NAME

lotos, LOTOS – Language of Temporal Ordering Specification

DESCRIPTION

LOTOS [BB88] is a formal specification language to describe communication protocols and distributed systems. It has been standardized by ISO/IEC in 1989 [ISO89]. The design of LOTOS was motivated by the need for a language with a high abstraction level and strong mathematical bases that would allow complex systems to be described precisely and unambiguously, then analyzed using formal methods supported by appropriate software tools. LOTOS features two clearly separated parts:

- The *data part* of LOTOS, intended to describe data structures, is based on the theory of abstract data types and algebraic specifications (especially the ActOne language defined by Ehrig and Mahr). In this approach, data structures are described by LOTOS *sorts*, which represent value domains, and LOTOS *operations*, which are mathematical functions defined on these domains. The meaning of operations is defined by algebraic *equations*. Value expressions are strongly-typed algebraic terms built from variables and operations. Sorts, operations, and equations are grouped in modules called *types*, which can be combined together using importation (with multiple inheritance), renaming, parametrization, and actualization. The underlying semantics is that of initial algebras.

- The *control part* of LOTOS is meant to describe the behaviour of concurrent processes that execute simultaneously, synchronize, and communicate using message-passing rendezvous. LOTOS is based on the process algebra approach for concurrency, and combines the best features of Milner's CCS and Hoare's CSP process calculi. It relies on a small set of basic operators, which express primitive concepts such as sequential composition, non-deterministic choice, guard, parallel composition, rendezvous, etc. These operators are used to build *behaviour expressions*, which are algebraic terms that describe the behaviour of concurrent systems, complex behaviours being obtained by combining simpler ones. The communication ports for rendezvous are called *gates*. Any behaviour expression can be given a name and reused several times by enclosing it in a *process* definition.

A comprehensive list of documents about the LOTOS language is available from http://cadp.inria.fr/tutorial

The remainder of this page is devoted to CADP-specific information about LOTOS. It gathers many technical details that, so far, were only known by CADP experts.

Note: LOTOS is a powerful yet involved language. Today, new users who do not know LOTOS already may prefer instead learning LNT (LOTOS New Technology), a modern language designed to be a replacement for LOTOS, simpler to learn but equally expressive. The present manual can yet be of interest to LNT users as well, since LNT is currently implemented by translation to LOTOS.

CADP TOOLS FOR LOTOS

The CADP toolbox provides four main tools to handle LOTOS specifications:

- **caesar.adt**(LOCAL) is a compiler for the data part of LOTOS. It translates LOTOS to C by generating an implementation for all the sorts and operations defined in a LOTOS specification.

- **caesar**(LOCAL) is a compiler for the control part of LOTOS. It translates LOTOS to Petri nets, and then to C code that can be used for, at least, three different purposes: (i) exhaustively explore all possible behaviours and generate the corresponding Labelled Transition System encoded in

aut(LOCAL) or **bcg**(LOCAL) format; (ii) explore possible behaviours on the fly by interfacing with the OPEN/CAESAR framework of CADP; (iii) connect the LOTOS specification to its environment by interfacing with the EXEC/CAESAR framework of CADP, e.g., to pilot an external device using the LOTOS specification as a controller.

- **caesar.indent**(LOCAL) is a pretty-printer that automatically reformats and indents LOTOS specifications.

- **lotos.open**(LOCAL) is a tool that allows running the OPEN/CAESAR application programs on LOTOS specifications. This tool is actually a shell script that invokes **caesar.adt** and **caesar**.

For historical reasons, and because the data part and control part of LOTOS are largely orthogonal, they are handled separately by the **caesar.adt** and **caesar** compilers, respectively. The landscape is however not so simple, as close connections exist between both compilers:

- Both compilers share a common front-end, which performs lexical and syntactic analyses, abstract tree construction, and static semantics checks (e.g., identifier binding, type checking, etc.). After these common, preliminary steps, each compiler performs additional checks for either the data or control part.

- The **caesar** compiler requires a *concrete* implementation in C for each *abstract* sort and operation defined in the LOTOS specification. Such an implementation can be automatically generated using **caesar.adt** or be manually written by the user.

  In most cases, it is advised to let **caesar.adt** produce the C code automatically, as the generated code will be both correct and, in most cases, highly efficient. For the most common classes of types (Booleans, enumerated types, bounded integers, records, unions, lists, trees, bit vectors, etc.), **caesar.adt** will generate optimal code (according to model checking demands, i.e., memory space first, then speed).

  On the other hand, manual writing gives full control on all implementation details and may thus be preferred for particular data structures, such as floating-point numbers, character strings, matrices, variable-length data with special encodings, etc. Manual writing also allows to reuse efficient C or C++ code libraries that already exist and can be imported in a LOTOS specification just by redeclaring their interface in the form of LOTOS abstract data types. Manual writing also allows to define functions that can perform "short-circuits" when evaluating their arguments (see the section below on CALL-BY-NEED EVALUATION).

  Both approaches can be combined, in the sense that certain LOTOS sorts and/or operations can be declared as *external* by the user and implemented manually; in such case, **caesar.adt** will only generate C code for the non-external sorts and/or operations, and will import the C code implementing those external sorts and/or operations.

  In any case, manual writing requires great care in order to produce correct implementations. There exist naming and interfacing conventions (described below) that must be strictly respected. Certain macro-definitions must be provided by the user to inform **caesar.adt** about key properties of the external sorts. Moreover, manually-written C functions are not allowed to have side effects, except in a very limited way (see the section below on SIDE EFFECTS for details). Examples of manually-written C code for the standard LOTOS types declared in the "**`$CADP/lib`**" directory (see files "**`X_*.lib`**") can be found in the "**`$CADP/incl`**" directory (see files "**`X_*.h`**" and "**`adt_*.h`**").

- Yet, the C code generated by **caesar.adt** for LOTOS specifications can also be used in a stand-alone manner, independently from the **caesar** compiler. In this approach, LOTOS is only used as a higher-level language for describing data structures with their related operations, from which **caesar.adt** produces lower-level C code.

  This idea has been applied in two large projects: the development of **caesar.adt** (version 4.0 and higher), most of which is written in LOTOS data types and which bootstraps itself, and the development of the **xtl**(LOCAL) compiler, the largest part of which consists of LOTOS code too.

- Note that both **caesar** and **caesar.adt** take great care to generate C code that does not cause warnings when given to C compilers with demanding compiling options. So, if warnings are emitted while compiling some C code generated by the CADP tools, it is likely that these warnings have their origin in the LOTOS specification itself (e.g., unused operations or unused operation parameters) or in the C code manually written by the user.

TOOLS GENERATING LOTOS CODE

The CADP toolbox provides two tools that produce LOTOS code:

- **fsp2lotos**(LOCAL) takes specifications written in Magee and Kramer's FSP (Finite State Processes) language and translates them into LOTOS.

- **lnt2lotos**(LOCAL) takes specifications written in LNT (LOTOS New Technology) and translates them into LOTOS.

Other tools have been developed, but not distributed as part of CADP:

- **chp2lotos** takes specifications written in Martin's CHP (Communicating Hardware Processes) and translate them into LOTOS. See http://cadp.inria.fr/software/05-b-chp2lotos.html for details.

- **flac** takes models written in the FIACRE formalism and translates them into LOTOS. See http://cadp.inria.fr/software/11-d-flac.html for details.

Note: Following the above remark that LOTOS is an involved language for humans, it is involved for translators too. In practice, it is much easier to generate LNT code rather than LOTOS code, and let the **lnt2lotos**(LOCAL) translator cope with the intricacies of generating correct LOTOS code.

RESTRICTIONS ON THE DATA PART OF LOTOS

Abstract data types, as they exist in LOTOS, are a rather unconstrained formalism that is difficult to execute efficiently and from which it is difficult to generate executable code. To handle algebraic specifications, there exist techniques based on rewriting or symbolic evaluation, but they are often slow and memory-intensive. This is a major problem in a model-checking context, because state-space exploration is especially demanding in terms of performance.

Therefore, in order to enable the use of **caesar.adt**(LOCAL) for translating LOTOS abstract data types into C code automatically, the following restrictions have been set, which turn algebraic specifications into (more operational) term-rewrite systems with priorities (see [Gar89c] and [GT93] for details).

These restrictions also introduce a suitable discipline in LOTOS, which, even if it is a strongly-typed language and owns a system of modules, remains poorly structured and hardly readable, as sorts, operations, and equations may appear in arbitrary order, with no guarantee that operations related to the same sort or equations related to the same operation will be gathered in the same module.

DISTINCTION BETWEEN CONSTRUCTORS AND NON-CONSTRUCTORS

The user must split LOTOS operations into two classes: the *constructors*, which are primitive operations, and the *non-constructors*, which are non-primitive operations defined using equations. To this aim, the user must explicitly indicate which operations are constructors by attaching a special comment to them (see below the section on SPECIAL COMMENTS FOR OPERATIONS).

The distinction enables LOTOS abstract data types to be efficiently implemented. However, this distinction does not exist in the standard definition of the language [ISO89], so that new constraints on the syntax and static semantics must be added.

