Susanne Graf, Carlos Rodriguez and Véronique Coelho for their valuable suggestions and comments on this manual.

A Known bugs

This section attempts to summarize all known problems the user can be confronted with. Although Cæsar was carefully tested, using an automated Lotos verification suite of more than 600 examples, bugs might still remain. Please, report any bug to `hubert@imag.imag.fr`, sending Lotos programs that make Cæsar fail.

This section also reports errors and ambiguities discovered in the DIS (Draft International Standard) definition of Lotos [ISO87] and states corrections and interpretations chosen by Cæsar.

A.1 Lexical and syntactic analysis phase

- infix operator declaration syntax "_F_" is quite difficult to parse. At the expense of internal complexity, Cæsar does it properly but can not check that operation F is not a reserved keyword. Though it may be possible to declare an infix operator whose name is a keyword, any further occurrences of this operator are syntactically rejected.
- during lexical and syntactic analysis, the length of identifiers is unlimited: all characters are significant. However all identifiers are truncated to 64 characters when they are put in the symbol table.

A.2 Libraries

- the full Lotos construction:

  library T_1**, ...** T_n **endlib**

  is not recognized. The user has to break it into n library definitions:

  library T_1 **endlib**
  ...
  library T_n **endlib**

- "**endlib**" is not a reserved keyword.
- each include file "*name*`.lib`" should only contain a definition of type "*name*". In fact it can contain anything (e.g., processes definitions) provided that inclusion leads to correct Lotos programs.
- library inclusion may violate Lotos rules concerning visibility of identifiers declared in libraries. This subtle difference should be of no interest to most users.
- the maximum number of include files is limited to 32.

A.3 Special comments

- the user should carefully observe the syntax of special comments (see section 5.1) since they are scanned at the lexical level and not the syntactic one. Otherwise, SYNTAX enters into a horrifying panic-mode error recovery process

- if separators are omitted after "`(*!`" or before "`*)`", SYNTAX recovers from this error by inserting an horizontal tabulation, which is noted "`\t`". This may seem cryptic to everybody unfamiliar with C.

- all C identifiers occurring in special comments are truncated to 32 characters.

A.4 Semantic analysis phase

- in section 7.3.4.5.e of [ISO87], requirement e1 is strengthened, stating that, when clause "**accept...in**" is missing, the functionality of behavior B_1 occurring in "B_1 `>>` B_2" shall be "**exit**".

- in section 7.3.4.5.v of [ISO87], a (probably missing) requirement v2 is added, stating that in a guard "`[`L`=`R`]`" both value-expression L and R shall have the same sort.

- LOTOS parsing is ambiguous since the grammar itself does not allow to make the difference between variables and operations with no arguments. The disambiguation is left to further static semantics phase.

  According to [ISO87], which is not clear about this point, variable identification might very well be done at the same time as operation overloading resolution!

  CÆSAR properly dissociates these two problems. First, variables are bound to their definitions; when a conflict occurs between a variable and a nullary operation which share the same name, preference is always given to the variable. Variables which can not bind are supposed to be nullary operations. Then, operation overloading is solved.

- requirement a3 in section 7.3.4.3.a of [ISO87] seems to be erroneous, because it does not prevent sorts and operation from overlapping, as in the following example:

  Example 17

```
specification ...
behavior ...
where
   type T1 is
         sorts S
   endtype
   type T2 is
         sorts S
   endtype
endspec
```

■

  Requirement a3 was replaced by the following constraint:

$$combine(complete(\{TE_i | 1 \leq i \leq n\}))$$

  shall be a non-overlapping data presentation.

There have been some changes between the Draft Internation Standard and the International Standard that are not taken into account by CÆSAR. It seems for instance, that the final definition of LOTOS allows formal sorts and formal operations to be actualized, whereas CÆSAR semantic analysis phase does not.

A.5 Expansion phase

CÆSAR deviates from [ISO89] on the meaning of gate relabelling when the relabelling function is not injective.

Example 18
For instance, according to standard LOTOS semantics, the following behavior:

```
   P [c, c]
where
   process P [a, b] : noexit :=
      a; stop || b; stop
   endproc
```

is equivalent to "**stop**" whereas CÆSAR implements *call-by-value* and considers that this behavior is equivalent to "`c;` **stop**". ■

When such a situation is encountered, CÆSAR issues a warning. In the absence of warning, the translation conforms to the standard.

