Compilation and Verification of LOTOS Specifications

Hubert GARAVEL[*]
VERILOG Rhône-Alpes
Centre HITELLA
46 avenue Félix Viallet
38031 GRENOBLE cedex
FRANCE
e-mail: `hubert@imag.imag.fr`

Joseph SIFAKIS[†]
L.G.I.
I.M.A.G. Campus
BP 53X
38041 GRENOBLE cedex
FRANCE
e-mail: `sifakis@imag.imag.fr`

The ISO specification language LOTOS *is a Formal Description Technique for concurrent systems. This paper presents the main features of the* CÆSAR *system, intended for formal verification of* LOTOS *specifications by model-checking. This tool compiles a subset of* LOTOS *into extended Petri Nets, then into state graphs, which can be verified by using either temporal logics or automata equivalences. The design choices and the principles of functioning of* CÆSAR *are described and compared to those of other* LOTOS *tools. The paper also proposes ideas to deal with the state explosion problem arising in verification by model-checking.*

Introduction

System designers are often confronted with the *verification problem*, i.e., checking that a specification is correct with respect to some requirements, and the *implementation problem*, i.e., deriving an executable realization of a correct specification. In these areas, using automated tools to assist human work is highly desirable. Formal Description Techniques, such as LOTOS [ISO88] [BB88], are a prerequisite for the development of automatic methods.

This paper presents an approach for the efficient compilation and verification of LOTOS specifications. Although this approach was originally intended for formal verification by model-checking, it is in fact more general and can also be applied to interactive simulation, test generation and code generation purposes.

The technique described here has been fully implemented in the CÆSAR system. CÆSAR belongs to the CESAR family of verification tools for concurrent systems (QUASAR [FSS83] for a CSP-like language and XESAR [RRSV87] [GRRV89] for an ESTELLE dialect).

This paper is organized as follows. Section 1 discusses verification issues for LOTOS. Section 2 presents the general principles of the compilation technique used in CÆSAR. The architecture of CÆSAR and the translation algorithms are described in section 3, although it is not possible to give here all technical details (a formal presentation can be found in [Gar89a]). Finally, section 4 discusses the results and the limitations of the CÆSAR approach, and suggests solutions to cope with the state explosion problem inherent to the model-checking approach.

1 Verification of LOTOS specifications

A first step in computer-aided analysis of LOTOS specifications is debugging. Several simulators (i.e., interpreters) are available for LOTOS, for instance HIPPO [vE89] and UO-LOTOS [GL88]. These tools help designers discovering semantic errors by tracing and monitoring some possible execution sequences.

[*]This work has been partially supported by I.N.R.I.A.
[†]This work has been partially supported by ESPRIT Basic Research Action "Spec"

However, the simulation approach is not always sufficient, since it does not allow to find all design errors. Moreover, it is likely that LOTOS specifications will often be changed and updated, as most computer files and programs; the latest revision should always be verified, but the interactive aspects of simulation make this rather difficult and cumbersome.

For these reasons, it is necessary to go beyond mere simulation and develop more powerful techniques of automated analysis. Some simulators have been enhanced to deal with formal verification [SSC89]; it is not sure, however, that efficient verification can be carried out with the same techniques as those developed for simulation.

1.1 Model-checking versus proof

The natural approach, for verifying formally programs written in a high-level language like LOTOS, would be based on source level analysis. This general idea relies on the intuition that properties can be proven more easily at the highest abstraction level. As far as LOTOS is concerned, this idea is even more plausible, because LOTOS is a functional language, using clean concepts borrowed from process algebras and abstract data types, with a simple and well-defined semantics.

This should allow to develop formal methods taking advantage of the algebraic structure of LOTOS programs, considered as terms. For instance, there have been studies on the application of algebraic transformations and theorem proving techniques to LOTOS programs [AF88] [Boo89]. These approaches, despite their "mathematical flavor" have major flaws:

- the efficiency of existing theorem provers is not sufficient, because of the intrinsic complexity of the underlying decision algorithms and the inadequacy of current computer architectures for symbolic computations.
- proof techniques used for sequential programs cannot be efficiently transposed to parallel programs, due to the fact that many useful properties cannot be verified in a "compositional" way, since they rely on a knowledge of the system's global state.
- the exploitation of these techniques strongly depends on user's skills. Full automation is not possible since most steps of the proof require inventive intervention from the user.

For these reasons, proof techniques have only been illustrated in small examples of limited practical interest. An alternative solution is *model-checking* (also called *exhaustive simulation* or *reachability analysis*). In this approach LOTOS specifications are translated, when it is possible, into state/transition models which describe, possibly modulo some abstractions, the execution sequences. There are strong arguments in favor of model-checking:

- in the general case, the verification problem is not tractable, since properties expressing requirements to be verified may not be decidable. In fact, most significant properties of concurrent systems (such as termination, liveness, deadlock freedom, fairness, ...) are not decidable in general. However, if the system has only a finite set of states, every property is decidable. Under this assumption, model-checking appears more appropriate than theorem proving, since it is practically more efficient.
- verifying properties at source level for high-level languages does no seem to be realistic, since the set of language constructs is generally not minimal from a semantic point of view, due to the existence of non-primitive constructs provided for expression convenience. It is therefore preferable to carry out semantic analysis at some lower level. Most compilers, for instance, do not perform static analysis and code optimization at source-level, even for high-level languages; they generally operate on low-level forms: tuples, basic blocks, flow graphs, use-def chains, ...
- whatever verification method is chosen, one must evaluate properties on value expressions (such as deciding whether a boolean guard is true or not, for instance). Model-checking techniques avoid formal manipulation and comparison of algebraic terms, and the subsequent loss of efficiency, because they only operate on closed terms, in which variables are bound to known values.

  To get rid of symbolic variables in value expressions, *value flattening* is applied to first-order terms: for a given variable, each possible value must be considered. For instance, given the LOTOS behavior "**choice** $X : BOOL$", it is necessary to analyze both cases ($X = \mathit{false}$) and ($X = \mathit{true}$), such that further occurrences of variable X can be replaced by a single value, *false* or *true*.

These general principles are applied in CÆSAR, but not in a too restrictive way. For example, CÆSAR performs source level transformations during the expansion phase (see section 3.2) and allows the user to prevent value expansion, whenever it can be avoided (see section 4.3).

1.2 The graph model

When verification is performed by model-checking, the choice of the appropriate model relies on the dynamic semantics of the language. The control part of LOTOS is based on a process algebra. The terms of this algebra (*behavior expressions*) denote parallel computations. The rules of LOTOS operational semantics [ISO88] define a mapping from behavior expressions to *labelled transition systems*, i.e., automata, called here *graphs* since they can also be viewed as directed labelled graphs.

It is therefore quite natural to choose graphs as the low-level model on which verification is done. Fortunately, this well-known model supports a wide range of analysis techniques: graph algorithms, automata theory results, comparison by using equivalence relations, and evaluation of temporal logic formulas.