CONSTRAINTS ON EQUATIONS

To be accepted by **caesar.adt**, each equation must match the non-terminal symbol E that is the axiom of the following BNF grammar:

```
E ::= [ G, ..., G => ] F (P, ..., P) = V
G ::= V | V = V
P ::= X | C | P C P | C (P, ..., P) | P of S | (P)
V ::= X | C | V C V | C (V, ..., V)
    | F | V F V | F (V, ..., V) | V of S | (V)
```

where:

- square brackets denote an optional element (here, premisses)
- C is a terminal symbol denoting a constructor identifier (either constant, infix binary, or prefix)
- E is a non-terminal symbol denoting a well-formed equation
- F is a terminal symbol denoting a non-constructor identifier (either constant, infix binary, or prefix)
- G is a non-terminal symbol denoting an equation premiss (either a Boolean guard or an equality test)
- P is a non-terminal symbol denoting a pattern (i.e., a value expression that does not contain any non-constructor identifier)
- S is a terminal symbol denoting a sort identifier
- V is a non-terminal symbol denoting a value expression
- X is a terminal symbol denoting a value identifier (i.e., a variable)

Any variable occurring in a guard G or in the right-hand side expression V must also occur in (at least one of) the left-hand side patterns P.

In such case, one says that equation E *defines* the non-constructor F.

All constructors must be *free*, meaning that the above syntax does not allow constructors to be defined by equations.

If a specification contains non-free constructors, it can be transformed into an equivalent specification containing only free-constructors. This can be done in a systematic way (see Section 1.7 of [Gar89c]).

CONSTRAINTS ON SORTS AND CONSTRUCTORS

- If a sort *S* is not external, there must exist at least one constructor returning a result of sort *S*. Otherwise, a compiler warning will be emitted and **caesar.adt** will consider that *S* is implicitly external, so that no C code will be generated to implement this sort.

- If a sort *S* is not external, then all the constructors returning a result of sort *S* should be non-external operations.

- If a sort *S* is external, then all the constructors returning a result of sort *S* should be external operations.

- To enforce a modular specification style that LOTOS does not encourage by default, each constructor returning a result of sort *S* should be declared in the same LOTOS type as *S*, or a compiler warning will be emitted.

- Each sort *S* must be *productive*, i.e., there must exist at least one *ground term* that is *well-typed* (a ground term being an an algebraic term that contains only constructors, and neither free variables nor non-constructors). For instance, a sort *S* with only one constructor defined as "*Succ* **:** *S* **->** *S*" is unproductive (technically, its initial algebra is empty).

CONSTRAINTS ON NON-CONSTRUCTORS AND EQUATIONS

- If a non-constructor *F* is not external, there should exist at least one equation defining *F*. Otherwise, a compiler warning will be emitted, and **caesar.adt** will consider that *F* is implicitly external, so that no C code will be generated to implement this operation.

- If non-constructor *F* is external, then no equations should define *F* (if such equations have already been written, it is advised to preserve them by commenting them out rather than simply deleting them).

- To enforce a modular specification style that LOTOS does not encourage by default, each equation defining a non-constructor *F* should be located in the same LOTOS type as *F*, or a compiler warning will be emitted.

CONSTRAINTS ON RENAMING

- Sort renaming is supported as follows. If a sort *S2* renames a sort *S1*, then **caesar.adt** generates no C code for implementing *S2*, and the CADP tools simply replace all occurrences of *S2* by *S1* in the C code they generate, meaning that both sorts share the same implementation in C.

- Consequently, it is forbidden to define constructors that return a result of the renaming sort *S2*, meaning that one cannot modify a renamed sort by adding constructors to its renaming sort(s). See item #903 in file "**$CADP/HISTORY**" for details.

- Operation renaming is supported as follows. If an operation *F2* renames an operation *F1*, then **caesar.adt** generates no C code for implementing *F2*, and the CADP tools simply replace all occurrences of *F2* by *F1* in the C code they generate, meaning that both operations share the same implementation in C.

- Consequently, it is forbidden to write equations that define the result of the renaming operation *F2*, meaning that one cannot modify a renamed non-constructor by adding equations to its renaming operation(s). See item #2190 in file "**$CADP/HISTORY**" for details.

CONSTRAINTS ON ACTUALIZATION

- Type actualization (i.e., on the one hand, generic types parameterized by formal sorts, formal operations, and/or formal equations, and, on the other hand, actualized types obtained by instantiating

generic types) is supported only in a partial way. The CADP tools indeed parse and statically check such type definitions, but **caesar.adt** later ignores them silently and generates no C code to implement them.

- As a consequence, **caesar.adt** does not handle certain types of the LOTOS standard library (e.g., *Element* and *Set*). To address this limitation, the user has to flatten actualized types manually by making a copy of parameterized and substituting all formal elements by the corresponding actual ones.

REWRITE STRATEGY

The C code generated by **caesar.adt** for evaluating LOTOS value expressions uses a term-rewrite strategy that can be characterized by the two following rules:

call-by-value (or functional evaluation):
When several subterms can be rewritten simultaneously, the innermost ones are rewritten first.

decreasing priority between equations:
When, for a given subterm, several equations apply simultaneously, the equation that occurs first in the LOTOS source text is chosen.

Notice that such strategy is not fully deterministic, since it does not specify in which order subterms at the same nesting level (e.g., the various actual arguments of a function call) will be evaluated. The decision is deferred to the C compiler, which may take advantage of such degree of freedom to optimize machine-code generation. Therefore, the user should not expect leftmost subterms to be rewritten first.

When needed, the call-by-value strategy can be changed to a call-by-need one by using external functions and C macro-definitions (see the section below on CALL-BY-NEED EVALUATION).

With respect to the well-known theory of term-rewrite systems, the following remarks can be made:

Completeness:
The issue of completeness arises when there are *not enough* equations to define a non-constructor. For instance, let *F* be a non-constructor defined by a single equation "*F* (*true*) = *true*"; the value "*F* (*false*)" remains undefined. In standard LOTOS, this value cannot be reduced and is added to the initial algebra of the *Bool* sort, thus leading to Booleans having more than two values. This problem is avoided in the CADP tools, as the C code generated by **caesar.adt** raises a run-time error when trying to evaluate a call to a non-constructor whose result is not defined by any equation matching the values of the actual parameters supplied with the call. The **-debug** option of **caesar.adt** can be used to print these values when a run-time error occurs. Therefore, **caesar.adt** extends LOTOS with the concept of partially-defined functions, which are implemented using (uncatchable) exceptions.

Confluence:
The issue of confluence arises when there are *too many* equations to define a non-constructor. For instance, let *F* be a non-constructor defined by the two following equations "*F* (*x*) = *true*" and "*F* (*x*) = *x*"; the value "*F* (*false*)" can be rewritten either to *true* or *false*. In standard LOTOS, such a definition of *F* implies that both values *true* and *false* are equal, thus leading to Booleans having a single value. This problem is avoided with the decreasing priority between equations enforced by **caesar.adt**, which ensures that the evaluation of value expressions is deterministic. Moreover, **caesar.adt** warns about equations that are never used because they are overriden by equations of

higher priority (note that, when premisses are used, such detection of unused equations cannot be done systematically).

Termination:

The LOTOS equations are expected to satisfy the termination property, but **caesar.adt** makes no attempt to check this property (anyway, only sufficient conditions for termination could be checked). If the user writes non-terminating equations such as "*F* (*X*) **=** *F* (*X*)", the generated C code will loop forever or cause a stack overflow when executed. In such case, the C debugger and/or the **-trace** option of **caesar.adt** can be used to understand the reason of the problem.

In principle, if the LOTOS equations are written to be confluent and terminating, then all rewrite strategies (including the one implemented by **caesar.adt**) should produce the same result when evaluating a given term. However, in the case of partially-defined non-constructors, there will be a difference as **caesar.adt** will trigger a run-time error. Moreover, seeking for confluent equations prevents from taking advantage of priority between equations, i.e., from using the "if-then-else" facility offered by priority, which often leads to shorter specifications and more efficient implementations. A striking example of the benefits of priority is the ability to define an equality operation *eq* for any sort *S* using only two equations:

forall x, y **:** *S*
ofsort *Bool*
x *eq* x = *true* **;**
x *eq* y = *false*

Examples of issues that may arise when adding new operations to existing LOTOS sorts are given in Section 2.a of [Gar13].

RESTRICTIONS ON THE CONTROL PART OF LOTOS

The CADP tools accept a very large class of LOTOS behaviour expressions and process definitions: all behavioural operators are accepted and value expressions are handled as well. However, there are a few restrictions dictated either by the need to produce finite-state transition systems, or by the efficiency of the translation algorithms. Some restrictions also depend on the kind of analysis performed (e.g.,exhaustive state-space exploration vs sequential C code generation).

STATIC-CONTROL CONSTRAINTS

For efficiency reasons (namely, to ensure that the interpreted Petri nets generated by the translation have at most one token per place), the CADP tools lay additional constraints on LOTOS behaviour expressions. Only a subset of LOTOS behaviour expressions is accepted, those that satisfy the so-called *static control* property: any recursive call to a process is prohibited if it occurs in any of the five following contexts:

- on the left- or on the right-hand side of a parallel composition operator ("**||**", "**|||**", "**|[...]|**")
- through a "**par**" operator
- through a "**hide**" operator
- on the left-hand side of a "**>>**" (enable) operator
- on the left-hand side of a "**[>**" (disable) operator

See Section 3.1 of [Gar89b] for a formal definition of static control constraints, illustrated with detailed examples. Notice that a call to a recursive process is not necessarily a recursive process call (although the converse is true).