If the LOTOS specification is defined with formal variable parameters, these parameters are not handled symbolically as in the reference definition of LOTOS dynamic semantics. Instead all possible values are considered. In fact CÆSAR replaces the following LOTOS text:

```
specification P [G1, ... Gn] (X1:S1, ... Xn:Sn) : ...
      ...
   behaviour
      B
endspec
```

by:

```
specification P [G1, ... Gn] : ...
      ...
   behaviour
      choice X1:S1, ... Xn:Sn [] B
endspec
```

A.6 Generation phase

Presently a network cannot have more than 65 535 places.

When a deadlock is found, CÆSAR issues a diagnostic which attempts to localize the problem by displaying both behaviors which can not synchronize. The user might be surprised by the fact these behaviors are expressed in SUBLOTOS and not in LOTOS. However they are often useful to remove undesirable deadlocks.

A.7 Simulation phase

CÆSAR does not verify that concrete types supplied in the "*name*`.h`" are compatible with abstract types defined in the "*name*`.lotos`" file. For instance, implementing the boolean constant "`true`" as 0 leads to nasty problems.

The simulator program "*name*`.c`" cannot be read easily (understatement here).

Major performances gains could be obtained by enhancing the optimization phase to suppress more ϵ-transitions.

When generating the simulator program, Cæsar always uses the assignment operator "`=`" to copy concrete values. The user should be allowed to define its own functions for this purpose. This could be done by adding a "**copiedby**" part to special comments attached to Lotos sorts.

Cæsar initializes all simulator variables to a bit string containing nothing but zeros; as a consequence, pointer-type variables are initialized to "`NULL`". The user should be allowed to specify an initialization value.

All graph formats have not been updated to handle values expressions. As a matter of fact, the "`auto`", "`mec`", "`scan`", "`squiggles`" and "`xesar`" do not work if the edges of the graph are labelled by value expressions.

A more compact graph format, with binary encoding, should be available.

A.8 Error messages

Due to SYNTAX features, the lexical and syntactic phase long diagnostics are always given in English, even if the "`french`" option is set.

B A complete example

This example is a description in Lotos of the alternating bit protocol, taken from [QPF88] [QPF89].