The graph model is described by:

- a finite set of *states*
- an *initial state*,
- a finite set of *edges*, each edge consisting of:
  - an *origin state*,
  - a *destination state*,
  - a *gate*, which can be either a visible gate (i.e., a gate parameter of the LOTOS specification, declared just after the "**specification**" keyword), or "**i**" (i.e., either a gate "**i**" in the LOTOS source text, or a internal gate declared by the "**hide**" operator),
  - an *offer*, which is a list of closed LOTOS value expressions (this list is always empty if the gate is "**i**").

2 Compilation of LOTOS specifications

To verify LOTOS specifications by model-checking, it is necessary to perform some kind of translation from behavior expressions to graphs. In fact, this translation is also needed whenever one wants to execute LOTOS specifications. The translation can be more or less explicit, automatical, and complete, according to the purposes sought:

- for model-checking *verification*, it is necessary to compute every possible evolution of the system and generate all the states reachable from the initial state. Finite termination requires detection of circuits: this is usually done by storing all states in memory and by comparing every new state to previously encountered ones.
- for *interactive simulation*, it is not necessary to keep track of all accessible states. It is often sufficient to store the current state or, if backtracking is allowed, the states which belong to the path followed from the initial state. If several evolutions are possible from the current state, the choice is either interactive (by the user) or automatical (according to some selection criteria).
- for *test generation*, only states on the current path should be kept, but it may be necessary to visit states more than once.
- for *sequential code generation*, the LOTOS specification must be translated into a sequential program which interacts with an environment. At run-time, only the current state is needed. In case of non-determinism, random or interactive choices are made to select an evolution among others.

To summarize, all these activities, despite their apparent differences, share a common task: translating a LOTOS specification into a graph, even if the whole graph is not built. They mainly differ in two points: the choice of states which are kept in memory, and the choice of successor states to be explored from the current state.

It is therefore likely that an efficient translation technique from behavior expressions into graphs can lead to significant progress in model-checkers, simulators, test generators, and code generators for LOTOS.

2.1 Compilation versus interpretation

The reference definition of LOTOS [ISO88] describes the semantics of behavior expressions by means of a derivation system, which defines how the labelled transition system corresponding to a given LOTOS specification can be generated. It provides a set of rewrite rules that specify the initial state, as well as every possible transition from a given state.

This dynamic semantics of LOTOS is operational: it can be interpreted by some machine to generate the graph corresponding to a LOTOS specification. Some LOTOS tools take advantage of this, by giving these rules to a term-rewriting engine, which can be either general [AF88] [Boo89] [NQS89], or specifically designed for LOTOS [MdM88]; some other tools rely on PROLOG [BC88] [GL88]. These approaches are straightforward and directly ensure the correctness of the translation, but they suffer from lack of efficiency:

- the determination of possible evolutions from a given state is slowed down by the fact that no efficient technique exists to find redexes in a LOTOS algebraic term.
- a state of the graph is naturally represented by a behavior expression, which is a term of the LOTOS process algebra. This generally requires large amounts of memory, even if only the current state is stored. Depending on the size of the LOTOS specification, it may not be possible to keep many states in memory.
- to detect circuits in the graph, in the case of model-checking, it is necessary to compare the current state to the previous ones. Determining whether two syntactically different behavior expressions denote the same state is undecidable; therefore, any comparison criterion can be only a sufficient condition. Simulators usually rely on syntactical identity [GL88], which is probably not strong enough. It seems not easy, anyway, to give a comparison criterion, both precise and practically efficient, which operates at the level of behavior expressions.

The performances of this approach may be found sufficient for interactive simulation, but not for verification, which has much stronger requirements. For this reason, CÆSAR does not use the reference semantics of LOTOS as a basis for direct implementation. It does not use rewrite techniques either. It is based on a completely different translation scheme, which provides a common and efficient algorithm for simulation, verification, test and code generation purposes.

For achieving significant performance gains, the basic idea is to discover computations performed many times that could be done only once. This naturally leads to replacing the interpretation scheme by a compilation scheme: graph generation should be divided into compile-time and run-time phases. Whenever possible, computations should be performed statically (at compile-time), and not dynamically (at run-time). This means that computations executed several times at run-time should be shifted to compile-time and would have their results saved in memory to be available at run-time.

These differences appear clearly, for instance, in the way rendez-vous are handled. The aforementioned LOTOS tools follow the interpretation scheme, having only a run-time phase relying on rewriting techniques: at each step, it is necessary to determine possible rendez-vous, although these computations may have already been done at previous steps. On the opposite, CÆSAR is based on a compilation scheme: a significant part of rendez-vous computations is done only once, at compile-time.

2.2 The network model

According to the compilation principle, LOTOS specifications should not be directly translated into graphs; stepwise transformations should be generated instead. It is therefore necessary to design an intermediate-level semantic model between LOTOS and the graph model in the translation process. Translation from LOTOS to this model corresponds to compile-time, whereas translation from this model to the graph is run-time. This model should satisfy the following properties:

- first of all, it should allow to store information determined at compile-time (pre-computed rendez-vous, for example).
- it should be expressive enough to represent all behaviors that can be described in LOTOS, or at least a large subset of them, sufficient in practice.
- it should provide a compact representation of control and data flow, unlike the graph model whose complexity may be very high, since all possible execution sequences are represented. This implies in particular that the expansion theorem (i.e., LOTOS algebraic rules which express parallel composition in terms of sequential composition and non-deterministic choice), as well as value flattening (described above), must be applied only at run-time, but not when building the intermediate model.

A brief examination of existing models and languages for concurrency, with respect to these criteria, leads to the following conclusions:

- formalisms making use of shared memory (such as semaphores, monitors, critical sections, ...) seem to be inappropriate for LOTOS.
- extended communicating automata (also called *abstract machines*) are not powerful enough to meet both the second and the third of the above criteria, about the expression of parallel composition. They are well-adapted to languages such as CSP or ESTELLE, which describe the parallel execution of sequential processes, but not to LOTOS where any combination of sequential operators (i.e., sequential composition, non-deterministic choice, ...) and parallel operators is allowed. Problems arise when communicating automata are used as an intermediate model for LOTOS: either behaviors like "B_1 `[]` `(`B_2 `|||` B_3`)`" cannot be compiled [Dub89], or the expansion theorem must be used in order to compute the product automaton corresponding to "B_2 `|||` B_3" in which case the complexity of the intermediate model can grow exponentially [NQS89].
- using Petri Nets solves this problem, since they allow to combine sequentiality, parallelism and non-determinism in a compositional way. A Petri Net is generally much smaller than the corresponding graph, since it allows to avoid the interleaving expansion. Moreover it appears that LOTOS rendez-vous can be directly expressed with Petri Nets synchronization features. Several algorithms have been proposed for translating the control part of LOTOS into standard Petri Nets [ML88] [BvB90], but they do not take variables and value expressions into account. A survey of the relationships between process algebras, automata and Petri Nets can be found in [Tau90].
- of course, standard Petri Nets are not powerful enough for LOTOS, since they only describe control aspects. To handle values, it is necessary to extend the Petri Net model by adding a data part, for defining and manipulating variables. The intermediate model used in CÆSAR has global variables, and its transitions are labelled by conditions and actions on these variables.

