A Toolbox for the Verification of LOTOS Programs

Jean-Claude FERNANDEZ* Hubert GARAVEL† Laurent MOUNIER*
Anne RASSE* Carlos RODRIGUEZ† Joseph SIFAKIS*

Abstract

This paper presents the tools ALDÉBARAN, CÆSAR, CÆSAR.ADT and CLÉOPÂTRE which constitute a *toolbox* for compiling and verifying LOTOS programs. The principles of these tools are described, as well as their performances and limitations. Finally, the formal verification of the *rel/REL* atomic multicast protocol is given as an example to illustrate the practical use of the toolbox.

Keywords: reliability, formal methods, LOTOS, verification, validation, model-based methods, model-checking, transition systems, bisimulations, temporal logics, diagnostics

Introduction

There is an increasing need for reliable software, which is especially critical in some areas such as communication protocols, distributed systems, real-time control systems, and hardware synthesis systems. It is now agreed that reliability can only be achieved through the use of rigorous design techniques. This has motivated a lot of research on specification formalisms and associated verification methods and tools.

Verification means the comparison of a *system* — given as a *program* in a language with a formal operational semantics — with its *specifications*, namely the description of its expected service. There are two main approaches to the verification problem:

proof-based methods which attempt to carry out verification at the source program level using theorem provers;

model-based methods which first translate the source program into a (possibly finite) *model*, then comparing this model with the specifications.

In this paper, only model-based methods are considered. The reasons for this choice are discussed in [GS90]: basically, model-based methods are expected to be more efficient on practical examples and they are also fully automated. However, their main limitation is their complexity (often exponential in the size of the program).

Model-based techniques are usually divided into two classes, according to the formalism used for the specifications [Sif86]. Both are useful, depending on the nature of the specification to verify:

behavioral specifications describe the *behavior* of the system, possibly taking into account some abstraction criteria. Such specifications are usually expressed in terms of transition systems (i.e., states and transitions between these states). Process algebras, automata, or even the language used to describe the system under verification, can be used for this purpose.

As the program and its behavioral specifications are both represented by transition systems, verification consists in comparing them with respect to some equivalence or preorder relation. Any decision procedure for such relations defines a verification method.

logical specifications characterize the *global properties* of the system, such as deadlock freedom, mutual exclusion, or fairness. *Temporal logics* have been designed so as to express such properties.

In this case, a *satisfaction relation* between the program and logical formulas is defined; a property is an assertion stating that the program satisfies a given formula. Any decision procedure for the satisfaction relation constitutes a verification method.

A well-studied class of verification methods for finite systems is based on the model of *transition systems* (also *labeled transition systems*, *finite state automata*, or *state graph*, or *graphs*). In fact, the study of parallel and reactive systems has shown that the essential features of their functioning can be modeled using finite transition systems.

*LGI-IMAG, IMAG Campus, BP 53X, 38041 GRENOBLE cedex, FRANCE; e-mail: {fernand, mounier, rasse, sifakis}@imag.fr

†VERILOG Rhône-Alpes, La Cascade — Le Pré Milliet, 38330 MONTBONNOT SAINT-MARTIN, FRANCE; e-mail: {hubert, crodrig}@imag.fr

Many formalisms have been proposed for describing parallel systems; among them, the standardized Formal Description Technique LOTOS [ISO88b] holds the attention of the scientific community. LOTOS (*Language Of Temporal Ordering Specification*) combines ideas from process algebras (especially CSP [Hoa78] [Hoa85] and CCS [Mil80]) and abstract data types (namely ACTONE [EM85]).

This language has been the object of extensive studies, leading to the production of tools covering various user needs: simulators [GHHL88] [vE89], compilers [MdM88], program analysis [QPF89], etc. However, the application of model-based techniques to LOTOS has not been studied enough, despite the fact that LOTOS semantics is fully formalized.

This paper presents a model-based *toolbox* for the formal verification of LOTOS programs. This toolbox provides an integrated set of tools, able to deal with both behavioral and logical specifications. The paper is organized as follows. Section 1 describes the overall architecture of the toolbox. Sections 2 up to 5 present in turn each of the toolbox components. Section 6 illustrates the toolbox use for the verification of an atomic multicast protocol.

1 Presentation of the toolbox

The toolbox components can be conceptually divided into two classes:

compilers: they translate the program into a graph. The toolbox contains two compilers, CÆSAR.ADT and CÆSAR, which respectively handle the data and control parts of LOTOS programs.

verifiers: they perform verifications on the graphs generated by compilers. The toolbox contains a verifier for behavioral specifications, ALDÉBARAN, which compares two graphs with respect to several equivalence and preorder relations, and a verifier for logical specifications, CLÉOPÂTRE, which evaluates formulas of the branching-time temporal logic LTAC [QS83] on a graph. If the graphs under verification are not correct, both tools provide *diagnostics* in terms of execution sequences, in order to help the user to understand the reason of the error.