B.1 Preparing the ".lotos" and ".lib" files

The "`DATALINK.lotos`" file contains the Lotos specification to deal with. The source text given in [QPF88] is reprinted here verbatim, with some slights changes: "`Boolean`" is imported in "`FrameType`"; two occurrences of "`Boolean`" are replaced by "`Bool`"; special comments are introduced; numeric suffixes are appended to some variable names to avoid ambiguity in further explanations.

```
specification Datalink [Get, Give] : noexit
   library Boolean endlib
behaviour
   hide Timeout, Send, Receive in
      (
         (
         Transmitter [Get, Timeout, Send, Receive] (0)
         |||
         Receiver [Give, Send, Receive] (0)
         )
      |[Timeout, Send, Receive]|
         Line [Timeout, Send, Receive]
      )
where
   type SequenceNumber is Boolean
      sorts SeqNum (*! implementedby SEQNUM comparedby CMP_SEQNUM
                       printedby PRINT_SEQNUM *)
      opns 0 (*! implementedby ZERO *) : -> SeqNum
           Inc (*! implementedby INC *) : SeqNum -> SeqNum
           Equal (*! implementedby EQ_SEQNUM *) : SeqNum, SeqNum -> Bool
      eqns
         forall X, Y:SeqNum
         ofsort SeqNum
            Inc (Inc (X)) = X;
         ofsort Bool
            Equal (0, Inc (X)) = false;
            Equal (Inc (X), 0) = false;
            Equal (Inc (X), Inc (Y)) = Equal (X, Y)
endtype

type Bitstring is Boolean
   sorts BitString (*! implementedby BITSTRING comparedby CMP_BITSTRING
                    enumeratedby ENUM_BITSTRING printedby PRINT_BITSTRING *)
   opns Empty (*! implementedby EMPTY *) : -> BitString
        Equal (*! implementedby EQ_BITSTRING *) : BitString, BitString -> Bool
   eqns
      ofsort Bool
         forall X:BitString
            Equal (X, X) = true
endtype

type FrameType is Boolean
   sorts FrameType (*! implementedby FRAMETYPE comparedby CMP_FRAMETYPE
                    printedby PRINT_FRAMETYPE *)
   opns Info (*! implementedby INFO *),
        Ack (*! implementedby ACK *) : -> FrameType
        Equal (*! implementedby EQ_FRAMETYPE *) : FrameType, FrameType -> Bool
   eqns
      ofsort Bool
         forall X:FrameType
            Equal (X, X) = true;
            Equal (Ack, Info) = false;
            Equal (Info, Ack) = false
endtype

process Transmitter [Get, Timeout, Send, Receive] (Seq1:SeqNum) : noexit :=
   Get ?Data1:BitString;
      Sending [Timeout, Send, Receive] (Seq1, Data1) >>
         Transmitter [Get, Timeout, Send, Receive] (Inc (Seq1))
where
   process Sending [Timeout, Send, Receive] (Seq:SeqNum, Data:BitString) : exit :=
      Send !Info !Seq !Data;
         (
         Receive !Ack !Inc (Seq) !Empty;
            exit
         []
         Timeout;
            Sending [Timeout, Send, Receive] (Seq, Data)
         )
   endproc
endproc

process Receiver [Give, Send, Receive] (Exp:SeqNum) : noexit :=
   Receive !Info ?Rec:SeqNum ?Data3:BitString;
      (
      [Rec = Exp] ->
```

```
            Give !Data3;
               Send !Ack !Inc (Rec) !Empty;
                  Receiver [Give, Send, Receive] (Inc (Exp))
         []
         [Inc (Rec) = Exp] ->
            Send !Ack !Inc (Rec) !Empty;
               Receiver [Give, Send, Receive] (Exp)
         )
   endproc

   process Line [Timeout, Send, Receive] : noexit :=
      Send ?F:FrameType ?Seq2:SeqNum ?Data2:BitString;
         (
         Receive !F !Seq2 !Data2;
            Line [Timeout, Send, Receive]
         []
         i;
            Timeout;
               Line [Timeout, Send, Receive]
         )
   endproc
endspec
```

Since the LOTOS specifications contains a library declaration ("**library** `Boolean` **endlib**"), it is necessary to have a "`BOOLEAN.lib`" file in the current directory. This file should be taken from the standard library, either by performing a copy:

```
cp /usr/local/caesar/lib/BOOLEAN.lib .
```

or, better, by creating a symbolic link:

```
ln -s /usr/local/caesar/lib/BOOLEAN.lib .
```

B.2 Preparing the ".h" file

The user also has to supply a "`DATALINK.h`" file which contains concrete implementation for abstract data types. The correspondence between LOTOS identifiers and C identifiers is defined by the special comments shown above; it is summarized in the "`DATALINK.map`" file that CÆSAR generates when the "`map`" option is set.

It is necessary to provide an iteration macro "`ENUM_BITSTRING`" for enumerating the domain of the "`BitString`" sort. Since this domain is theoretically infinite, only a finite subset can be enumerated. The subset chosen here is $\{1, ..., 100\}$ (choosing a subset limited to a single value, for instance $\{1\}$, would lead to the ordinary alternating bit protocol).

```
typedef unsigned char BOOL;
#define TRUE() 1
#define EQ_BOOL(B1,B2) ((B1) == (B2))

typedef unsigned char SEQNUM;
#define ZERO() 0
#define INC(N) (!(N))
#define CMP_SEQNUM(N1,N2) ((N1) == (N2))
#define EQ_SEQNUM(N1, N2) ((N1) == (N2))
```

```
#define PRINT_SEQNUM(F,N) fprintf (F, "%u", N)

typedef unsigned char BITSTRING;
#define EMPTY() 0
#define CMP_BITSTRING(N1,N2) ((N1) == (N2))
#define EQ_BITSTRING(N1,N2)  ((N1) == (N2))
#define PRINT_BITSTRING(F,N) fprintf (F, "%u", N)
#define ENUM_BITSTRING(N) for(N = 0; N < 100; ++ N)

typedef enum {ack, info} FRAMETYPE;
#define ACK() ack
#define INFO() info
#define CMP_FRAMETYPE(T1,T2) ((T1) == (T2))
#define EQ_FRAMETYPE(T1,T2)  ((T1) == (T2))
#define PRINT_FRAMETYPE(F,T) fprintf (F, ((T)==ack) ? "ack" : "info")
```