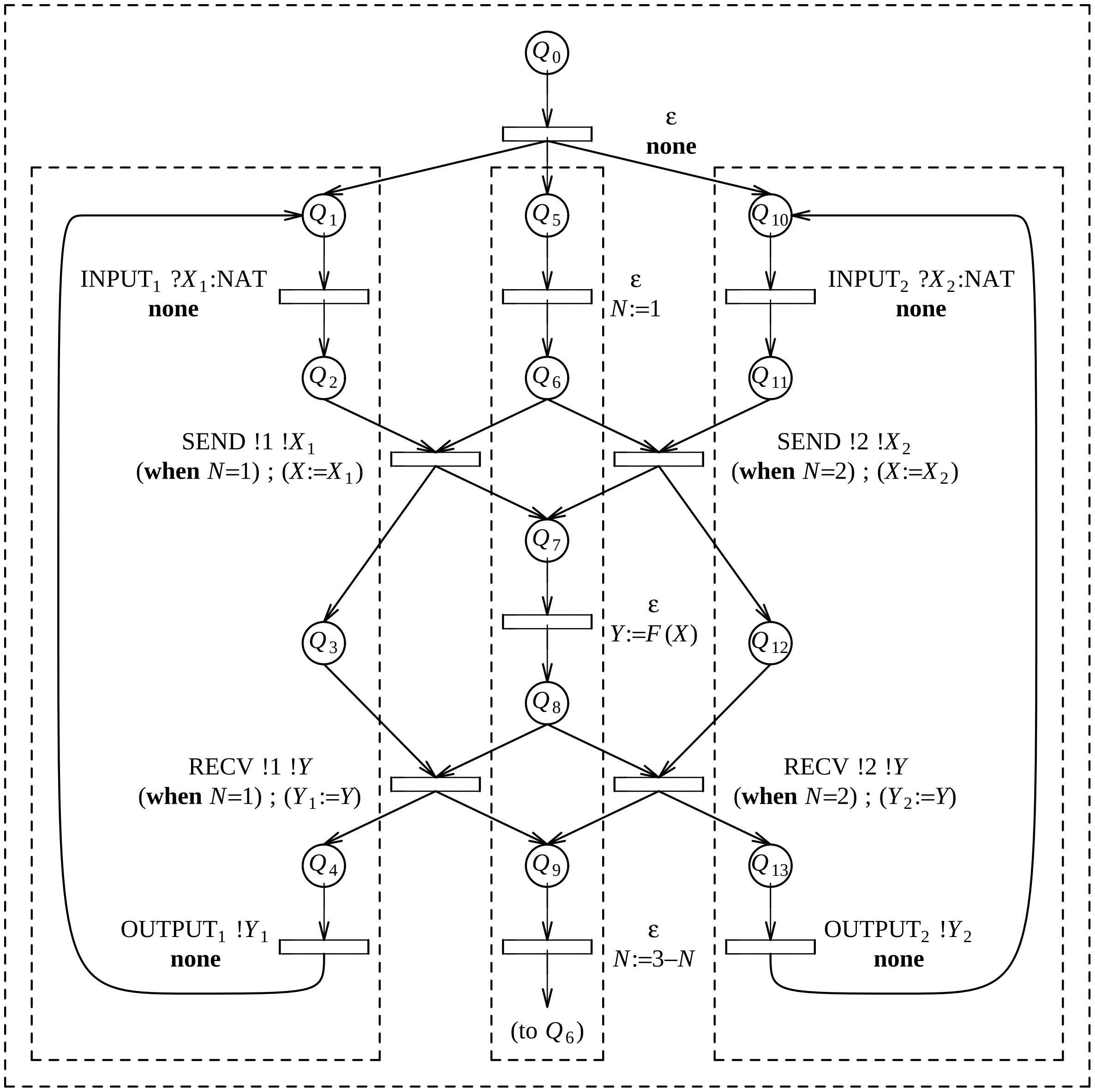
This intermediate model, called *network*, is described by:

- a finite set of *variables*,
- a finite set of Petri Net *places*,
- an *initial place*,
- a finite set of Petri Net *transitions*, each transition consisting of:
  - a set of *input places*,
  - a set of *output places*,
  - a *gate*, which can be either a visible LOTOS gate, or "τ", or "ε",
  - an *offer*, which is a list whose items have either the form "`!`V", where V is a value expression, or the form "`?`X`:`S", where X is a variable and S a sort (this list is always empty if the gate is "τ" or "ε"),

- an *action*, which is any sequential or collateral composition of *primitive actions*. A primitive action can be:
    - either the null action "**none**",
    - or a (vectorial) *assignment* "$X_0, ...X_n := V_0, ...V_n$" of value expressions $V_0, ...V_n$ to variables $X_0, ...X_n$ respectively,
    - or a *condition* (also called *guard*) "**when** V" where V is a boolean value expression,
    - or an *iteration* "**for** X_0 **among** S_0" of variable X_0 over the domain of sort S_0.

- a set of *units*, which determines a partition of the set of places. Intuitively each unit is a subset of places which represents a sequential behavior. A unit can also be hierarchically refined into sub-units to express that the corresponding behavior is composed of several concurrent sub-behaviors.

For instance, the following network $\mathcal{N}$ contains three sequential components running in parallel: a server computes a function $Y = F(X)$ alternatively for two clients, which send computations to the server and receive results. Places and transitions are represented by circles and rectangles, according to Petri Net graphical conventions; dashed boxes are used to represent units.

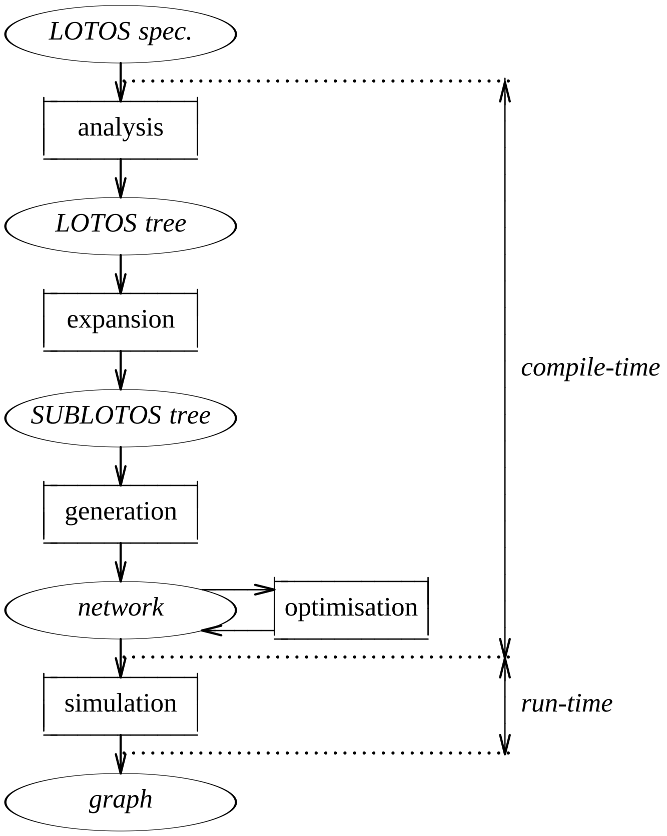


The rules for translating a LOTOS specification into a network are given in section 3.3. Those for translating a network into a graph are presented in section 3.5.