Figure 1 illustrates the toolbox architecture in a typical situation: a protocol is checked against its expected service. The protocol to verify is described in LOTOS, and then translated into a graph. The service is either specified by a LOTOS program (translated into a graph), or by LTAC logical formulas.

2 The CÆSAR.ADT tool

CÆSAR.ADT [Gar89b] is a compiler which translates the abstract data type definitions of a LOTOS program into a C program. Therefore CÆSAR.ADT produces automatically a corresponding prototype implementation from a formal specification.

2.1 Functional description

CÆSAR.ADT takes as input a LOTOS program. Only the data part of this program is taken into account (process definitions are discarded).

CÆSAR.ADT generates as output a C library containing for each sort (*resp.* operation) defined in the LOTOS program a corresponding C type (*resp.* function).

Some *special comments* must be inserted in the LOTOS program, in order to provide a correspondence between the names of LOTOS objects and the names of the C objects implementing them.

CÆSAR.ADT lays down restrictions on the subset of accepted LOTOS programs. The user has to follow the *constructor programming* discipline, consisting in:

- Dividing the set of LOTOS operations into *constructors* (primitive operations) and *non-constructors* (derived operations, defined in terms of constructors). Constructor operations must be explicitly indicated using special comments.

- Orienting the equations, in order to consider them as term rewrite rules, the right-hand side specifying how the left-hand side is to be rewritten. The rewrite strategy used is call-by-value, enhanced with a decreasing-priority rule between equations (whenever several equations simultaneously apply, the one which appears first in the LOTOS source text is selected).

- Making sure that each equation has either the form:

  $$f(v_1, ..., v_n) = v$$

  or the conditional form:

  $$c_1, ..., c_p \Longrightarrow f(v_1, ..., v_n) = v$$

  where:

  - $n \geq 0$ and $p \geq 1$;
  - f is a non-constructor operation;
  - $v_1, ..., v_n$ are terms containing only constructors and universally quantified variables;
  - $v, c_1, ..., c_p$ are terms such that any variable occurring in these terms also occurs in some v_i.