In practice, LOTOS specifications that do not satisfy these constraints often exhibit a non-regular behaviour and, thus, cannot be translated to finite-state transition systems.

FINITE-DOMAIN CONSTRAINTS

The three following LOTOS behaviour expressions require a nondeterministic selection, in the domain of sort *S*, of some value to be stored in variable *x*:

- "**choice** *x* **:** *S* **[]** ..."
- "**exit (any** *S***)**"
- "*G* **?***x* **:** *S* **;** ..." (provided that this input action on the visible gate *G* is not synchronized with an ouput action "*G* **!***v* **;** ..." that would impose value *v* to *x*).

In general, to execute such behaviour expressions, the CADP tools enumerate all possible values in the domain of sort S and try all possible execution paths, one for each value. That is, these tools perform *concrete* rather than *symbolic* execution.

Enumerating the domain of sort *S* is only possible if this domain is finite or, in the case of infinite sorts (such as lists, strings, trees, etc.), if the enumeration is restricted to a finite subset (e.g., only those lists whose length is less than five).

If the domain of sort *S* is infinite, or if it is so large that the user wants to restrict it to a smaller subset in order to avoid state-space explosion issues, or if *S* is an external sort, the user must provide an *iterator* for sort S, i.e., a fragment of C code that enables a variable *x* of sort *S* to enumerate all possible values in the chosen subset.

If sort *S* is never used in any of the three kinds of behaviour expressions listed above, then no iterator for sort *S* is required, as the domain of *S* will not be enumerated. Thus, sorts with infinite or large domains are perfectly accepted, as long as no attempt is made at enumerating their domains. Notice also that, by default, *caesar.adt* automatically generates iterators for all finite, non-external sorts (and bounded iterators for infinite, non-external sorts isomorphic to natural numbers).

Notice that, if a LOTOS specification satisfies the static-control property and the finite-domain constraints for all its sorts that must be enumerated, then its labelled transition system is necessarily finite. However, even if finite in theory, it may still be too large to be generated in practice.

CLOSED-WORLD MODELLING

As mentioned in the previous section, in the particular case of an input action "*G* **?***x* **:** *S* **;** ..." not synchronized, on the visible gate *G*, with a corresponding output action, the CADP tools will enumerate all possible values in the domain of sort *S*. Such enumeration is indeed conformant with the operational semantics of LOTOS, in which both terms "*G* **?***x* **:** *S* **;** *B*" and "**choice** *x* **:** *S* **[]** *G* **!***x* **;** *B*" are equivalent modulo strong bisimulation.

Note: in certain cases, value enumeration could be avoided. For instance, the LOTOS terms "*G* **?***x* **:** *S* **[***x***=***v***]** **;** *B*" and "*G* **?***x* **:** *S* **;** **[***x***=***v***]** **->** *B*" are strongly equivalent to the term "*G* **!***v* **;** *B*" and thus do not require enumerating all values in the domain of sort *S*. However, such optimizations are not yet implemented in the **caesar** compiler.

To be precise, value enumeration only takes place when performing exhaustive state-space exploration (namely, when using **caesar** to generate a Labelled Transition System encoded, e.g., in the **aut**(LOCAL) or **bcg**(LOCAL) format), or when performing on-the-fly exploration using one of the tools based upon the

OPEN/CAESAR framework. However, when **caesar** is used to generate C code that will be connected to a real environment using the EXEC/CAESAR framework, such iteration does not take place and is replaced by a call to a so-called "gate function" that will actually input a value from the real environment, rather than enumerating all possible values that could be input.

To restrict value enumeration on an infinite or excessively-large sort domain, the definition (or redefinition) of an ad-hoc iterator (mentioned in the previous section) is sometimes non feasible, because it would affect all other places in the LOTOS specification where this sort is used. In such case, alternative techniques must be used.

If the sort domain is finite or isomorphic to natural numbers (which **caesar.adt** implements using a finite interval), the enumeration can be straightforwardly restricted by adding a predicate to the input action, i.e., "*G* **?***x*:*S* **[***f*(*x*)**] ;** ...", so that only those values *v* satisfying *f*(*x*) will be selected. This may cause an overhead in CPU time, but it is usually negligible.

Another technique to restrict the enumeration is to synchonize each input action with one or many output actions defining the possible values that can be sent by the external environment in which the LOTOS specification is evolving. This requires to introduce, in the LOTOS specification, an extra process that runs in parallel with the remainder of the specification and synchronizes on input gates (at least). This added process describes scenarios of input (and possibly, output) actions exchanged between the system described by the LOTOS specification and its environment. In model-checking terminology, a specification extended with a model of the environment is often called a *closed-world* description, as opposed to an *open-world* description, where no assumption is made about the environment. In the LOTOS literature, using parallel composition as a logical conjunction to specify additional properties is known as a *constraint-oriented* specification style.

OTHER EXTENSIONS TO LOTOS

For convenience, a few enhancements to LOTOS have been implemented in the CADP tools. These extensions deviate from the standard definition of LOTOS [ISO89] but they can be disabled by giving the command-line option "**-iso**", which enforces the standard definition of LOTOS. These extensions are the following:

1. In standard LOTOS, "i" and "I" are reserved keywords, which forbids to declare any identifier named "i" or "I". The CADP tools relax this constraint by turning "i" and "I" into reserved identifiers for gates only. This makes it possible to declare, e.g., variables or operations named "i" or "I". See item #1512 in file "**$CADP/HISTORY**" for details.

2. In standard LOTOS, the sorts of formal variable parameters are not used to resolve overloading ambiguities in actual value expressions of process instantiations. For instance, if a process P is declared with a formal variable parameter X of sort Nat, invoking this process with the actual value zero requires to write "0 of Nat" rather than simply "0" if sorts Nat and Int both have a constant 0. To address this issue, the CADP tools use extended type-checking rules that allow to write simply "0" rather than "0 of Nat", because the declaration of X is taken into account to infer that 0 has sort Nat. See item #1803 in file "**$CADP/HISTORY**" for details.

3. In standard LOTOS, the sorts of the value parameters and exit results of the top-level specification must be declared in the section of global definitions (located before the "behaviour" keyword). When this section is empty, the CADP tools look into the local definitions (located after the "behaviour" keyword) to find the declaration of these sorts. See item #1875 in file "**$CADP/HISTORY**" for details.

4. The CADP tools use a modified semantics for flattening parameterized types, rather than the standard (seemingly questionable) semantics of LOTOS. The modified semantics is described in [GS95a]; in practice, however, the differences between both semantics should not matter too much, given that parameterized types are not fully implemented by the CADP tools. See item #346 in file "**$CADP/HISTORY**" for details.

5. The CADP tools recognize a special comment of the form "**(*! atomic *)**" that, if present in a LOTOS specification (usually, after some ">>" operator), removes all hidden transitions (labelled "**i**") created by the ">>" operators. Such special comment is mostly used for the translation of LNT to LOTOS, and purposedly deviates from the standard operational semantics of LOTOS behaviour expressions. See item #1327 in file "**$CADP/HISTORY**" for details.

OTHER DEVIATIONS FROM THE LOTOS STANDARD

This section lists a few minor differences between the LOTOS standard [ISO89] and the way it is implemented in the CADP tools:

- In the labelled transition systems generated by the CADP tools, the auxiliary gate used for sequential composition, which is noted using the Greek letter "delta" in the standard semantics of LOTOS, is noted "**exit**" (always in lower case). This decision is justified by the need to use Latin-alphabet letters only, by the fact that gate "delta" can only be produced by the "**exit**" operator of LOTOS, and by the fact that no user-defined gate can be named "**exit**" since this name is a reserved keyword of LOTOS (which is not the case of "delta").

- In the labelled transition systems generated by the CADP tools, the invisible (or internal) gate, which is usually noted using the Greek letter "tau" in the scientific literature, is noted "**i**" (always in lower case). The standard semantics of LOTOS also uses the same "**i**" notation.

- When a LOTOS specification is defined with formal variable parameters, the standard semantics of LOTOS does not say much on how these parameters should be handled. Certain LOTOS implementations (e.g., the TOPO tool) reject such parameterized specifications. The CADP tools (namely, **caesar**) also reject them, unless the "**-root**" option is used to provide actual value expressions that instantiate the formal variable parameters.

  Note: Earlier versions of **caesar** handled formal variable parameters by enumerating the domains of their respective sorts, but this is no longer the case. See item #2014 in file "**$CADP/HISTORY**" for details.

- The CADP tools may differ from [ISO89] when handling process calls in which two (or more) actual gate parameters are identical. For instance, the following behavior:

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

  is strongly equivalent to "**stop**" according to the standard semantics of LOTOS, because the body of process *P* is evaluated *after* applying the gate relabelling function that maps *a* to *c* and *b* to *c*. However, this behaviour is equivalent to "*c* **; stop**" according to the CADP tools because the expansion phase of **caesar** implements a *call-by-value* semantics that applies the gate relabelling function *before* evaluating the body of *P*, which becomes therefore "*c* **; stop ||** *c* **; stop**". Both semantics coincide in most cases, and may only diverge if a process is called with identical actual gate parameters and if some of the corresponding formal gate parameters are synchronized

together in the body of the process. **caesar** always emits a warning message if the relabelling function is not injective; in the absence of such a warning, the call-by-value semantics conforms to the standard LOTOS semantics.