The network model is characterized by a static control structure: it describes a fixed set of tasks, with a fixed set of variables, communicating via a fixed set of gates. It is obvious that some dynamic features (e.g., dynamic creation and destruction of tasks, variables, and gates) cannot be expressed in this framework. It is therefore not possible to translate any Lotos specification into this model. The following syntactic restrictions, named *static control constraints*, must be satisfied: it is not allowed that a process recursively instantiates itself (even transitively), either on the left- or right-hand side of an operator "`|[...]|`", or in an operator "**par**", or on the left-hand side of an operator "`>>`" or "`[>`".

3 Architecture of the CÆSAR system

The Cæsar system translates Lotos specifications to networks and graphs by using successive steps. Its functional architecture can be described as follows:



3.1 The analysis phase

The front-end part of the tool performs syntactic analysis, by using the compiler-generator SYNTAX[1] and full static semantic analysis. It conforms, as much as possible, to the definition of Lotos given in the ISO Draft International Standard. As a result of this analysis phase, an abstract syntax tree is built.

3.2 The expansion phase

The translation from Lotos to the network model is divided into two successive phases, named *expansion* and *generation*, communicating via an intermediate form, called SubLotos, which can be seen as a subset of Lotos.

The expansion phase translates Lotos behavior expressions into SubLotos ones in a bottom-up way, according to syntax-directed rules defined by structural induction on Lotos behavior expressions:

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

- all occurrences of LOTOS operators "**par**" and "**choice**" on gates are replaced by equivalent expressions in terms of "`[]`" and "`|[...]|`", respectively.

- all occurrences of LOTOS operators "`|||`", "**exit**", "`>>`...**accept**" are replaced by equivalent forms making use of parallel composition, hiding, and rendez-vous, where the role of the termination gate "δ" is explicited.

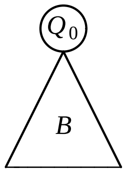
- non-recursive LOTOS processes are developed "in-line", i.e., process instantiations are replaced by process definitions, with appropriate substitution of actual gate parameters to formal ones. This transformation is correct with respect to LOTOS semantics, except if several actual gate parameters are identical, in which case CÆSAR issues a warning diagnostic; fortunately, this situation seldom occurs in practical examples.

- recursive LOTOS processes are also developed "in-line", until actual gate parameters are found to be identical to formal ones. This transformation always terminates if static control constraints are assumed.

Expansion ensures that all SUBLOTOS gates are "constants", i.e., SUBLOTOS dynamic semantics is free from gate relabelling notion.

3.3 The generation phase

The generation phase translates SUBLOTOS behavior expressions into corresponding network fragments by bottom-up synthesis. The translation rules are defined by structural induction on SUBLOTOS behavior expressions. Given a SUBLOTOS behavior expression B and a place Q_0, these rules allow to construct the network corresponding to B with Q_0 for initial place. This network is graphically represented as follows:



The network corresponding to a behavior B is recursively built from the sub-networks corresponding to the sub-behaviors contained in B:

- the generation algorithm for sequential operators "**stop**", "`;`", and "`[]`" is similar to non-deterministic automaton construction for regular expressions [ASU86, p. 121–125].
  - the network corresponding to "**stop**" has only one place, Q_0, and no transition (see figure 1).
  - the network corresponding to "$G\ O_1, ...O_n$ `[`V_0`]` `;` B_0" is obtained by creating a new place Q_1, generating the network corresponding to B_0 with Q_1 for initial place, and creating a transition from Q_0 to Q_1, whose gate is G, whose offer is $O_1, ...O_n$ and whose action is "**when** V_0" (see figure 2). The translation is slightly more complex when the operator "**;**" comes from the expansion of an operator "`>>`" or "**exit**".
  - the network corresponding to "B_1 `[]` B_2" is obtained by generating both networks corresponding to B_1 and B_2 with Q_0 for initial place (see figure 3).

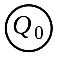


figure 1

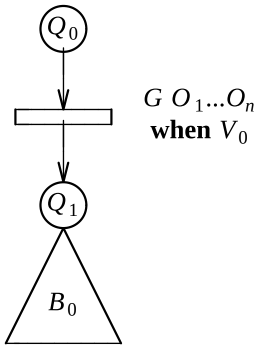


figure 2

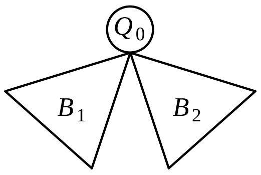


figure 3

- operators "`->`", "**let**", and "**choice**" on values, are translated into ε-transitions whose actions are, respectively, conditions, assignments, and iterations.
  - the network corresponding to "`[`V_0`] ->` B_0" is obtained by creating a new place Q_1, generating the network corresponding to B_0 with Q_1 for initial place, and creating an ε-transition from Q_0 to Q_1 whose action is "**when** V_0" (see figure 4).
  - the network corresponding to "**let** X_0`:`S_0`=`V_0 **in** B_0" is the same as the former, but the action of the ε-transition is "X_0`:=`V_0" (see figure 5). If the "**let**" operator defines several variables, the action is the collateral composition of the corresponding assignments.
  - the network corresponding to "**choice** X_0`:`S_0 `[]` B_0" is the same as the former, but the action of the ε-transition is "**for** X_0 **among** V_0" (see figure 6). If the "**choice**" operator defines several variables, the action is the collateral composition of the corresponding iterations.