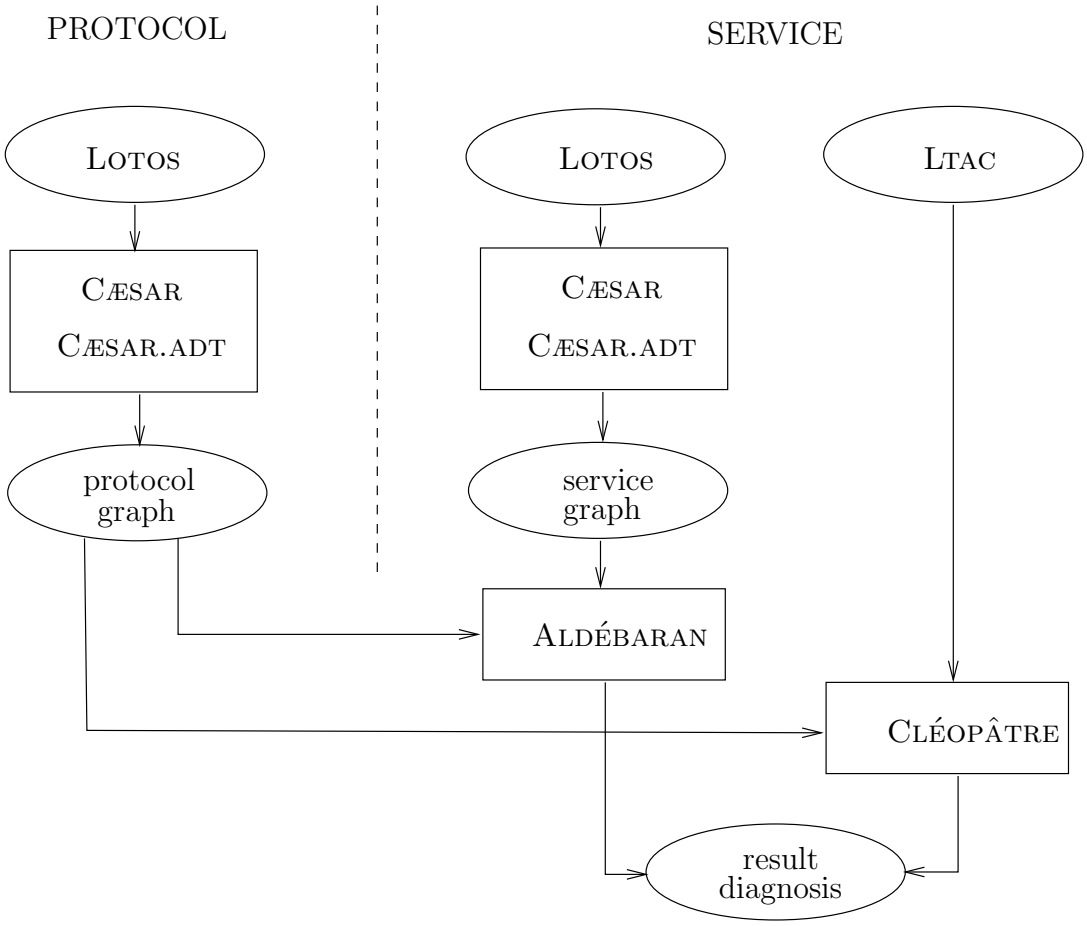

Figure 1: Architecture of the LOTOS toolset.

These restrictions achieve a good tradeoff between expressiveness and efficiency. Practically, they lead to a methodology for defining abstract data types in a simple manner and, provided that the constructors are known, any LOTOS data type definition can be transformed into the required form (even in presence of conditional equations). Due to these restrictions, CÆSAR.ADT can generate fastly an efficient code and carry out a number of static and dynamic verifications, in order to check the completeness and consistency of the equations.

CÆSAR.ADT allows sorts and operations to be declared "external", which means that the implementation in C of those sorts and operations is provided by the user, instead of being generated automatically by CÆSAR.ADT. This feature is very useful in practice: when efficiency (in space or time) is needed, the most critical parts of a LOTOS program can be coded manually; the changes needed for such an implementation refinement are minimal and do not mess up the formal specification. This feature also allows to interface existing code: sorts and operations can be mapped onto hand-written data structures and functions; it is therefore possible to create abstract "views" of existing software, which is suitable for reverse engineering.

The current version of CÆSAR.ADT has de facto limitations (for instance, parameterized types are not handled yet). However, this is only a matter of implementation, not a theoretical impossibility.

2.2 Functioning principles

CÆSAR.ADT translates a declarative formalism (LOTOS abstract data types) into an imperative language (the C language). After the syntax analysis phase — using the SYNTAX[1] compiler construction system — and the static semantics analysis phase, the translation is performed in three steps:

- The first step performs various verifications and transforms the equations in order to put them under a suitable form for further processing.

- The second step determines the implementation of sorts and constructors. The way a given sort S is represented in C only depends on the set of constructors returning a result of sort S. CÆSAR.ADT uses a general algorithm able to compile any sort, enhanced by a collection of specialized algorithms which provide optimal implementations for particular cases of common use (numerals, enumerations, and tuples).

- The third step implements the non-constructors, using a *pattern-matching* compiling algorithm [Sch88]. The C function generated for a non-constructor F is built from the equations defining F.

2.3 Results and perspectives

Most of the algorithms used in CÆSAR.ADT are linear, so that *combinatorial explosion* (in time, space, or size of the generated code) does not happen.

For instance, a "real" LOTOS program containing 2 000 lines of abstract data types (20 sorts and 170 operations) compiles in 20 seconds on a SUN SparcStation and produces 6 000 lines of C code. Moreover, it is proven that the generated code is, in some sense, optimal [Sch88].

At present, CÆSAR.ADT is mainly used in conjunction with CÆSAR to carry out formal verification of protocols and distributed systems. But abstract data types are naturally applicable to other areas; for example, CÆSAR.ADT has beed used:

- to develop quickly prototype compilers, one for the extended temporal logic XTL and another one for the timed process algebra ATP [NS90];

- to obtain a prototype implementation of the MAA (Message Authenticator Algorithm) standard [ISO92] from its formal description in LOTOS [Mun91].

The current version of CÆSAR.ADT is approximately 5 500 lines of C code. A major rewriting of CÆSAR.ADT has been undertaken. Most of the new version will be written directly in LOTOS abstract data types (instead of C in the current version). The new version will be self-compiled, using the existing version for bootstrapping.

It should bring various improvements, such as the optimal implementation for a broader class of type definitions, the removal of certain restrictions on the form equations by applying semantic transformations, and the introduction of memory management schemes specifically adapted to model-based verification.