B.3 Generating the ".h" file with CÆSAR.ADT

It is also possible to produce automatically the "`DATALINK.h`" file by using CÆSAR.ADT. To do so, some changes have to be made in the type definitions of the LOTOS specification:

- one must indicate constructor operations to CÆSAR.ADT, by using special comments of the form "`(*! ... constructor *)`"
- in type "`SequenceNumber`", the equation "`Inc (Inc (X)) = X`" does not satisfy the "constructor discipline" constraints enforced by CÆSAR.ADT. Given the fact that sort "`SeqNum`" describes integers modulo 2, a simple way to overcome this problem is to introduce an operator denoting the value "`1`"; the equations are modified consequently
- in type "`BitString`", the domain of the sort "`BitString`" is not sufficiently defined, since there is only a single constructor, "`Empty`". To be compatible with the implementation by hand given above, one has to add a new constructor "`Succ`" (sort "`BitString`" becomes isomorphic to natural numbers). Finally an equation is added in order to define completely the semantics of the operation "`Equal`" (notice that this equation takes advantage of priority between equations).

The new type definitions are the following:

```
type SequenceNumber is Boolean
   sorts SeqNum (*! implementedby SEQNUM comparedby CMP_SEQNUM
                    printedby PRINT_SEQNUM *)
   opns 0 (*! implementedby ZERO constructor *) : -> SeqNum
        1 (*! implementedby ONE constructor *) : -> SeqNum
        Inc (*! implementedby INC *) : SeqNum -> SeqNum
        Equal (*! implementedby EQ_SEQNUM *) : SeqNum, SeqNum -> Bool
   eqns
      forall X, Y:SeqNum
         ofsort SeqNum
            Inc (0) = 1;
            Inc (1) = 0;
         ofsort Bool
            Equal (X, X) = true;
            Equal (0, 1) = false;
            Equal (1, 0) = false;
```

```
endtype

type Bitstring is Boolean
   sorts BitString (*! implementedby BITSTRING comparedby CMP_BITSTRING
                       enumeratedby ENUM_BITSTRING printedby PRINT_BITSTRING *)
   opns Empty (*! implementedby EMPTY constructor *) : -> BitString
        Equal (*! implementedby EQ_BITSTRING *) : BitString, BitString -> Bool
    eqns
      ofsort Bool
         forall X, Y:BitString
            Equal (X, X) = true
endtype

type FrameType is Boolean
   sorts FrameType (*! implementedby FRAMETYPE comparedby CMP_FRAMETYPE
                       printedby PRINT_FRAMETYPE *)
   opns Info (*! implementedby INFO constructor *),
        Ack (*! implementedby ACK constructor *) : -> FrameType
        Equal (*! implementedby EQ_FRAMETYPE *) : FrameType, FrameType -> Bool
   eqns
      ofsort Bool
         forall X:FrameType
            Equal (X, X) = true;
            Equal (Ack, Info) = false;
            Equal (Info, Ack) = false
endtype
```

Then CÆSAR.ADT is run:

```
caesar.adt -english DATALINK
```

As the execution progresses, the following messages are displayed on the standard output:

```
-- caesar.adt 2.0 -- (c) IMAG/LGI -- C. Bard (caesar@imag.imag.fr) --

caesar.adt: syntax analysis of ``DATALINK''
caesar.adt: semantic analysis of ``DATALINK''
caesar.adt:    - gates binding
caesar.adt:    - processes binding
caesar.adt:    - types binding
caesar.adt:    - signature analysis
caesar.adt:    - sorts binding
caesar.adt:    - variables binding
caesar.adt:    - operations binding
caesar.adt:    - functionality analysis
caesar.adt: verification of ``DATALINK''
caesar.adt: compilation of the sorts of ``DATALINK''
caesar.adt: compilation of the operations of ``DATALINK''
caesar.adt: indentation of ``DATALINK''
```