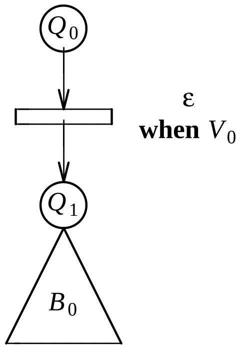


figure 4

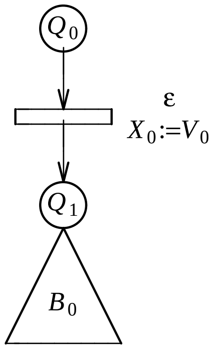


figure 5

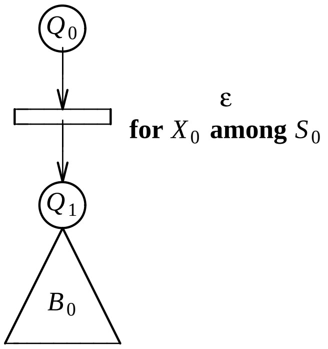
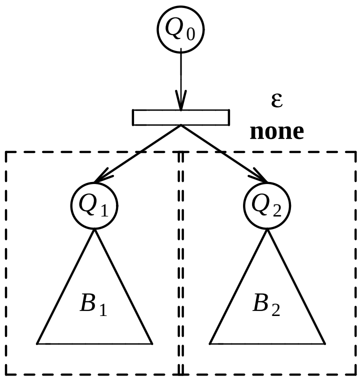


figure 6

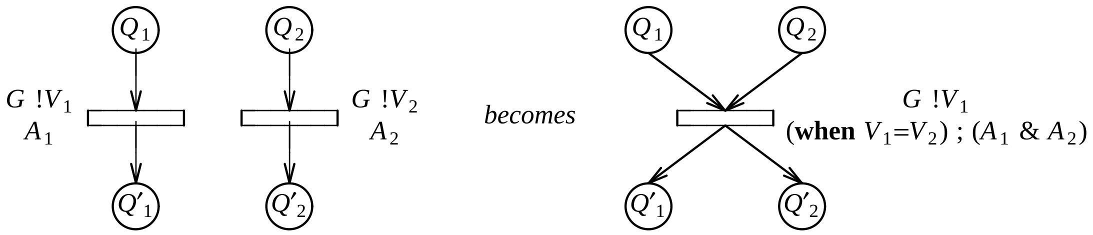
- translation of parallel operators is performed in two steps. First, concurrency is expressed by creating a *fork* transition, i.e., an ε-transition which starts several concurrent behaviors. Then synchronization and communication are expressed by *merging* all transitions which may engage in rendez-vous.

  The network corresponding to "B_1`|[`$G_0, ...G_n$`]|`B_2" is obtained by creating two places Q_1 and Q_2, generating the networks corresponding to B_1 and B_2 with Q_1 and Q_2 for initial places, and creating a fork ε-transition with action "**none**" from Q_0 to Q_1 and Q_2. Two units are also created to encapsulate the places of the networks corresponding to B_1 (including Q_1) and B_2 (including Q_2) respectively; this implies that the decomposition of a network into units is not determined by process definitions, but according to parallel composition operators.

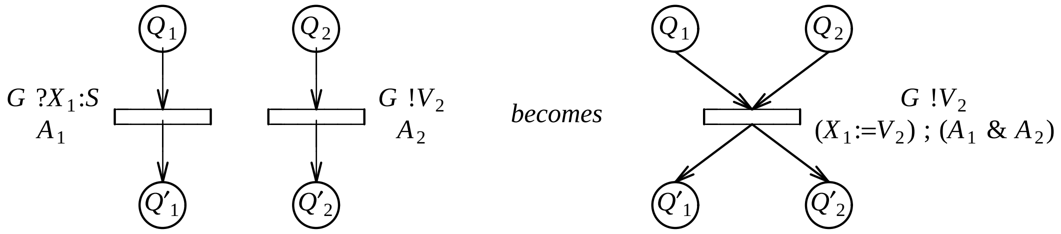


Two transitions T_1 and T_2, respectively contained in the networks corresponding to B_1 and B_2, must be merged if they have the same gate G belonging to $\{G_0, ...G_n\}$ and offers of the same sort. Merging transitions T_1 and T_2 is done by replacing them by a single transition T from the input places of T_1 and T_2 to the output places of T_1 and T_2, whose gate is G. The offer and action of T are defined as follows, depending on the respective offers O_1 and O_2 and actions A_1 and A_2 of T_1 and T_2:

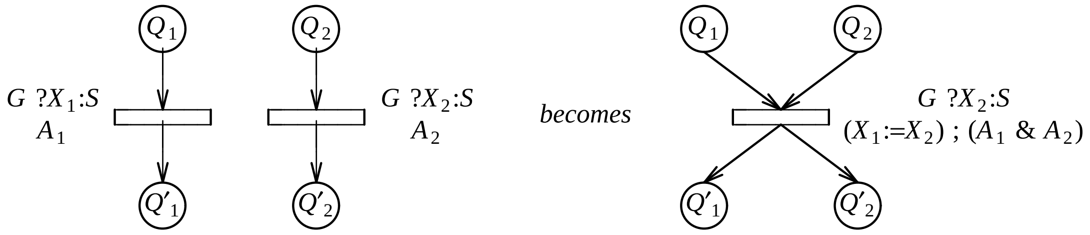
- for *value matching*, i.e., when O_1 has the form "$!V_1$" and O_2 has the form "$!V_2$", the offer of T is O_1 and the action of T is the condition "**when** V_1=V_2" followed by the collateral composition of A_1 and A_2.



- for *value passing*, i.e., when O_1 has the form "$?X_1:S$" and O_2 has the form "$!V_2$", the offer of T is O_2 and the action of T is the assignment "$X_1:=V_1$" followed by the collateral composition of A_1 and A_2. A symmetrical scheme is applied when O_1 has the form "$!V_1$" and O_2 has the form "$?X_2:S$".



- for *value generation*, i.e., when O_1 has the form "$?X_1:S$" and O_2 has the form "$?X_2:S$", the offer of T is O_2 and the action of T is the assignment "$X_1:=X_2$" followed by the collateral composition of A_1 and A_2.



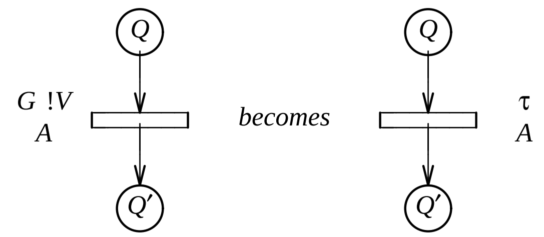
This scheme is general enough to handle all features of LOTOS rendez-vous. It can be easily generalized to the case where T_1 and T_2 have zero or more than one offers. The n-ary rendez-vous is naturally handled by repeatedly applying this scheme to each parallel operator.

The translation is somewhat different if the parallel operator comes from the expansion of an operator "`>>`". Although sequential composition appears in LOTOS as a particular case of parallel composition, it is not necessary, in such case, to create a fork transition, nor units. Merging transitions with the termination gate "δ" is sufficient to express sequentiality.

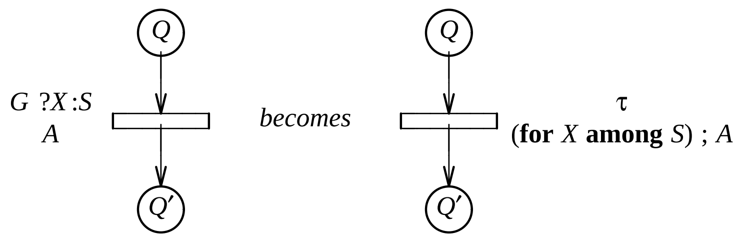
- translation of the hiding operator does create neither places nor transitions, but it modifies the gates, the offers, and the actions of the transitions to be hidden, which prevents these transitions to be merged at some later stage.