3 The CÆSAR tool

CÆSAR [Gar89a] [GS90] is a tool for compiling and verifying LOTOS programs according to the model-based approach[2]. It translates the source program into a graph which describes all possible evolutions. The edges of the graph are labeled by *actions* corresponding to LOTOS rendez-vous; each action consists of a synchronization gate, possibly accompanied by the list of values sent or received during the rendez-vous communication. The states of the graph are labeled by the values of program variables , i.e., the local variables of all concurrent processes.

[1] SYNTAX is a registered trademark of INRIA

[2] The development of this tool was supported in part by INRIA

3.1 Functional description

CÆSAR takes as input the LOTOS program to verify and a C implementation of the abstract data types (either written by hand, or automatically generated by CÆSAR.ADT), as shown on Figure 2.

CÆSAR generates as output an extended Petri net and a graph. The informations contained in the graph can be exploited by various tools (automata minimizers, temporal logic or μ-calculus evaluators, diagnostic tools). CÆSAR is able to generate this graph in multiple formats to interface with various existing tools: ALDÉBARAN and CLÉOPÂTRE, but also AUTO and AUTOGRAPH (INRIA), MEC (University of Bordeaux), PIPN (LAAS-VERILOG), XESAR (LGI-IMAG), etc.

CÆSAR lays down the following restriction: to be accepted, a LOTOS program must not contain any recursive process instantiation either on the left or right-hand side of a parallel operator "|[...]|", or on the left-hand side of an enabling operator ">>", or on the left-hand side of a disabling operator "[>".

These constraints aim at forbidding the dynamic creation/destruction of process instances. In practice, they reach a good compromise between the expressive power left to the user and the efficiency of the compilation and verification algorithms. This issue is discussed in [GS90].

3.2 Functioning principles

After the phases of syntax and static semantics analysis (the same as in CÆSAR.ADT), the translation of a LOTOS program into a graph is performed in four successive steps, formally defined in [Gar89a] and summarized in [GS90]:

- The *expansion* phase translates the LOTOS program into an equivalent SUBLOTOS program, SUBLOTOS being a process algebra which can be viewed as a simplified subset of LOTOS.

- The *generation* phase translates the SUBLOTOS program into an intermediate form, called *network*, defined by:

  - a *control part*, represented as a Petri net, consisting of *places* and *transitions*, with an additional structure of *units* to keep track of the communicating processes decomposition which exists in the LOTOS source program;
  - a *data part*, consisting of global and typed *variables*, the values of which can be accessed or modified by *actions* attached to the transitions.

- The *optimization* phase attempts to reduce the network complexity (i.e., to decrease the number of places, transitions, units, and variables) by applying transformations which preserve strong bisimulation equivalence (*cf.* § 4.2). These optimizations address both the control part of the network (using Petri nets-like transformations) and the data part (using data flow analysis techniques similar to those used in compiler optimizers).

- The exhaustive *simulation* phase produces the reachability graph corresponding to the optimized network. All visited states are stored in main memory, whereas the edges of the graph are written on a file as soon as they are generated. Basically, this phase is analogous to the marking graph construction for a Petri net, but it also takes into account the data part (this is done by generating, compiling and finally executing a C program which includes the C implementation provided for the abstract data types of the LOTOS program).

3.3 Results and perspectives

The current version of CÆSAR is about 40 000 lines of C code. It can generate graphs of approximately one million states (on SUN workstations). The generation speed may reach 500 states per second, but it strongly depends on the program considered and the amount of main storage available.

Among various applications of CÆSAR, one can mention the verifications of an atomic multicast protocol [BM91] (*cf.* § 6), a subset of the FIP protocol [ADV90], and a overtaking protocol for cars [EFJ90].

At present, model-based methods constitute a realistic solution for verifying non-trivial programs. However, they may fail when applied to large programs, due to the *state explosion problem* (see, for instance, [GRRV89]).

In fact, model-based verifiers attempt to generate large graphs (as does CÆSAR during its simulation phase) which may have to be reduced later (for instance, by using ALDÉBARAN). An attractive alternative would consist in verifying not *after* but *during* the simulation phase [JJ89], or even *before* it [GS90]. Such approaches are currently experienced in several ways.

By applying, during the optimization phase, various transformations of the Petri nets generated by CÆSAR, reductions at the network level are obtained which, even small, lead to large reductions at the graph level. Such reductions preserve either strong bisimulation equivalence or weaker equivalences, such as safety equivalence [Rod88].