There are a few other subtle differences, which should not matter to most users:

- When declaring an infix operation "**_***F***_ :** *S1*, ..., *Sn* **->** *S*", the standard syntax of LOTOS allows spaces to be inserted between *F* and its two surrounding underscores (see *operator-descriptor* in Section 6.2.5 of [ISO89]). Such syntax is difficult to parse; at the expense of internal complexity, the CADP tools parse it properly but do not check that identifier *F* is not a reserved keyword. Though it may be possible to declare an infix operation whose name is a keyword, any further occurrences of this operation will be syntactically rejected.

- In Section 7.3.3 of [ISO89], it is acknowledged that the syntax of LOTOS is ambiguous, as it does not allow to make the difference between a variable (i.e., value identifier) and a nullary operation occuring in a value expression. The disambiguation can only be done later using static semantics information, but the remainder of the section does not state clearly how to bind an identifier *x* when both a variable and a nullary operation exist with both the same name *x* and both an appropriate sort.

  The CADP tools proceed in two steps. First, they search whether a variable named *x* exists; if so, the identifier *x* is bound to this variable, whatever the sort of the variable (a type-checking error will occur later if the variable does not have the proper sort). Second, if no variable named *x* exists, they perform operation binding and overloading resolution, which will have the effect of binding the identifier to a nullary operation *x* if it exists with an appropriate sort. Thus, preference is always given to variables over nullary operations, which seems compatible with the principle of binding with the innermost operation definition when resolving operation overloading.

- In Section 7.3.4.5.e of [ISO89], requirement (e1) was strengthened to state that, when the clause "**accept**...**in**" is missing, the behaviour expression *B1* occurring in "*B1* **>>** *B2*" must have the functionality "*exit*".

- The decision to implement the "**library**" declaration using file inclusion (see below the section on LIBRARY FILES) leads to slightly different rules for the visibility of identifiers declared in libraries.

FILENAMES

This section describes the various files related to LOTOS specifications and their implementation.

LOTOS FILES

The files containing LOTOS specifications should have a "**.lotos**" suffix. This is the preferred convention when using CADP. However, to ease the use of non-CADP tools, the alternative suffixes "**.lot**" and "**.l**" are also tolerated, i.e., properly recognized by the CADP tools.

LIBRARY FILES

The directive "**library** *f1*, ..., *fn* **endlib**" that is present in the standard LOTOS definition [BB88] without explicit meaning is implemented using file inclusion by the CADP tools, as follows: when such a clause is encountered, the CADP tools look for files named "*F1***.lib**", ..., "*Fn***.lib**" (where the strings "*F1*, ..., *Fn*" are obtained from "*f1*, ..., *fn*" by turning lower-case letters to upper case, since LOTOS identifiers are case insensitive whereas file names are often case sensitive) and include these files, in that order, at the place

where the directive "**library**" occurs (the directive itself is replaced by the contents of the files). This is similar to the effect of a "**#include**" directive with the C preprocessor.

The "**.lib**" files are searched first in the current directory, or else in the **$CADP/lib** directory, which contains a collection of predefined LOTOS libraries. If a file cannot be found using these search rules, a fatal error is reported. More elaborate search rules can be obtained by creating symbolic links to library files in the current directory.

Library inclusion can be transitive, meaning that "**.lib**" files may contain "**library**" directives. Circular inclusions are prohibited.

Contrary to the standard LOTOS definition [BB88], a "**library**" directive may occur at any place (i.e., not only in data-type definitions) and may contain arbitrary LOTOS code fragments (e.g., process definitions, rather than type definitions only). These extensions help splitting large LOTOS specifications into several files and enable one to develop reusable libraries of processes (e.g., buffers, shared variables, etc.).

INCLUDE FILES

To be processed by the CADP tools, a LOTOS specification contained in a given file *filename***.lotos** should come together with an implementation (in the C language) of the LOTOS sorts and operations. This implementation must be provided in a file named *filename***.h** located in the current directory.

This file can be either written by hand or automatically generated using **caesar.adt**(LOCAL). In the latter case, both approaches can be combined, as the user may (optionally) provide two additional files:

- a file named *filename***.t** (where "**.t**" stands for "types") that contains manually-written C code for implementing certain LOTOS sorts,
- a file named *filename***.f** (where "**.f**" stands for "functions") that contains manually-written C code for implementing certain LOTOS operations.

Each of these files, if present in the current directory at the moment when **caesar.adt** is invoked, will be #included in the *filename***.h** file generated by **caesar.adt**. If both files are present, *filename***.t** will be included before *filename***.f**.

The motivation behind the "**.t**" and "**.f**" files is to offer the possibility to introduce manually-written C code for certain sorts and/or operations, and also to customize the C code produced by **caesar.adt** without modifying the contents of the "**.h**" file generated by **caesar.adt**.

In particular, the files "**$CADP/incl/X_*.h**", which contain manually-written C code to implement standard LOTOS types, should be included (using a "**#include**" directive) in the "**.t**" file if the LOTOS specification imports the corresponding types defined in "**$CADP/lib/X_*.lib**".

The CADP tools generate auxiliary C programs that include the "**.h**", "**.t**", and "**.f**" files, then compile these programs and execute them to obtain information about the types defined in these files. To be more precise, **caesar.adt** compiles only the "**.t**" file if it exists, while **caesar** compiles only the "**.h**" file if it exists (keeping in mind that, if the "**.h**" file has been generated using **caesar.adt**, it should include the "**.t**", and "**.f**" files if they were present at the time **caesar.adt** was invoked).

So doing, the CADP tools check that the manually-written C code contained in these files is valid according to the definition of the C language. However, the CADP tools cannot verify whether this C code correctly implements the intended meaning of the sorts and operations defined in the LOTOS specification. In practice, such mistakes can be difficult to detect. When debugging such problems, one can safely assume that the errors are in the C code manually written by the user, rather than in the C code generated by the CADP tools.

Notice that the "**-external**" option of **caesar.adt** is helpful, as it generates, for the "**.t**", and "**.f**" files, skeletons that can be later filled in by the user.

TAILORED CODE GENERATION FOR DATA TYPES

caesar.adt analyzes LOTOS sorts and their constructors and recognizes various classes of sorts that can be implemented efficiently, if not optimally [Gar89c,GT93]. These classes are the following:

singleton sorts:
They have only one element; such sorts often arise when performing value abstraction, i.e., replacing a complex sort by a singleton.

enumeration sorts:
They correspond to the usual notion of enumerated types.

numeral sorts:
They correspond to the usual notion of (infinite) natural numbers, which **caesar.adt** restricts to a finite range 0...(*n*-1). The value of *n* is determined as follows: if the macro **CAESAR_ADT_HASH_***N* is defined in the "**.t**" file (see below the section on SPECIAL SYMBOLS for details), then *n* is given by the value of this macro; else if **caesar.adt** is invoked with option **-numeral**, then *n* is given by the integer value following this option; otherwise *n* is set to 256 (meaning that, by default, values of a numeral sort are stored using a single byte to avoid wasting memory during state-space exploration).

tuple sorts:
They correspond to the usual notion of record types.

ordinary sorts:
They correspond to the usual notion of discriminated union types.

Even when automatic code generation is used, it is still possible to instruct **caesar.adt** to adapt its C code generation to particular requirements. This can be done in four (non mutually exclusive) ways:

- By passing specific options on the command line (e.g., **-debug**, **-infix**, **-macro**, **-prefix**, **-trace**) to modify the generated C code;

- By declaring certain LOTOS sorts as *external* (see the section below on SPECIAL COMMENTS FOR SORTS) and providing the C code that implements these sorts in *filename***.t**.

- By declaring certain LOTOS operations as *external* (see the section below on SPECIAL COMMENTS FOR OPERATIONS) and providing the C code that implements these operations in *filename***.f**.

- By inserting, in the aforementioned file *filename***.t**, various directives that influence the way certain LOTOS sorts are implemented in C. See below the section on SPECIAL SYMBOLS for details.

LOTOS AND C IDENTIFIERS

To execute LOTOS specifications, one must establish a correspondence between *abstract* data types (i.e., LOTOS sorts and operations) and their *concrete* implementations (i.e., C types and functions or macro-definitions). There is no direct, one-to-one correspondence, because: (i) LOTOS operation identifiers may

contain special characters (e.g., "#", "@", etc.) that are not allowed in C identifiers; (ii) LOTOS allows a sort and an operation to share the same identifier, whereas types and functions must have distinct identifiers in C; (iii) a LOTOS operation identifier may be overloaded and thus be implemented by different C functions or macro-definitions, each of which must have unique identifier in C. Notice also that, although LOTOS identifiers are case-insensitive, C identifiers are case-sensitive.

In the sequel, a *valid C identifier* denotes any identifier that obeys the rules of the C language; such an identifier must be different from the reserved keywords of C (i.e., "**if**", "**while**", etc.), but should also be different from POSIX names (e.g., "**exit**", "**FILE**", "**fopen**", "**malloc**", "**printf**", etc.). Moreover, it should not start with a prefix used by the CADP tools, among which "**ADT_**", "**adt_**", "**CAESAR_**", "**caesar_**", "**BCG_**", "**bcg_**", "**XTL_**", and "**xtl_**". It may be worth that users choose another prefix and use it systematically before their own C identifiers.