Eventually a "`DATALINK.h`" file is generated. It is sometimes necessary to edit this file and modify some default parameters. For instance, to have only 100 different messages, one must replace:

```
#define ENUM_BITSTRING(X) for ((X) = 0; (X) < 255; ++(X))
```

by:

```
#define ENUM_BITSTRING(X) for ((X) = 0; (X) < 100; ++(X))
```

B.4 Generating the graph

When the user has prepared files "`DATALINK.lotos`", "`BOOLEAN.lib`" and "`DATALINK.h`", Cæsar can be run, for instance with the following options:

```
caesar -english -network -aldebaran DATALINK
```

As the execution progresses, the following messages are displayed on the standard output:

```
-- caesar 3.2 -- (c) IMAG/LGI & INRIA -- H. Garavel (hubert@imag.imag.fr) --

caesar: syntax analysis of ``DATALINK''
caesar: semantic analysis of ``DATALINK''
caesar:    - gates binding
caesar:    - processes binding
caesar:    - types binding
caesar:    - signature analysis
caesar:    - sorts binding
caesar:    - variables binding
caesar:    - operations binding
caesar:    - functionality analysis
caesar: restriction of ``DATALINK''
caesar: expansion of ``DATALINK''
caesar: generation of ``DATALINK''
caesar: optimization of ``DATALINK''
caesar: network dump for ``DATALINK''
caesar: simulation of ``DATALINK''
caesar:    - simulator production
caesar:    - simulator compilation
caesar:    - simulator execution
caesar:    - graph dump for ``DATALINK'' using ``aldebaran'' format
```

When the execution terminates, two files have been created by Cæsar:

- the "`DATALINK.net`" file contains the network corresponding to the alternating bit specification. This network has 14 places, 14 transitions and 8 variables.
- the "`DATALINK.aut`" file contains the graph corresponding to the alternating bit specification. With 100 messages, the graph has 164 201 states and 244 800 edges (this may be the biggest model ever seen for an alternating bit protocol!). It is generated in less than 11 minutes on a SUN 3/50 workstation and approximately 5 minutes and a half on a SUN 3/60. Each state takes 14 memory bytes. The number of generated states per second equals 255 on the SUN 3/50 and 500 on the SUN 3/60.

Note: all numerical data given here (number of places, transitions, states, edges and computation time) may change with next versions of Cæsar, Cæsar.adt and Aldébaran.

B.5 Reducing the graph

This graph must be checked to ensure that "good properties" are satisfied. More often than not, the graphs generated by Cæsar are too complex to be checked manually. Validation is done by applying abstraction criteria.

The simplest way to do this is to reduce the graph according to automata equivalences (such as strong, observational, testing, or trace equivalence, ...) by using ALDÉBARAN. Executing the following command:

```
aldebaran -omin DATALINK.aut
```

minimizes, according to observational equivalence, the graph generated by CÆSAR and displays the result on the standard output:

```
des (0, 200, 101)
(0, "Get  !0", 1)
(0, "Get  !1", 2)
(0, "Get  !2", 3)
...
(0, "Get  !99", 100)
(1, "Give !0", 0)
(2, "Give !1", 0)
(3, "Give !2", 0)
...
(100, "Give !99", 0)
```

This graph is small enough to be verified by hand. Its correctness is immediate, since each "`Get !`*i*" event can only be followed by a "`Give !`*i*" event.

B.6 Comparing two graphs

Another approach to verification consists in writing a short and obviously correct LOTOS specification which should be equivalent, under abstraction criteria, to the initial specification. Both graphs are built and compared with respect to automata equivalences. More often than not, the protocol is checked against the service it implements. For instance the service observationally equivalent to the alternating bit protocol, can be described as follows:

```
specification Datalink_Service [Get, Give] : noexit
behaviour
   Service [Get, Give]
where
   type Bitstring is
      sorts BitString (*! implementedby BITSTRING comparedby CMP_BITSTRING
                          enumeratedby ENUM_BITSTRING printedby PRINT_BITSTRING *)
   endtype

   process Service [Get, Give] : noexit :=
      Get ?M:BitString;
         Give !M;
            Service [Get, Give]
   endproc
endspec
```