At the same time, work started in order to extend the functionalities of CÆSAR beyond mere graph generation. This initiative, named OPEN/CÆSAR, aims at using the compiling algorithms of CÆSAR as a basis for elaborate verification techniques, especially "intensive" simulation [Wes86] [JGM88] [Hol89] [ACD+93], "on the

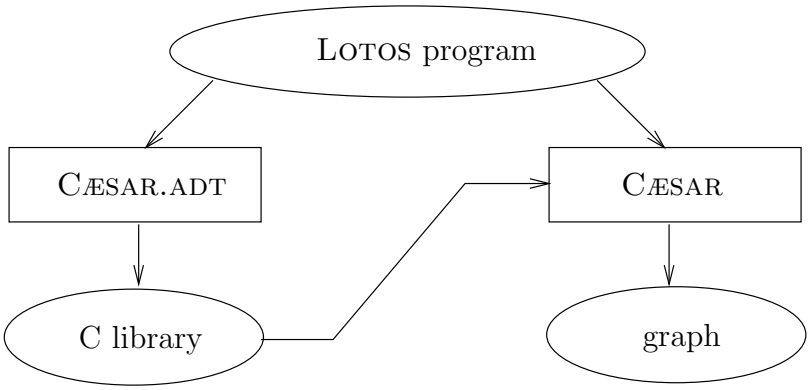

Figure 2: Combined use of CÆSAR and CÆSAR.ADT.

fly" methods [JJ89] [JJ91] [Mou92], and "partial order" methods [Val90] [VT91] [GW91a] [GW91b].

The OPEN/CÆSAR tool should also have applications in other domains than verification, for instance interactive simulation, sequential or distributed code generation, and test case generation. In this respect, several prototype tools based on OPEN/CÆSAR have already been developed.

4 The ALDÉBARAN tool

ALDÉBARAN [Fer88] [Fer90] [FM91a] [FM91b] is a tool performing reduction and comparison of graphs according to various equivalence relations and preorders.

Several equivalence and preorder relations have been proposed for the verification of parallel systems, among which *strong bisimulation equivalence* [Par81] plays a central role. This equivalence, characterized by an elegant fixed-point definition, is too strong from a program verification point of view: it does not take into account abstraction criteria, especially the concept of *silent* (or *internal*, or *invisible*) actions [Mil80]. However, there are *weaker* equivalence relations (supporting abstraction criteria) whose definitions are based on strong equivalence. Furthermore, the algorithms associated with strong equivalence can be extended to tackle such weaker equivalences.

Apart from strong bisimulation, ALDÉBARAN implements other weaker bisimulation-based relations, namely:

- observational equivalence [Mil80]
- acceptance model equivalence [GS86]
- branching equivalence [vGW89] and its preorder
- delay equivalence [NMV90]
- $\tau^* a$ equivalence [FM90] and its preorder
- safety equivalence[3] [Rod88] and its preorder.

4.1 Functional description

ALDÉBARAN has two major functionalities corresponding to distinct practical needs:

graph comparison allows to compare, modulo various equivalences or preorder relations, the graph of a program with the graph of its behavioral specification . In this case, the chosen relation and both graphs are given as inputs to ALDÉBARAN and the result of the comparison ("*true*" or "*false*") is obtained as output. If the result is *false*, ALDÉBARAN also provides *diagnostic sequences* leading from the initial states of the graphs to two immediately distinguishable states (i.e., states that do not accept the same set of actions).

graph reduction allows to generate the *quotient* of a graph with respect to a given equivalence relation, i.e, the smallest graph which is equivalent to the original one. In this case, the graph and the equivalence relation are given as inputs, and the quotient is obtained as output.

4.2 Functioning principles

Two main approaches exist for deciding whether two graphs are strongly bisimilar. Both of them can be extended to weaker bisimulation-based relations.

- The first approach consists in computing successive refinements of an initial partition of the states of the graph. When stabilization is reached, the partition obtained coincides exactly with the equivalence classes of strong bisimulation (which are

[3] Safety equivalence is of interest because it exactly characterizes *safety properties* [BFG+91] (see also section 5.1).

the states of the quotient graph). Two graphs are strongly bisimilar if and only if their quotients are identical (modulo renaming of states). An efficient partition refinement algorithm, proposed by Paige & Tarjan [PT87], is implemented in ALDÉBARAN.

This approach also applies to weaker bisimulation-based relations. This is achieved by modifying each graph, taking into account abstraction criteria, and then computing its quotient with respect to strong bisimulation.

The major drawback of this method lies in the fact that the application of abstraction criteria is done by adding new transitions to the graph. Therefore, the number of transitions may become too large for the available memory space.

Moreover, if two graphs are not related, the resulting diagnostic sequences are not easy to analyze, since they come from the quotient graphs, and not from the original graphs. It is therefore difficult for the user to interpret such diagnostics in terms of the source LOTOS program.