The standard definition of LOTOS makes no provision for mapping abstract data types to concrete ones, so that interface conventions often differ across tools that implement LOTOS. The next section describes the conventions adopted by the CADP tools for LOTOS.

SPECIAL COMMENTS

Mapping information is inserted directly in the LOTOS specifications by means of *special comments*. Such special comments have the following syntax "**(*! ... *)**" and are thus a subset of ordinary LOTOS comments "**(* ... *)**". But, unlike ordinary comments, their content is meaningful and parsed.

Note: users should carefully respect the syntax of special comments because they are scanned at a lexical rather than syntactic level. Thus, in case of syntax errors in a special comment, the error-recovery actions performed by the SYNTAX compiler-generator system used to build the CADP tools may be less intuitive than usually, yielding a cascade of cryptic error messages.

There are two main classes of special comments: those attached to LOTOS operations, and those attached to LOTOS sorts. Unless stated otherwise, special comments are optional. If present, they must occur immediately after the declaration of the sort or the operation they are attached to.

It is not permitted to attach a special comment to a LOTOS sort or operation defined by renaming an other sort or operation, because **caesar.adt** generates no C code for renamed sorts and operations, and just implements these as synonyms of the sorts and operations they rename.

It is not permitted to attach a special comment to a LOTOS sort or operation that is either formal (i.e., generic) or defined by actualizing an other sort or operation, because **caesar.adt** generates no C code to implement parameterized data types.

Numerous examples of special comments can be found by examining the predefined LOTOS libraries contained in directory "**$CADP/lib**". The simplest example is probably the *BIT* type declared in "**$CADP/lib/X_BIT.lib**" and implemented in "**$CADP/incl/X_BIT.h**".

SPECIAL COMMENTS FOR SORTS

A special comment attached to the declaration of a LOTOS sort *S* has the following syntax (where square brackets mean optional elements):

```
(*!
 [ implementedby N ]
 [ comparedby N1 ]
 [ iteratedby N2 and N3 ]
 [ printedby N4 ]
 [ list ]
 [ external ]
*)
```

where *N*, *N1*, *N2*, *N3*, and *N4* are valid C identifiers. Even if these elements are optional, their order cannot be modified. Spaces and blanks are allowed inside special comments. All letters in the words "**implementedby**", "**comparedby**", ..., "**external**" can be either in lower or upper case; it is advised to use the lower case only.

The meaning of special comments is the following:

- The attribute "**implementedby** *N*", if present, indicates that the abstract LOTOS sort *S* is implemented by a concrete C type named *N*. The simplest way is to declare *N* using a "**typedef**" instruction, but declaring *N* as a macro-definition is also possible.

  Type *N* can be any C type whose values occupy a fixed number of bits, so that values of type *N* can be copied using a standard C assignment operator "**=**". This obviously allows (signed and unsigned) integers and characters, reals, and enums. This also allows struct and union types, for which assignment is also permitted in C; the CADP tools take into account the case of certain C compilers that copy each field of structs or unions, but do not copy the padding bits that may exist between these fields: in such case, assignment is replaced by a call to the **memcpy()** function to avoid creating uncanonical values (i.e., identical struct or union values having a different representation in memory). Array types are not allowed because assignment is not permitted for them. Finally, pointer types (to, e.g., vectors, matrices, linked lists, binary trees, etc.) are also allowed, provided that the allocated memory cells to which they point are assigned only once and not modified later. Thus, any function with an argument of pointer type should not modify this argument but first make a copy of it and modify this copy only. Under this condition, structural sharing (i.e., several pointers referring to the same memory cell) will work correctly. See below the section on SPECIAL SYMBOLS for a discussion on structural sharing, as well as other points such as bit fields.

  If the attribute "**implementedby**" is absent, the CADP tools will generate a unique C identifier for the implementation of *S*. Nothing else should be assumed about this identifier. In particular, it may change with future versions of the CADP tools.

  The option **-comments** of the CADP tools emits a warning for each unspecified "**implementedby**" attribute.

  The option **-map** of the CADP tools produces a file that gives the name correspondence between abstract LOTOS sorts and concrete C types.

- The attribute "**comparedby** *N1*", if present, indicates that the equality comparison between two abstract values of sort *S* is implemented by a concrete C function (or macro-definition) named *N1*. This function takes two arguments of type *N* and returns a result of type **int**, which is zero if both arguments are equal, or non-zero (yet not necessarily one) if both arguments are different.

  If the attribute "**comparedby**" is absent, a unique C identifier is generated for *N1*, with no other guarantee.

  Notice that function *N1* should always exist, even if sort *S* has no associated comparison operation (e.g., "*eq* **:** *S*, *S* **->** *Bool*") defined explicitly in the LOTOS specification. Indeed, one must compare values of sort *S* whenever they occur in equation premisses ("*X***=***Y* **=>** ..."), in Boolean guards ("**[***X***=***Y***]** **->** ..."), or in rendezvous with value matching ("*G***!***X* ... **||** *G***!***Y* ...").

  For simple types (e.g., integers or enumerated types), function *N1* is often a mere equality test (noted "**==**" in C). However, for more complex types, computations may be more involved, e.g.,

comparison of two real numbers up to a given precision, or deep comparison of linked data structures.

The definition of function *N1* should always take into account the cases where one or both of its arguments is equal to a bit pattern consisting only of zeros, even if such a bit-zero pattern does not normally belong to the admissible values of type *N* (for instance, if *N* is a pointer type denoting C character strings, the case of the **NULL** value should nevertheless be considered). This is due to the fact that **caesar** initializes all simulator variables to an undefined value represented by a bit-zero pattern, and also resets variables to this undefined value as soon as they are no longer used.

- The attribute "**iteratedby** *N2* **and** *N3*", if present, indicates that the enumeration of all values in the domain of sort *S* (or in a finite subset if this domain is infinite or too large) is implemented by two macro-definitions named *N2* and *N3* (see above the section on FINITE-DOMAIN CONSTRAINTS for a discussion on iterators). The macro *N2* has no argument and returns a constant value of type *N*, which is the "first" value to be enumerated in the domain of *S*. The macro *N3* takes one argument, which is an l-value of type *N*, and returns a result of type **int**; *N3* tries to advance its argument to the "next" value in the domain of *S* and returns a non-zero result if such a next value exists, or a zero result if the argument was already equal to the "last" value in the domain of *S*. Therefore, enumerating the domain of *S* can be achieved using the following fragment of C code:

```
N x;
x = N2;
do {
  ...
} while (N3 (x));
```

For an external sort, the iteration macros *N2* and *N3* have to be provided by the user in the "**.t**" file. Notice that, for the external sorts *Bit*, *Bool*, *Char*, *Int*, and *Nat* defined in the "**$CADP/lib/X_*.lib**" files, manually-written iterators are provided in the corresponding "**$CADP/incl/X_*.h**" and "**$CADP/incl/adt_*.h**" files. For the two latter sorts, the bounds of the iterator can be modified by defining the macros **ADT_INF_NAT**, **ADT_SUP_NAT**, **ADT_INF_INT**, and/or **ADT_SUP_INT** before including the "**X_*.h**" file.

For a non-external sort, these macros are generated automatically by **caesar.adt** if and only if the domain of the sort is finite; however, these macros are also generated for numeral sorts, even if the domain of these sorts is infinite, but **caesar.adt** restricts the iterations to a finite range (by default, 0...255).

Even when the iteration macros are generated by **caesar.adt**, the user can still provide alternative definitions for these macros (e.g., to enumerate only prime numbers below 1000). This can be done by inserting, in the "**.t**" file, the following directives:

```
#undef N2
#define N2 ...
#undef N3
#define N3(...) ...
```

For numeral sorts, a simpler way to restrict the iterations to the range 0...(*n*-1) is to define the macro **CAESAR_ADT_HASH_***N* to *n* (see below the section on SPECIAL SYMBOLS for details).

If the attribute "**iteratedby**" is absent, unique C identifiers are generated for *N2* and *N3*, with no other guarantee.

Note: the attribute "**iteratedby**" was introduced in April 2004; earlier versions of the CADP tools used another attribute "**enumeratedby**" that is now deprecated as it could not provide iteration for "complex" LOTOS sorts. See item #903 in file "**$CADP/HISTORY**" for details and comparison between old-style and new-style iterators.

- The attribute "**printedby** *N4*", if present, indicates that the abstract values of sort *S* can be displayed using a concrete C function (or macro-definition) named *N4*. This function takes two arguments, a POSIX stream (of type "**FILE** *") and a value of type *N*, and returns a result of type **void** (i.e., no result at all). This function prints the value to the stream as a human-readable character string, on a single line and using printable characters only (carriage-return or line-feed character are forbidden, and non-printable characters must be escaped).

  If the attribute "**printedby**" is absent, a unique C identifier is generated for *N4*, with no other guarantee.

  The definition of function *N4* should always take into account the case where the value to print is undefined (i.e., equal to a bit-zero pattern).

- The attribute "**list**", if present, signals that sort *S* is a list data structure and instructs **caesar.adt** to generate a printing function *N4* that displays the values of this sort as lists rather than algebraic terms, e.g., "{*x*, *y*, *y*}" rather than "CONS (*x*, CONS (*y*, CONS (*z*, NIL)))". See item #1475 in file "**$CADP/HISTORY**" for details.