  The network corresponding to "**hide** $G_0, ...G_n$ **in** B_0" is obtained by generating the network corresponding to B_0 with Q_0 for initial place. Then, for each transition T with a gate G belonging to $\{G_0, ...G_n\}$, with an offer O and an action A, the following transformation is applied:

  – if O has the form "`!`V", the gate of T is replaced by τ, its offer is removed and its action remains unchanged.



  – if O has the form "`?`X`:`S" the gate of T is replaced by τ, its offer is removed and its action is replaced by the iteration "**for** X **among** S" followed by A. This expresses flattening of all possible input values for X.



- translation of the disabling operator "`[>`" is done by creating ε-transitions from the places of the disrupted behavior to the initial place of the disrupting behavior. The scheme becomes more complex when the disrupted behavior contains concurrent sub-behaviors, but it remains feasible under static control assumptions.

- translation of process instantiation depends on the fact that the instantiation is recursive or not. If not, a new place Q_1 is created and the network corresponding to the behavior of the instantiated process is generated with Q_1 for initial place. Otherwise, this network has already been generated and let also Q_1 be its initial place. In both cases, an ε-transition from Q_0 to Q_1 is created, whose action is a vectorial assignment of the actual value parameters to formal ones.

3.4 The optimization phase

Networks generated from SUBLOTOS behavior expressions are generally not optimal (i.e., they do not have a minimal number of places, transitions, and variables), because they are built in a systematical and compositional way. The optimization phase solves this problem by applying a list of optimizing transformations on networks. These transformations take into account both control and data flow aspects:

- as for control, local Petri Net optimizations are applied in order to reduce the number of places, transitions, and units. Special efforts are made to remove as much ε-transitions as possible, which greatly improves speed at run-time.

- as for data, global data-flow analysis techniques are used to perform standard optimizations, such as removing unused variables, eliminating useless assignments, folding constants, and evaluating constant guards statically. CÆSAR can also find sufficient conditions proving that two variables are equal, or that the value of a variable is constant, which reduces the number of variables.

All these optimizations are practically effective, due to the fact that the network model is simple enough to allow efficient and precise static analysis. This is even more true in the case of networks generated by CÆSAR, which have interesting properties, as they are obtained from descriptions written in a high-level functional language.

3.5 The simulation phase

The run-time phase, called *simulation* in the CÆSAR terminology, generates the graph corresponding to a network, by performing reachability analysis. This is done by building a C program, called *simulator*, then by compiling and running it. The simulator mainly consists of three components, which are respectively in charge of the following tasks:

- determination of the "next state" relation, i.e., computation for a given state of the outgoing edges and successor states (according to the operational semantics of the network defined below),
- management of a fast-access extensible hash table, in which all encountered states are searched and inserted,
- breadth-first exploration and construction of the graph, starting from the initial state. As the execution progresses, the states and edges of the graph are written to a file.

During expansion and generation, value expressions are handled symbolically. At run-time, however, evaluation of expressions cannot be delayed any more. It is not done by using rewriting techniques, to avoid the subsequent overhead; instead, the user has to provide appropriate C types and functions to implement LOTOS sorts and operations. This concrete implementation of abstract data types can be either written by hand, or automatically generated by an auxiliary tool, CÆSAR.ADT [Gar89b]. Correspondence between LOTOS and C identifiers is expressed by inserting special comments into the LOTOS source text.

The execution rules defining the operational semantics of the network model can be summarized as follows:

- each state of the graph denotes a global configuration of the network. It consists of a pair $\langle M, C\rangle$ where M is a *marking* and C a *context*:
  - a *marking* represents the control part of a state. According to standard Petri Net semantics, it is the subset of places that are currently marked with a token. For instance, $\{Q_0\}$, $\{Q_1, Q_5, Q_{10}\}$, and $\{Q_2, Q_5, Q_{10}\}$ are markings for network $\mathcal{N}$. In the CÆSAR network model, there can be several tokens simultaneously, but each place may contain at most one token (due to static control constraints; otherwise the number of tokens per place could be unbounded, and each token should carry context information).
  - a *context* represents the data part of a state. It contains the current values of each variable, i.e., the global memory state of the network. Formally, a context is a partial application which maps each variable to its value, or to the undefined value "$\perp$" if the variable is not initialized yet. For instance, $\{N \leadsto \perp, X \leadsto \perp, X_1 \leadsto \perp, X_2 \leadsto \perp, Y \leadsto \perp, Y_1 \leadsto \perp, Y_2 \leadsto \perp\}$ and $\{N \leadsto 1, X \leadsto \perp, X_1 \leadsto 0, X_2 \leadsto \perp, Y \leadsto \perp, Y_1 \leadsto \perp, Y_2 \leadsto \perp\}$ are contexts for network $\mathcal{N}$; as a shorthand, they will be noted $\{\}$ and $\{N \leadsto 1, X_1 \leadsto 0\}$ by omitting uninitialized variables. In the CÆSAR network model, variables have a global scope and can be assigned several times. These imperative features aim at efficiency. They contrast with the the functional characteristics of LOTOS, where each variable is local and bound to a single value. However, since networks are generated from LOTOS descriptions, a variable can never be used before it is assigned.
- the initial state of the graph is defined to be the pair $\langle M_0, C_0\rangle$, where M_0 is the marking containing only the initial place of the network, and C_0 the context with no variable initialized.

- deciding whether two states are identical is done by comparing their respective markings and contexts.
- rules to determine whether a transition T can be *fired* from the current state $\langle M, C\rangle$ and to compute the resulting state(s) $\langle M', C'\rangle$ take into account both marking and context information:
  - with respect to markings, standard Petri Nets evolution laws apply: T can be fired only if all its input places are in M; then M' will be defined as M, from which all input places of T have been removed, and to which all output places of T have been added. In network $\mathcal{N}$, for example, a transition can be fired from marking $\{Q_0\}$, leading to marking $\{Q_1, Q_5, Q_{10}\}$.
  - with respect to contexts, firing rules depend on the action attached to the transition: T can be fired only if each condition of its action evaluates to *true*. If so, C' is the resulting context obtained after the execution of its action in C. Assignments and iterations have their intuitive meaning: they allow to modify the value of variables, either by assigning them a single value, or by enumerating all values in the domain of a sort (in which case several contexts C' must be considered: this is value flattening). For example, applying action "`(`**when** N`=1) ; (`X`:=`X_1`)`" to context $\{N \rightsquigarrow 1, X_1 \rightsquigarrow 0\}$ leads to context $\{N \rightsquigarrow 1, X \rightsquigarrow 0, X_1 \rightsquigarrow 0\}$, whereas applying action "`(`**when** N`=2) ; (`X`:=`X_2`)`" does not lead to any context, since the guard is false.
- there is a direct relationship between network transitions and graph edges:
  - firing a transition with a visible gate creates a corresponding edge labelled by the gate and the offer of the transition.
  - similarly, firing a transition with a "τ" gate creates a corresponding edge labelled by "**i**".
  - on the contrary, firing a transition with an "ε" gate does not create an edge in the graph. The ε-transitions of the network model have no edge counterpart in the graph model. They have been introduced to allow compositional construction of networks and behave like ε-transitions in the theory of non-deterministic automata. As for automata, their semantics is based on an ε-closure algorithm [ASU86, p. 118–120]. However, the standard ε-closure algorithm is not sufficient for LOTOS, because automata describe sequential behaviors whereas networks represent concurrent ones. More precisely, the ε-closure only preserves language equivalence (i.e., trace equivalence) but not strong equivalence [Mil80]. For this reason, an auxiliary rule is added to the standard closure algorithm, which aims at preserving the atomicity of ε-sequences.