- The second approach consists in comparing the two graphs by performing a depth-first traversal of their *synchronous product* ("on the fly" comparison [FM90]). By varying this product, several bisimulation-based equivalences and preorders can be processed. However, unlike the first approach, these algorithms only perform comparisons but not reductions.

  Theoretically, the time and space complexities of the "on the fly" algorithm are higher than Paige & Tarjan's ones in the general case. However, the time complexity is reduced when one of the two graphs is deterministic, which is usually the case in practice. Similarly, it was shown that its space complexity can be reduced without significantly increasing the time required to perform the check [JJ91]. Thus, its implementation in ALDÉBARAN allowed to verify larger graphs and to reduce the execution time. Moreover, this algorithm provides diagnostic sequences extracted from the original graphs.

4.3 Results and perspectives

The current version of ALDÉBARAN is about 24000 lines of C code and 6 000 lines of C++ code. Its performances are fair: only a few minutes are needed (on SUN workstations) to reduce graphs of some thousands states modulo strong bisimulation or observational equivalence, or to compare graphs of more than one hundred thousands states modulo safety equivalence.

More often than not, ALDÉBARAN is used with CÆSAR, for instance to compare the graph of a protocol with the graph of its expected services. ALDÉBARAN is also integrated in other tools: it plays the role of an internal component used to reduce graphs. At present, interfaces exist with CLÉOPÂTRE and OCMIN, an optimizer of the OC code generated from the synchronous languages LUSTRE and ESTEREL.

Keeping graphs in memory, as it was done initially in ALDÉBARAN, makes the processing of very large graphs prohibitive. A possible solution to this problem is to combine the comparison and the graph generation phase.

The latest version of ALDÉBARAN [Mou92] implements such an approach. It accepts as input a *composition expression* consisting of parallel processes composed together using the LOTOS operators of parallel composition ("`|[...]|`") and hiding ("**hide**"). A given LOTOS specification (usually a protocol definition) can be translated into such a composition expression by splitting it into a tree of parallel processes. The leaves of the tree are translated into graphs using CÆSAR. In a next step ALDÉBARAN is used to compare the composition expression with another graph (usually the service definition). The comparison is done "on the fly" and does not need to build the complete graph corresponding to the comparison expression. Notice that the splitting operation must be done by hand, since there are often many possible solutions; choosing the "best" splitting requires knowledge of the protocol and the verification techniques.

Another approach experimented in ALDÉBARAN relies on a *minimal model generation* algorithm [BFH90] which is implemented using Binary Decision Diagrams [Bry86] [EFT91]. The prototype implementation allows, starting from a composition expression, to generate directly its quotient graph modulo various equivalence relations, without generating the whole graph.

5 The CLÉOPÂTRE tool

CLÉOPÂTRE is a validation tool for specifications expressed in the branching-time temporal logic LTAC [QS83], which is expressively equivalent to CTL [CES83]. This tool includes:

- a verification module [Rod88], which checks the validity of the formulas on a graph generated from the source program,

- an explanation module [Ras90], which provides, when a formula is not valid, diagnostic sequences extracted from the graph.

5.1 Functional description

CLÉOPÂTRE takes as inputs a graph generated by CÆSAR from the LOTOS program to verify and a set of LTAC formulas. The subset of LTAC formulas used for the verification of LOTOS programs is described by the following grammar:

$$T \mid init \mid enable(a) \mid after(a) \mid sink \mid$$
$$f \wedge g \mid \neg f \mid inev[f]g \mid pot[f]g$$

where f and g are formulas and a a label attached to a transition of the graph.

The models of the formulas of this logic are *execution trees*, i.e., infinite trees obtained by unfolding the graph from one of its state. A state s *satisfies a formula* f, which we note $s \models f$, if and if only s is the root of an execution tree for f. Intuitively, the relation $\models$ is defined as follows:

- any state satisfies T;
- the initial state of the program satisfies $init$;
- a state s satisfies $enable(a)$ if it is possible to execute action a from state s;
- a state satisfies $after(a)$ if it can only be reached immediately after the execution of action a;
- a state satisfies $sink$ if it has no outgoing transition;
- a state satisfies $f \wedge g$ if it satisfies f and g;
- a state satisfies $\neg f$ if does not satisfy f;
- a state s satisfies $inev[f]g$ if, for every execution of the program from s, f is true until g becomes true;
- a state s satisfies $pot[f]g$ if there exists an execution from s such that f is true until g becomes true.