- The attribute "**external**", if present, instructs **caesar.adt** not to generate C code for *S*, which has to be implemented manually. For this sort, the user must provide, in the "**.t**" file, the corresponding C type *N* that implements *S*, the comparison function *N1*, the iteration macros *N2* and *N3* (if needed), and the printing function *N4*.

  If a sort is declared to be external, the attributes "**implementedby**", "**comparedby**", "**iteratedby**" (if needed), and "**printedby**" should be present, so that the names *N*, ... *N4* are fixed and will not change even if the LOTOS specification is modified or if the CADP tools are upgraded to a new version.

  Other constraints on external sorts have been listed above in the section entitled CONSTRAINTS ON SORTS AND CONSTRUCTORS.

SPECIAL COMMENTS FOR OPERATIONS

A special comment attached to the declaration of a LOTOS operation *F* has the following syntax (where square brackets mean optional elements):

```
(*!
 [ implementedby N ]
 [ constructor ]
 [ external ]
*)
```

where *N* is a valid C identifier. Even if these elements are optional, their order cannot be modified. Spaces and blanks are allowed inside special comments. All letters in the words "**implementedby**", "**constructor**", "**external**" can be either in lower or upper case; it is advised to use the lower case only.

The meaning of special comments is the following:

- The attribute "**implementedby** *N*", if present, indicates that the abstract LOTOS operation *F* is implemented by a concrete C function or macro-definition named *N*. The parameters and result of *N* must be compatible with those of *F*.

  If *F* is a LOTOS constant (i.e., an operation with arity zero) and if *N* is a C macro-definition, then *N* must be defined to be followed by parentheses surrounding an empty list of arguments, so that any call to *N* should be written "*N()*" rather than "*N*".

  If the attribute "**implementedby**" is absent, the CADP tools will generate a unique C identifier for the implementation of *F*. Nothing else should be assumed about this identifier. In particular, it may change with future versions of the CADP tools.

  The option **-comments** of the CADP tools emits a warning for each unspecified "**implementedby**" attribute.

  The option **-map** of the CADP tools produces a file that gives the name correspondence between abstract LOTOS operations and concrete C functions.

- The attribute "**constructor**", if present, instructs **caesar.adt** that *F* is a constructor operation. Otherwise, **caesar.adt** will assume that *F* is a non-constructor, i.e., an operation whose meaning is defined by the algebraical equations contained in the LOTOS specification.

- The attribute "**external**", if present, instructs **caesar.adt** not to generate C code for *F*, which has to be implemented manually.

  If *F* is a non-constructor, the user must provide, in the "**.f**" file, the corresponding C function or macro-definition *N* that implements *F*.

  If *F* is a constructor, the user must provide, in the "**.t**" file, the corresponding C function or macro-definition *N* that implements *F*, a *tester* function *N0* taking one argument *x* of type *N* and returning a non-zero result if *x* matches at its top level the constructor *F*, and *n selector* functions *N1*, ..., *Nn* (where *n* is the arity of *F*) taking one argument *x* of type *N* that satisfies *N0* (*x*) != 0 and returning, respectively, the values *x1*, ..., *xn* such that *x* = *N* (*x1*, ..., *xn*).

  If an operation is declared to be external, the attribute "**implementedby**" should be present, so that the name *N* of the implementation is fixed and will not change even if the LOTOS specification is modified or if the CADP tools are upgraded to a new version.

  If the operation is a constructor, attributes should exist for fixing the names of the tester and selector functions, but such attributes are not implemented yet.

  Other constraints on external sorts have been listed above in the section entitled CONSTRAINTS ON NON-CONSTRUCTORS AND EQUATIONS.

SIDE EFFECTS IN EXTERNAL FUNCTIONS

As a general principle, any C function implementing a (constructor or non-constructor) LOTOS operation should not perform side effects because these are not present in the formal semantics of LOTOS abstract data types.

When the manually-written C code is to be used together with **caesar** (i.e., mixed with C code automatically generated by **caesar** for state-space exploration), the above principle holds quite strictly. In practice, side effects may be possible but only under very limited forms: allocating new memory cells, storing data values in a unique way using hash tables, or writing information to files (e.g., traces on log files). All other forms of side effects are prohibited. In particular, certain optimizations performed by **caesar** may become incorrect in presence of external functions with side effects.

When the manually-written C code is not to be used with **caesar**, the aforementioned prohibition can be relaxed, so as to obtain the same features as monads in functional languages. It becomes even possible to provide manually-written C functions that keep internal variables or modify previously allocated data structures (e.g., update certain fields of a linked list).

This is a risky practice that requires care and insight. An example of monads can be found by inspecting the *ACTION* type declared in "**$CADP/lib/X_ACTION.lib**" and implemented in "**$CADP/incl/X_ACTION.h**".

CALL-BY-NEED EVALUATION IN EXTERNAL FUNCTIONS

When implemented manually as C macro-definitions, certain LOTOS operations can avoid the call-by-value semantics enforced by **caesar.adt** (i.e., the actual arguments of a function are always evaluated before calling this function) and rely on call-by-need semantics instead.

A first example is given by the *and_then* and *or_else* operators declared in "**$CADP/lib/X_BOOLEAN.lib**". These operators are implemented in "**$CADP/incl/X_BOOLEAN.h**" as macro-definitions that expand to the C operators "**&&**" and "**||**". Depending on the value of their first argument, these operators may skip evaluating their second argument (performing so-called "short-circuits").

A second example is given by the *if_then* and *if_then_else* operators declared in "**$CADP/lib/X_ACTION.lib**". These operators are implemented in "**$CADP/incl/X_ACTION.h**" as macro-definitions that expand to the C ternary operator "(...**?**...**:**...)". The value of the first argument determines which one of the remaining arguments will be evaluated.

MEMORY CONCERNS WITH DATA TYPES

Saving memory is of utmost importance when using verification techniques based on state-space exploration. This section presents some effective ways to save memory when using LOTOS abstract data types with the CADP tools.

Let *C* be a constructor that may allocate memory when invoked (for instance, *C* can be the *cons* operator of linked lists). Any call to *C* in the patterns on the left-hand side of an equation will not allocate memory. However, any call to *C* in a premiss or on the right-hand side of an equation will allocate memory, so one should be careful about such calls if memory space needs to be optimized.

Dynamically-allocated data types are expensive. By default, **caesar.adt** tries to reduce their use by introducing as few pointer types as needed in the generated C code, only resorting to pointers where they are necessary to break dependencies between circular types, or where the user has explicitly asked for pointers (see the **CAESAR_ADT_HASH_***N* symbol below).

For a dynamically-allocated type that needs pointers (e.g., a list type), the user can request (still using the **CAESAR_ADT_HASH_***N* symbol) to store all the values of this type in a hash table, meaning that each value is represented by its index in the table. Different types have different hash tables, meaning that the sizes of tables and the numbers of bits for indexes can be tuned for each type independently. The user has to predict a maximal size for each table, i.e., an upper bound on the number of values that will be inserted. The advantages of this approach are twofold: (i) memory consumption is significantly reduced, as identical values are stored in memory only once; and (ii) the CPU overhead required for insertion and lookup in hash

tables is usually compensated by the gain in comparing values, because only indexes have to be compared (it is no longer necessary to perform "deep" structural comparison of algebraic terms). Because the "**caesar_table_1**" library of OPEN/CAESAR is used to handle these hash tables, any "**.h**" file generated by **caesar.adt** must be linked, if it uses hash tables, against the "**libcaesar.a**" library of OPEN/CAESAR and, possibly, the complement library "**libcaesar_plug.a**".

Finally, for dynamically-allocated types not stored in tables, it is also possible to activate a conservative garbage collector, using the "**CAESAR_ADT_GARBAGE_COLLECTION**" macro defined below.

SPECIAL SYMBOLS USED IN THE C CODE

A number of variables, functions, and macro-definitions may or must be used to exploit at its best the C code generated by **caesar.adt**. Here are the most important ones, sorted into different categories.

VERSION CHECKING

The four following symbols are used for checking versions:

CAESAR_ADT

This macro is defined in any "**.h**" file generated by **caesar.adt** and is equal to the version number of **caesar.adt** (e.g., 5.4) at the time the file was produced. This macro can be consulted in the "**.f**" and "**.t**" files written by the user.

CAESAR_ADT_EXPERT

In any "**.h**" file not generated by **caesar.adt** and containing manually-written C code to be used with **caesar**, the user should insert a macro-definition of the following form:

#define CAESAR_ADT_EXPERT x.y

where x.y is the version number of **caesar.adt** at the time the file was written. **caesar** will check this number and, should the conventions evolve in the future, use it to ensure backward compatibility or warn about deprecated contents. See items #622 and #1033 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_EXPERT_F

In any "**.f**" file, the user should insert a macro-definition of the following form:

#define CAESAR_ADT_EXPERT_F x.y

where x.y is the version number of **caesar.adt** at the time the file was written. **caesar** and **caesar.adt** will check this number and, should the conventions evolve in the future, use it to ensure backward compatibility or warn about deprecated contents. See items #622 and #1033 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_EXPERT_T

In any "**.t**" file, the user should insert a macro-definition of the following form:

#define CAESAR_ADT_EXPERT_T x.y

where x.y is the version number of **caesar.adt** at the time the file was written. **caesar** and **caesar.adt** will check this number and, should the conventions evolve in the future, use it to ensure backward compatibility or warn about deprecated contents. See items #622 and #1033 in file "**$CADP/HISTORY**" for details.