Simulation always terminates in finite time: either normally, or because there is not enough memory to store all states of the graph. In the latter case, CÆSAR displays a warning message and stops, but obviously cannot decide if the translation failed because the graph is infinite, or finite but too large (this question is equivalent to the halting problem).

From this operational semantics, one can understand why generating an intermediate-level model presents several advantages over the usual interpretative approach:

- states can be represented in a more efficient way than with the usual interpretative approach: storing a pair $\langle$marking, context$\rangle$ takes much less memory than storing the corresponding behavior expression.
- searching, from a given state, which transitions can be fired is much faster than determining which rewrite rules can be applied. This is even more true because a significant amount of computations, especially the determination of rendez-vous possibilities, has been done at compile-time.
- networks generated from LOTOS specifications are structured by units, which define a hierarchy of communicating sequential processes. It is possible to take advantage of units for:
  - compressing marking representation, since two places in the same unit cannot have a token simultaneously;
  - displaying graphically the architecture of the LOTOS specification, in terms of boxes (representing units) connected by communication channels (representing rendez-vous transitions between units);

– producing parallel code, where each unit is implemented by a distributed task, possibly with concurrent sub-tasks. The network model seems to provide an interesting base for parallel code generation, but this problem has not been tackled yet.

4 Discussion

4.1 Results

The algorithm for translating Lotos specifications into graphs constitutes an original implementation-oriented semantics for Lotos. The practical interest of replacing the usual interpretation scheme by a compilation scheme is clearly illustrated by the performance measurements of Cæsar.

The current version of the tool is made up of approximately 25 000 lines of C code. It allows to build large graphs (the largest graph generated on a SUN 4 workstation with 8 megabytes of main memory has 800 000 states and 3 500 000 edges) within reasonable time (the average speed ranges from 40 to 500 states per second, depending on the complexity of the specifications).

These performances appear to be satisfactory when compared to those of another model-checker for Lotos [BC88], and also to simulators using the interpretation scheme. Significant gains are still possible: it is likely, for instance, that extending the optimization phase and computing ε-closures at compile-time instead of run-time would sometimes lead to major improvements in speed.

The graphs generated by Cæsar can be verified by using several techniques. A first approach consists in reducing a protocol graph, or comparing a protocol graph to a service graph, modulo various equivalences relations defined for automata or process algebra, e.g., strong equivalence [Mil80], branching equivalence [GV90], observational equivalence [Mil80], testing equivalence [NH84], trace equivalence, ... A second approach makes use of temporal logics: requirements to be verified are expressed as formulas which are evaluated on the graph model [RRSV87] [Gar89a].

Cæsar does not embody verification algorithms: it only translates Lotos specifications into graphs. However, it smoothly interfaces with various existing verification systems, most of which performing graph reduction, namely Aldébaran (LGI-IMAG), Auto (INRIA), Mec (Univ. of Bordeaux), Pipn (LAAS/VERILOG), Scan (ADI/BULL) and Squiggles (Univ. of Pisa). Connection to Xesar (LGI-IMAG) is also possible; formulas of the temporal logic LTAC [Rod88] can therefore be evaluated on the graphs generated by Cæsar.

Cæsar could be easily extended to deal with simulation, test generation and sequential code generation. This would only require to adjust the simulation phase not to generate and store all states of the graph; all other phases of Cæsar would remain unchanged. The performances could even be better than for verification, since the overhead of state table management would be avoided.

4.2 Limitations

Unfortunately, the proposed technique does not apply to all Lotos specifications. There are two different restrictions:

- due to the verification technique by model-checking, Lotos specifications must translate into finite graphs. Such constraint is reasonable, because a Lotos specification leading to an infinite graph often describes, in fact, a set of finite implementations. Practically, the user has to put upper bounds on some parameters of the system, e.g., the number of concurrent processes, the number of values in the domain of a sort, ...
- due to the compilation technique used in Cæsar, Lotos specifications must satisfy static control constraints, which are also assumed by many other Lotos tools. These constraints are acceptable, since they do not prevent useful Lotos specification styles, such as implementation-oriented or process-oriented styles. They only forbid the use of control structures to describe infinite or unbounded data structures (e.g., queues, stacks, ...); for this purpose, abstract data types should be used instead [Led87].

There is a tight relationship between both restrictions:

- static control constraints are a sufficient condition for generating finite graphs, when each sort S, to which value flattening applies, has a finite domain. The domain of S must be finite only if exhaustive enumeration of all values of sort S is necessary, i.e., if S occurs either in a "**choice** $X : S$" clause, or in a "G ?$X : S$" clause that does not match a "G !V" clause, or in a "**exit** (**any** S)" clause that does not match a "**exit** (V)" clause, where V is any value expression. Practically, even if the domain of a sort is theoretically infinite, only a subset of it, chosen by the user, will be enumerated.

- conversely, LOTOS specifications that do not satisfy static control constraints generally lead to infinite graphs and therefore cannot be verified. It seems that there is a single case of practical interest where the non-respect of static control restrictions still generates finite graphs: using recursion in parallel composition to start a finite number of concurrent processes. In such case, the user has to write by hand as many process instantiations as necessary, which is tedious but feasible.

There seems to be an "efficiency versus expressiveness" tradeoff for LOTOS: restriction to a well-chosen subset of LOTOS allows good performances that could probably not be achieved for the whole language. Compared to other approaches for compiling and verifying LOTOS specifications, the limitations of CÆSAR are mild: CÆSAR handles the data part, unlike some translators [BC88] [ML88] that only deal with basic LOTOS; static control constraints are much weaker than those of another compiling algorithm for LOTOS [Dub89]; non-guarded recursion is allowed by CÆSAR, whereas it is forbidden by most simulators.

4.3 Prospects

The major drawback of model-checking techniques is *state explosion* [GRRV89]. Model-checkers generate large graphs and reduce them later, according to the properties to verify. Problems arise when the graphs are too large, either to be generated, or to be reduced. Clearly, the generated graphs contain more information than actually needed for verification. A solution would consist in improving model-checking to generate directly "already reduced" graphs.