In the sequel, we use the abbreviations:

$$\begin{aligned} al[f]g &= \neg pot[f]\neg g \\ some[f]g &= \neg inev[f]\neg g \\ f \vee g &= \neg(\neg f \wedge \neg g) \\ f \Rightarrow g &= \neg f \vee g \end{aligned}$$

We call *temporal operators* the operators *pot*, *some*, *al*, and *inev*. They are used to express usual properties of the protocols. These properties are divided into two classes [AL88]:

- *safety properties*, meaning that something "bad" can never happen. Such properties are expressed by formulas like "$\neg pot\ bad$" or "$al\ \neg bad$",
- *liveness properties*, meaning that something "good" will eventually happen. These properties are expressed by formulas like "$inev\ good$".

CLÉOPÂTRE gives as output the result of the evaluation of each formula, i.e., one of the following messages: *valid formula*, *formula always false* or *formula false for k states out of n*. If the formula is not valid, CLÉOPÂTRE also provides diagnostics, in order to explain the reason of the error. As safety and liveness properties characterize *all* the executions of the program, it is sufficient to exhibit a single *counter-example* execution sequence.

5.2 Functioning principles

Logic formulas are evaluated by optimized algorithms performing graph traversal [Rod88]. Every operator of LTAC is evaluated by an algorithm which is linear (in space and time) with respect to the size of the graph.

When a formula f' is not valid, CLÉOPÂTRE explains why there exists a state s satisfying f, where $f = \neg f'$. Then, a diagnostics for "f' *is not valid*" is an *explanation* of $s \models f$. The first step of the generation of an explanation of $s \models f$ consists in rewriting f under canonical form $\bigvee_i \bigwedge_j f_{ij}$, where all f_{ij} are either:

- predicates T, $init$, $sink$, $enable(a)$, $after(a)$ or their negation,
- or formulas like $op[f]g$, where $op = al$, $inev$, pot or $some$, f and g being themselves under canonical form.

Then, explanations are generated by structural induction on f, the outermost operator being processed first:

- for T, $init$, $sink$ and their negations, the explanation is directly obtained by considering state s;
- for $after(a)$, $enable(a)$, and their negations, the explanation is obtained by considering the ingoing and outgoing transitions of s;
- for $f \wedge g$ (*resp.* $f \vee g$), it is necessary to explain why $s \models f$ and (*resp.* or) $s \models g$;
- formulas like $al[f]g$ and $inev[f]g$ cannot be explained in terms of executions sequences, because these formulas express properties of all executions of the program; an explanation would therefore contain the set of all executions from one state, which is usually too large to be processed. Practically, this case does not happen when only safety and liveness properties are considered.
- for the operators *pot* and *some*, an explanation is generated by searching a path from s, such that the states of this path satisfy given conditions. The explanation of the *some* operator may require to exhibit an infinite path, which is done by computing the strongly-connected components of the graph.

Among all the possible paths, those with minimal length are selected. Furthermore, CLÉOPÂTRE can display concisely the set of all these paths in term of an ω-regular expression over the set of actions of the program.

As the formula evaluation algorithm, the diagnostic generation algorithm is linear in time and space.

5.3 Results and perspectives

CLÉOPÂTRE was first implemented for XESAR [Rod88], a verification tool for a variant of ESTELLE [ISO88a]. Then, CLÉOPÂTRE was adapted in order to provide for the need of a diagnostic tool for LOTOS.

The current version of CLÉOPÂTRE (about 23000 lines of C code) is able to analyze graphs with several hundred thousands states. A formula with two nested temporal operators is usually evaluated in a few seconds on a SUN workstation.

Further evolution will consist in adapting this tool to other specification formalisms, such as observers [JGM88]. Also, CLÉOPÂTRE could be extended to generate test-case sequences guided by properties, since this problem is closely related to the diagnosis problem.

6 Verification of the rel/REL protocol

This section describes how the toolbox was used to verify an atomic multicast protocol. This experience is reported in [BM91] and [Mou92]. Another example of formal verification of a similar protocol can be found in [BGR+90].

6.1 The *rel/REL* Service

The *rel/REL* protocol [SE90] supports atomic communications between a transmitter and several receivers, in spite of an arbitrary number of failures from the stations involved in the communications. Two versions of the *rel/REL* protocol are described in [SE90]; this section focuses on the rel/REL_{fifo} version which preserves the order of the messages sent by a given transmitter.

The service provided by the rel/REL_{fifo} protocol consists of the two following (informal) properties, which are both safety properties (*cf.* section 5.1):

atomicity: if a station E sends a message m to a group of stations G, then either all the functioning elements of G receive m, or none of them does, even if several crashes occur in the group $\{E\} \cup G$.

causality: if a station E sends a sequence of messages, in a defined order, to a group of stations G, then no functioning element of G may receive these messages in a different order.