INTERFACING

The following symbol determines the usage of a "**.h**" file:

CAESAR_ADT_INTERFACE

The "**.h**" file generated by **caesar.adt** often contains more than mere interface declarations, as it also contains definitions of variables and functions. Thus, the "**.h**" file can be included only once, in one single "**.c**" file, otherwise double definitions will ensue at link-edit time. The **CAESAR_ADT_INTERFACE** macro-definition addresses this issue. If this macro is defined before including the "**.h**" file, i.e.:

```
#define CAESAR_ADT_INTERFACE
#include "filename.h"
```

only the interface declarations contained in "*filename*.**h**" will be included. See item #859 in file "**$CADP/HISTORY**" for details.

INITIALIZATION AND TERMINATION

The two following symbols perform initialization and termination:

CAESAR_ADT_INIT()

The "**.h**" file generated by **caesar.adt** defines a function named **CAESAR_ADT_INIT()** that must be invoked before using any other primitive contained in the "**.h**" file. See items #212, #1253, and #1914 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_TERM(...)

The "**.h**" file generated by **caesar.adt** defines a function named **CAESAR_ADT_TERM()** that can be (optionally) invoked by the user when the other primitives contained in the "**.h**" file will not be called any more. This function takes a single parameter of type "**FILE** *" and, if this parameter is not **NULL**, prints to this file the contents of the hash tables for which the user has provided a format by means of the symbols **CAESAR_ADT_FORMAT_***N* or **CAESAR_ADT_FORMAT** defined below. This function then deletes all the hash tables allocated in memory. See item #1250 in file "**$CADP/HISTORY**" for details.

TYPE TUNING

The following symbols enable the representation of types to be modified:

ADT_BITS_NAT

If defined to a natural value *n* before including the file "**$CADP/incl/X_NATURAL.h**" that implements the natural numbers defined in the "**X_NATURAL.lib**" library, this macro indicates that natural numbers are implemented on *n* bits only. See item #1584 in file "**$CADP/HISTORY**" for details.

ADT_BITS_INT

If defined to a natural value *n* before including the file "**$CADP/incl/X_INTEGER.h**" that implements the integer numbers defined in the "**X_INTEGER.lib**" library, this macro indicates that integer numbers are implemented on *n* bits only. See item #1584 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_BITS_*N*

If defined, this macro indicates that the C type *N* implementing some LOTOS sort *S* can be stored as a bit field in a C struct or union type. The value of the macro should have the following form ": *n*" (without the surrounding double quotes), where *n* is the number of bits needed for representing type *N*, knowing that *n* <= **sizeof** (*N*). For non-external sorts, this macro is automatically defined by **caesar.adt**, and, for external sorts, it can be manually defined by the user in the "**.t**" file; forgetting to define this macro when appropriate is harmless though less efficient, as more memory will be used than needed.

CAESAR_ADT_HASH_*N*

If defined by the user in the "**.t**" file as an integer value *n*, this macro specifies the way in which the C type *N* implementing a LOTOS sort *S* must be defined. If *S* is not external, then **caesar.adt** takes this macro into account to generate the C code for type *N*. If *S* is external, then **caesar.adt** takes this macro into account to implement or optimize the sorts that transitively depend on *S*.

Setting $n > 1$ means that the values of sort *S* are stored in a hash table that can contain at most *n* elements. Type *N* will be implemented as an index ranging between 0 and (*n*-1) that gives access to this table. **caesar.adt** checks that the value of *n* is not too large. *S* should be a tuple, an ordinary, an external, or a numeral sort (the latter case will be detailed below). If *S* is a singleton or an enumeration sort, a warning will be emitted and the definition of **CAESAR_ADT_HASH_***N* will be ignored. If the limit of *n* values is exceeded, the execution will stop with a **SIGTERM** signal unless the user has specified a different error handler using the **CAESAR_ADT_OVERFLOW_***N* macro defined below.

Setting $n < 0$ means that the values of sort *S* are stored in a hash table that can contain at most (2^-*n*) elements. This case is similar to the previous one, noticing that values of type *N* will be implemented on (-*n*) bits exactly. **caesar.adt** checks that the value of *n* is not too small.

Setting $n = 1$ means that the values of sort *S* should be implemented using pointers to structs or to unions (i.e., they should have a *boxed* representation). *S* should be a tuple, an ordinary, or a numeral sort (the latter case will be detailed below). If *S* is a singleton, an enumeration, or an external sort, a warning will be emitted and the definition of **CAESAR_ADT_HASH_***N* will be ignored.

Setting $n = 0$ means that the values of sort *S* should neither be implemented using pointers (i.e., they should have an *unboxed* representation) nor be stored in a hash table. *S* can be a singleton, an enumeration, a tuple, or an ordinary sort; in the two latter cases, an error is reported if there are cyclic dependencies that cannot be resolved by introducing pointers or table indexes because the user has forbidden to do so by requiring $n = 0$ for the mutually recursive types. If *S* is a numeral, an error message is also emitted as having $n = 0$ would be meaningless. If *S* is an external sort, taking $n = 0$ has no effect and is ignored silently.

As a consequence of the above, for singleton and enumeration sorts, the **CAESAR_ADT_HASH_***N* macro should either be undefined or have its value *n* set to zero.

In the case of numeral sorts, hash tables are never used, whatever which value is given to *n*. Setting $n > 0$ means that type *N* will represent the natural numbers ranging between 0 and (*n*-1). Setting $n < 0$ means that type *N* will represent the natural numbers coded on (-*n*) bits.

Finally, if the **CAESAR_ADT_HASH_***N* macro is undefined, **caesar.adt** automatically chooses the most appropriate implementation corresponding to either $n = 0$ or $n = 1$; **caesar.adt** will do its best efforts to avoid implementing *N* as a pointer type (i.e., choosing $n = 0$) unless this is necessary to break circular type dependencies. Hence, by default, no LOTOS sort will have its value stored in a hash table unless specifically requested by the user.

See items #623, #1250, #1251, #1255, #1332, #1435, #1494, and #1498 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_HASH_ADT_STRING

If defined to an integer value *n* before including the file "**$CADP/incl/X_STRING.h**" that implements the (variable-length) character strings defined in the "**X_STRING.lib**" library, this macro modifies the way strings are stored in memory. By default, strings are allocated dynamically using

malloc(), which can be memory-inefficient. If *n* > 1, strings are stored in a hash table with *n* entries at most. If *n* < 0, strings are stored in a hash table with 2^(-*n*) entries at most. See item #1495 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_SCALAR_*N*

If defined, this macro indicates that the constant 0 belongs to the C type *N* implementing some LOTOS sort *S*, so that variables of this type can be reset simply by assigning zero to them. In the C language, this corresponds to the usual notion of *scalar* type, which includes (signed and unsigned) integers and characters, reals, enums, and pointers, but excludes arrays, structs, and unions. For non-external sorts, this macro is automatically defined by **caesar.adt**, and, for external sorts, it can be manually defined by the user in the "**.t**" file; forgetting to define this macro when appropriate is usually harmless though slightly less efficient, as variables will be reset using a call to **memset()** rather than an assignment; however, the proper definition of this macro for external sorts is required when using the **-external** option of CAESAR. See item #493 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_UNCANONICAL_*N*

If defined, this macro indicates that the values of the C type *N* implementing some LOTOS sort *S* are not represented under a canonical form, meaning that two concrete C values stored in memory using two different bit strings may still denote the same abstract LOTOS value. This can occur if *N* is a pointer type (e.g., two pointers referring to two different memory cells having the same contents), but in other cases as well (e.g., a struct or union type with uninitialized padding bits between its fields, or a struct or union type containing pointer fields, and so on recursively). For non-external sorts, this macro is automatically defined by **caesar.adt**, and, for external sorts, it must be manually defined by the user in the "**.t**" file; forgetting to define this macro when appropriate is harmful as it leads to incorrect comparison and hashing on bit strings. For this reason, the values of non-canonical sorts should never be stored in hash tables. See items #623 and #1494 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_COLLISIONS_*N*

CAESAR_ADT_COLLISIONS

These two macros can be used to reduce the memory size of the hash table storing the values of the C type *N* that implements some LOTOS sort. By default, this table has as many hash entries as its maximal number of elements specified using the **CAESAR_ADT_HASH_***N* macro, meaning that the average length of collision lists is expected to be one if the table is entirely filled. The number of hash entries can be reduced, thus decreasing memory while potentially increasing access time. If the former macro, or else the latter macro is defined in the "**.t**" file as an integer value *n* >= 1, the number of hash entries will be the maximal number of elements divided by *n*, meaning that the average length of collision lists is expected to be *n* if the table is entirely filled. The latter macro applies to all types whose values are stored in hash tables, but it is overriden by the former macro as far as type *N* is concerned. See item #1250 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_CREATE_*N***()**

CAESAR_ADT_DELETE_*N***()**

CAESAR_ADT_SHOW_*N***(...)**

For each external LOTOS sort S implemented by a C type *N*, if the user has specified (by setting the macro **CAESAR_ADT_CREATE_***N* to a value different from 0 and 1) that the values of type *N* are canonical and stored in a hash table, the user has to implement this table manually and provide the corresponding implementation in the "**.t**" file; **caesar.adt** cannot generate code for this table because the implementation in C of the elements of this table is not known at the LOTOS level; to implement this table, the user may reuse the **caesar_table_1** library of OPEN/CAESAR.