The user should write this LOTOS specification to a "`DATALINK_SERVICE.lotos`" file and supply the following "`DATALINK_SERVICE.h`" file:

```
typedef unsigned char BITSTRING;
#define EMPTY() 0
#define CMP_BITSTRING(N1,N2) ((N1) == (N2))
#define EQ_BITSTRING(N1,N2)  ((N1) == (N2))
#define PRINT_BITSTRING(F,N) fprintf (F, "%u", N)
#define ENUM_BITSTRING(N) for(N = 0; N < 100; ++ N)
```

and execute the following commands:

```
caesar -english -network -aldebaran DATALINK
caesar -english -network -aldebaran DATALINK_SERVICE
aldebaran -oequ DATALINK.aut DATALINK_SERVICE.aut
```

The last command will display either "`TRUE`" or "`FALSE`", depending on the fact that both automata are observationally equivalent or not. For the alternating bit example, the answer is "`TRUE`".

B.7 Generating a reduced graph

In the alternating bit protocol graph, 83.5% of the edges are labelled by "τ". Most of them will "disappear" when the graph is reduced according to observation equivalence. Generating these edges and the corresponding states can be avoided by using *safety reduction* (see § 2.6). Practically, this is done by executing the following command:

```
caesar -english -network -aldebaran -safety DATALINK
```

For the alternating bit protocol, the graph obtained with safety reduction has only 40 031 states and 80 200 edges. It is obtained in about 4 minutes on a SUN 3/50. The reduced graph is observationally equivalent to the graph generated without reduction (this is not always true since safety reduction preserves trace equivalence, but not observational equivalence).

B.8 Designing verification tools

Since CÆSAR is an open system, users may want to write their own debugging tools to verify properties on the graphs generated by CÆSAR (for instance to find sequences of events leading to deadlocks, to prove that a sequence of events is possible, or inevitable, ...) Any contribution in these areas is welcome.

References

[Bar88] Christian Bard. *CÆSAR.ADT Reference Manual.* Laboratoire de Génie Informatique — Institut IMAG, Grenoble, August 1988.

[Fer88] Jean-Claude Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants.* Thèse de Doctorat, Université Joseph Fourier (Grenoble), May 1988.

[Fer90] Jean-Claude Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, 13(2–3):219–236, May 1990.

[FRV85] Jean-Claude Fernandez, Jean-Luc Richier, and Jacques Voiron. Verification of Protocol Specifications using the CESAR System. In Michel Diaz, editor, *Proceedings of the 5th IFIP International Workshop on Protocol Specification, Testing and Verification (Moissac, France)*, pages 71–90. North-Holland, June 1985.

[FSS83] Jean-Claude Fernandez, J. P. Schwartz, and Joseph Sifakis. An Example of Specification and Verification in CESAR. In G. Goos and J. Hartmanis, editors, *The Analysis of Concurrent Systems*, volume 207 of *Lecture Notes in Computer Science*, pages 199–210. Springer, September 1983.

[Gar89a] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[Gar89b] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th IFIP International Symposium on Protocol Specification, Testing and Verification (PSTV'90), Ottawa, Canada*, pages 379–394. North-Holland, June 1990.

[ISO87] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Draft International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Geneva, July 1987.

[ISO89] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Geneva, September 1989.

[QPF88] Juan Quemada, Santiago Pavón, and Angel Fernández. Transforming LOTOS Specifications with LOLA: The Parametrized Expansion. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 45–54. North-Holland, September 1988.

[QPF89] Juan Quemada, Santiago Pavón, and Angel Fernández. State Exploration by Transformation with LOLA. In Joseph Sifakis, editor, *Proceedings of the 1st Workshop On Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 294–302. Springer, June 1989.

[Rod88] Carlos Rodríguez. *Spécification et validation de systèmes en XESAR*. Thèse de Doctorat, Institut National Polytechnique de Grenoble, May 1988.

[RRSV87] Jean-Luc Richier, Carlos Rodríguez, Joseph Sifakis, and Jacques Voiron. Verification in XESAR of the Sliding Window Protocol. In Harry Rudin and Colin H. West, editors, *Proceedings of the 7th IFIP International Symposium on Protocol Specification, Testing and Verification (PSTV'87), Zurich, Switzerland*. North-Holland, May 1987.