Given a LOTOS specification, CÆSAR produces a graph which is strongly equivalent to the labelled transition system defined by the operational semantics of LOTOS. This graph is then minimized (e.g., by using ALDÉBARAN [Fer88] [Fer90]) modulo some equivalence weaker than strong equivalence. It would be much better to generate a graph reduced, partially or even completely, modulo this equivalence.

The stepwise translation technique of CÆSAR seems to provide an adequate framework for doing this. The basic idea is to perform such reduction not only *after* run-time, as it is the case when equivalences are applied to the graph model, but also *before* run-time, on the network model. This general principle has been practically experimented in two directions:

- the current version of CÆSAR implements a reduction technique named *safety reduction*, which applies an abstraction criterion to the network. During the optimization phase, it replaces all gates "τ" by "ε". Then, the simulation phase of CÆSAR generates the graph corresponding to the reduced network. This transformation is optional, since it does generally preserve neither strong nor observational equivalence (the resulting graph has no edge labelled "**i**"). However it preserves *safety equivalence* [Rod88], which is stronger than trace equivalence, and compatible with safety properties expressed in temporal logic.

  This approach has been applied satisfactorily to various examples. For instance, it gives significant results on the LOTOS specification of the token-ring scheduler [Mil80] with N cyclers. Without reduction the graph generated by CÆSAR has $O(2 \times 3^N)$ states (state explosion occurs for $N \geq 12$). By using safety reduction, CÆSAR directly produces a graph with only $N + 1$ states (this graph is also, as a matter of fact, the smallest graph observationally equivalent to the complete graph).

- the other reduction technique implemented in CÆSAR aims at preventing the undesirable effects of value flattening, which often generates a large number of "meaningless" states. More precisely, when a graph is reduced modulo some equivalence, many states are generally found to be equivalent, since they only differ with respect to the value of some variable of no interest for verification. In fact, the set of variables in a specification can be split into two classes:

- there are variables whose value is needed, either to determine the flow of control (e.g., variables used in boolean guards), or to evaluate the properties to be verified (e.g., variables used in temporal formulas). For these variables, value flattening is necessary.
- for other variables, value flattening should be avoided since the knowledge of the actual values of such variables is not essential to graph generation and verification. It would be better to handle these variables symbolically, by performing abstract interpretation.

A general approach to verification should therefore involve both symbolic computations and value flattening. CÆSAR judiciously combines symbolic computations (at compile-time) and value flattening (at run-time). This ensures efficiency, still taking advantage of syntactic transformations on algebraic terms (especially during the optimization phase).

Moreover, in the current version of CÆSAR, the user can prevent value flattening. This is simply done by providing appropriate concrete implementations for abstract data types. For each LOTOS sort, the user has to provide a C iterator which enumerates all concrete values in the domain of the sort; limiting this enumeration to a single value suppresses value flattening.

This technique has been successfully used for a number of examples. It is well-adapted to communication protocols, whose behaviors are determined by message headers, but are independent of message contents: value flattening is only applied to headers, whereas contents are handled symbolically. It has also been used to verify several specifications in LOTOS of systolic arrays computing the convolution product [Gar89a]: in these examples abstract data types are implemented by character strings representing algebraic terms, instead of being implemented by numeric types; the input values of the networks are not flattened, and the output values are symbolic expressions in terms of the input values.

A more elaborated approach would automatically determine which variables can be handled symbolically, by performing data flow analysis on the network model to find out variables whose value is never (transitively) used in a condition attached to a transition.

As a conclusion, it is likely that full verification of complex systems can be carried out in the model-checking framework. It is necessary, however, to combine model generation with reduction techniques, because “brute force” approach is in general not tractable. Such reductions on the network model have already been experimented with CÆSAR and seem to be fairly promising. Small reductions, at the network level, often lead to large ones, at the graph level. Moreover, they can be performed automatically, with no or little participation from the user. Developing other powerful reductions should be possible, since the network model used in CÆSAR is sufficiently simple and clean to allow accurate control and data flow analysis.

Acknowledgements

The authors are grateful to Susanne Graf, Daniel Pilaud, Carlos Rodriguez, Jacques Voiron, and the anonymous referees for their helpful comments and suggestions.

References

[AF88] Sukhvinder S. Aujla and Matthew Fletcher. The Boyer-Moore Theorem Prover and LOTOS. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 169–183. North-Holland, September 1988.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, January 1988.

[BC88] Tommaso Bolognesi and Maurizio Caneve. SQUIGGLES: A Tool for the Analysis of LOTOS Specifications. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 201–216. North-Holland, September 1988.

[Boo89] Rob Booth. An Evaluation of the LCF Theorem Prover using LOTOS. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*. North-Holland, December 1989.

[BvB90] Michel Barbeau and Gregor v. Bochmann. Deriving Analysable Petri Nets from LOTOS Specifications. Publication 707, Département IRO, Université de Montréal, January 1990.

[Dub89] Eric Dubuis. An Algorithm for Translating LOTOS Behavior Expressions into Automata and Ports. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*. North-Holland, December 1989.

[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.

[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 Verlag, 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.

[GL88] Renaud Guillemot and Luigi Logrippo. Derivation of Useful Execution Trees from LOTOS Specifications by Using an Interpreter. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 311–327. North-Holland, September 1988.

[GRRV89] Susanne Graf, Jean-Luc Richier, Carlos Rodríguez, and Jacques Voiron. What are the Limits of Model Checking Methods for the Verification of Real Life Protocols? 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 275–285. Springer Verlag, June 1989.

[GV90] Jan Friso Groote and Frits Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. CS-R 9001, Centrum voor Wiskunde en Informatica, Amsterdam, January 1990.

[ISO88] 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, Genève, September 1988.

[Led87] G. J. Leduc. The Intertwining of Data Types and Processes in LOTOS. In Harry Rudin and Colin H. West, editors, *Proceedings of the 7th International Symposium on Protocol Specification, Testing and Verification (Zurich)*. North-Holland, May 1987.

[MdM88] J. A. Manas and T. de Miguel. From LOTOS to C. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 79–84. North-Holland, September 1988.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[ML88] Saturnino Marchena and Gonzalo Leon. Transformation from LOTOS Specs to Galileo Nets. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 217–230. North-Holland, September 1988.

[NH84] R. De Nicola and M. C. B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[NQS89] Elie Najm, Jose Queiroz, and Ahmed Serhrouchni. Pre-Implementing and Verifying Process Algebras. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*. North-Holland, December 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 International Symposium on Protocol Specification, Testing and Verification (Zurich)*. IFIP, North-Holland, May 1987.

[SSC89] Pierre de Saqui-Sannes and Jean-Pierre Courtiat. From the Simulation to the Verification of ESTELLE* Specifications. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*. North-Holland, December 1989.

[Tau90] Dirk Taubner. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*, volume 369 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

[vE89] Peter van Eijk. *The Design of a Simulator Tool*. In Peter van Eijk et al., editors, *The Formal Description Technique LOTOS*. North-Holland, 1989.