Whatever which implementation is chosen for the table, the user must provide, in the "**.t**" file, the three above symbols, which must be implemented as macro-definitions, not as functions. The two former macros respectively allocate and deallocate the hash table; the latter macro prints the table contents to a file pointer passed as a parameter. These macros are invoked by **CAESAR_ADT_INIT()** and **CAESAR_ADT_TERM()**. Examples of such macros based upon the **caesar_table_1** library of OPEN/CAESAR can be found by examining the definitions of **CAESAR_ADT_CREATE_ADT_STRING()**, **CAESAR_ADT_DELETE_ADT_STRING()**, and **CAESAR_ADT_SHOW_ADT_STRING()** given in file "**$CADP/incl/X_STRING.h**". See item #1498 in file "**$CADP/HISTORY**" for details.

VALUE PRINTING

The following symbols modify the way terms can be printed:

CAESAR_ADT_INFIX

If defined, this macro indicates that the values containing constructor operations declared as infix in the LOTOS specification (i.e., "**_***F*_ **:** *S*, *S* **->** *S'*") should be printed in the infix form "x *F* y" rather than the prefix form "*F* (x, y)", the latter being the default. This macro is normally set by the options **-prefix** and **-infix** of **caesar.adt**, but the user can also define this macro before including the "**.h**" file generated by **caesar.adt**. See items #080 and #208 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_PRINT_OPEN_*N*

CAESAR_ADT_PRINT_CLOSE_*N*

These two macros determine how a constructor operation *F* implemented by a C function named *N* and returning a result of some record (or even singleton) sort is printed. By default, these macros are defined in the "**.h**" file generated by **caesar.adt** unless the user provides alternative definitions before including this file. For instance, values of type *N* can be printed as "[x1, x2, ..., xn]" rather than "*F* (x1, x2, ..., xn)". See item #1561 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_FORMAT_*N*

CAESAR_ADT_FORMAT

These two macros determine whether the hash table storing the values of the C type *N* that implements some LOTOS sort will be displayed in case the table overflows (because it cannot contain as many values as necessary) or when the **CAESAR_ADT_TERM()** function is executed. If none of these two macros is defined before including the "**.h**" file generated by **caesar.adt**, the table will not be displayed. If the former macro, or else the latter macro is defined in the "**.t**" file as an integer value *n*, the table for type *N* will be displayed under format *n*, following the conventions of the **CAESAR_PRINT_TABLE_1()** primitive of OPEN/CAESAR. The latter macro applies to all types whose values are stored in hash tables, but it is overriden by the former macro as far as type *N* is concerned. See item #1250 in file "**$CADP/HISTORY**" for details.

DYNAMIC MEMORY

The following symbols control dynamic memory allocation and reclaim:

CAESAR_ADT_ALLOC(...)

This macro allocates a memory cell to contain a value of a given type and assigns the address of this cell to a given variable. If the allocation fails, an error message is displayed and the execution is interrupted by a **SIGTERM** signal. By default, this macro is defined in the "**.h**" file generated by **caesar.adt** unless the user provides an alternative definition before including this file.

CAESAR_ADT_GARBAGE_COLLECTION

The "**.h**" file generated by **caesar.adt** checks whether a macro named **CAESAR_ADT_GARBAGE_COLLECTION** is defined. If so, the memory allocation primitive of the Boehm-Demers garbage collector will be used rather than the standard **malloc()** primitive; this will require linking with the "**$CADP/gc/bin.*/libgc.a**" library. This macro can be defined by the user when compiling the "**.h**" file generated by **caesar.adt**, but it is automatically defined by **caesar** when invoked with its "**-gc**" option. See item #653 in file "**$CADP/HISTORY**" for details.

ERROR HANDLING

The following symbols determine how erroneous situations are handled:

ADT_CHECK_NAT

If defined before including the file "**$CADP/incl/X_NATURAL.h**" that implements the natural numbers defined in the "**X_NATURAL.lib**" library, this macro activates systematic checks for overflows and underflows, which could otherwise remain undetected. See item #1584 in file "**$CADP/HISTORY**" for details.

ADT_CHECK_INT

If defined before including the file "**$CADP/incl/X_INTEGER.h**" that implements the natural numbers defined in the "**X_INTEGER.lib**" library, this macro activates systematic checks for overflows and underflows, which could otherwise remain undetected. See item #1584 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_ERROR(...)

This macro is invoked when a partially-defined non-constructor is called with actual parameters whose values are not matched by any equation; an error message is displayed and the execution is interrupted by a **SIGTERM** signal. By default, this macro is defined in the "**.h**" file generated by **caesar.adt** unless the user provides an alternative definition before including this file.

CAESAR_ADT_OVERFLOW_*N*(...)

If defined before including the "**.h**" file generated by **caesar.adt**, this macro provides an "overflow" function to be invoked when the hash table storing the values of the C type *N* that implements some LOTOS sort is full. If the user does not define this macro, **caesar.adt** generates a standard overflow function automatically. See item #1250 in file "**$CADP/HISTORY**" for details.

FUNCTION TRACING

The following symbols enable functions to be traced at run time:

CAESAR_ADT_TRACE_*N*

When invoked with its "**-trace**" option, **caesar.adt** will generate, for each LOTOS operation *F* implemented by a C function named *N*, additional C code that is executed when entering and exiting *N*. Compared to the usual function-tracing features offered by C debuggers, this code can print the values of actual parameters and returned results using the same notations as in the source LOTOS specification. The execution of this code can be enabled or disabled, for each LOTOS operation considered individually, using an integer variable named **CAESAR_ADT_TRACE_*N***, which is initialized to zero. Setting this variable to one (either at compile time by manually patching the "**.h**" file generated by **caesar.adt**, or at run time by assigning the variable directly from a debugger) allows the code to be executed when function *N* is called and returns. See item #180 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_ARGUMENT_TRACE(...)
This macro prints the actual parameters passed to functions under trace. By default, it is defined in the "**.h**" file generated by **caesar.adt** unless the user provides an alternative definition before including this file. See items #180 and #181 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_ENTRY_TRACE(...)
This macro prints the name and depth level of functions under trace when they are entered. By default, it is defined in the "**.h**" file generated by **caesar.adt** unless the user provides an alternative definition before including this file. See items #180 and #181 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_EXIT_TRACE(...)
This macro prints the name and depth level of functions under trace when they are exited. By default, it is defined in the "**.h**" file generated by **caesar.adt** unless the user provides an alternative definition before including this file. See items #180 and #181 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_RESULT_TRACE(...)
This macro prints the results returned by functions under trace. By default, it is defined in the "**.h**" file generated by **caesar.adt** unless the user provides an alternative definition before including this file. See items #180 and #181 in file "**$CADP/HISTORY**" for details.

DEPRECATED SYMBOLS

The following symbols are no longer supported:

CAESAR_ADT_NORMAL_FORM
See items #059, #082, and #235 in file "**$CADP/HISTORY**" for details.

CAESAR_ADT_POINTER_*N*
See items #235 and #623 in file "**$CADP/HISTORY**" for details.

CAESAR_INFIX_FORM_PRINTING
See items #080 and #208 in file "**$CADP/HISTORY**" for details.

BIBLIOGRAPHY

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems, vol. 14, num. 1, pages 25-59, January 1988.

[Gar89b] Hubert Garavel. Compilation et verification de programmes LOTOS. These de doctorat, Universite Joseph Fourier, Grenoble, November 1989. Available from http://cadp.inria.fr/publications/Garavel-89-b.html

[Gar89c] 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, Canada. North Holland, pages 147-162, December 1989. Available from http://cadp.inria.fr/publications/Garavel-89-c.html

[Gar13] Hubert Garavel et al. 25 Years of Compositionality Issues in CADP: An Overview. Lecture at the Workshop on the 25 Years of Combining Compositionality and Concurrency (WS25CCC), Koenigswinter, Germany, August 6-9, 2013. Available from ftp://ftp.inrialpes.fr/pub/vasy/presentations/Garavel-25CCC-13.pdf

[GS95a] Hubert Garavel and Mihaela Sighireanu. Defect Report Concerning ISO International Standard 8807 and Proposal for a Correct Flatenning of LOTOS Parametrized Types. Rapport SPECTRE, 95-11, VERIMAG, Grenoble, July 1995 http://cadp.inria.fr/publications/Garavel-Sighireanu-95-a.html

[GT93] Hubert Garavel and Philippe Turlier. CAESAR.ADT : un compilateur pour les types abstraits algebriques du langage LOTOS. In Rachida Dssouli and Gregor v. Bochmann, editors, Actes du Colloque Francophone pour l'Ingenierie des Protocoles (CFIP'93), Montreal, Canada, 1993. Available from http://cadp.inria.fr/publications/Garavel-Turlier-93.html

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

SEE ALSO

caesar(LOCAL), **caesar.adt**(LOCAL), **caesar.indent**(LOCAL), **lotos.open**(LOCAL), **fsp2lotos**(LOCAL), **lnt2lotos**(LOCAL)

Additional information is available from the CADP Web page located at http://cadp.inria.fr

Directives for installation are given in files **$CADP/INSTALLATION_***.*

Recent changes and improvements to this software are reported and commented in file **$CADP/HISTORY.**

BUGS

Please report bugs to cadp@inria.fr