Such an atomic multicast service has various applications. For instance, it is useful to implement transactions in distributed databases; it is also needed to manage the copies of replicated objects in fault-tolerant systems, where integrity constraints between the copies of an object have to be ensured.

6.2 The *rel/REL* Protocol

The *rel/REL* protocol is built on a *transport* layer protocol which provides a reliable (i.e., atomic and causal) message transmission between any *pair* of stations. In case of crash, stations are supposed to have a *fail-silent behavior*: they stop to send and to accept messages. It is also assumed that, even if multiple crashes occur, the network remains strongly connected: all functioning stations may still exchange messages.

The protocol is based on the *two phase commit* algorithm: the transmitter sends two successive copies of the message to all receivers; each message is uniquely identified, and an additional label indicates whether it is the first or the second copy. On receipt of the first copy, a station S waits for the second one; if it does not arrive before the expiration of a delay, then S assumes that the transmitter crashed and that some of the receivers may have not received a copy of the message. Then, S relays the transmitter and multicasts the two copies of the message, still using the *rel/REL* protocol. To reduce the network traffic, a station stops to relay as soon as a second copy of the message is received from the transmitter or from any other receiver.

6.3 Formal Description of the Protocol

LOTOS proved to be suitable for the description of the protocol and its data structures (message numbers, station addresses, tables and queues for storing received messages). The description process was straightforward; the only difficulty was the modeling of the fact that a station may crash at any moment. This problem was solved using the *constraint-oriented* programming style of LOTOS: the behavior of a station is represented by the parallel composition of a process describing the normal behavior and another process describing the possible failures.

6.4 Formal Verification of Atomicity

As defined above, atomicity means that "*an emitted message is either received by all its functioning receivers, or is not received by any of them*". If get_i denotes the receipt of a first copy of a message by station i, and $crash_i$ the crash of station i, this property can be rephrased in the following way: "*for any pair of stations (i,j), there is no execution sequence containing the action* get_i*, not*

containing the action $crash_i$ (station i received a message and it has not crashed), *and containing neither the action* get_j *nor the action* $crash_j$ (station j is still waiting for the message)."

The atomicity property can be expressed using the LTAC temporal logic. Let $waiting_i$ denote the fact that station i is waiting for a message (it has not received it yet and it has not crashed):

$$waiting_i = \neg after(get_i) \wedge \neg after(crash_i)$$

Moreover, the *fail-silent behavior* assumption implies that a crashed station i cannot receive a message after a crash:

$$al[T](crash_i \Rightarrow al[T](\neg get_i))$$

The atomicity property is then expressed by the following formula:

$$\neg \bigvee_{i \neq j} pot[waiting_j] f_{i,j}$$

where:

$$f_{i,j} = after(get_i) \wedge some[T](waiting_j) \wedge \neg after(crash_i)$$

This property was verified using CLÉOPÂTRE on a graph generated by CAESAR (50 000 states and 150 000 transitions); the LOTOS program described a configuration with a single transmitter, two receivers, and a single message sent (an analysis of the protocol [BM91] has shown that it is sufficient to verify the protocol for a single message).

6.5 Formal Verification of Causality

The second service property concerns the preservation of the message order: "*messages from a given transmitter are received in the same order as they were sent*". This is a safety property expressing that the received messages respect some conditions, but not ensuring their receipt. As it associates a transmitter and a receiver, it is sufficient to verify it for any pair (transmitter, receiver).

This property can easily be expressed using a transition system. Assuming that messages sent by a transmitter are identified by unique numbers $1, 2, ..., n$ according to their emission order, the expected behavior of a receiver can be represented, modulo appropriate abstraction and using safety equivalence, by the graph on Figure 3.

In this example, safety equivalence proves to be interesting, since it allows a straightforward expression of the service property (using safety equivalence here is allowed, since the property to be proven is a safety property). Should another equivalence (e.g., observational equivalence [Mil80] or branching bisimulation [vGW89]) be used, the transition system needed to express causality would be much more complex.

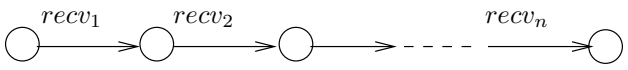

Figure 3: Graph expressing the atomicity property.

A first verification [BM91] of the causality property was carried out on a LOTOS specification describing a configuration with a single transmitter, two receivers and three different messages. ALDÉBARAN was used to compare, modulo safety equivalence, the graph of Figure 3 with the graph produced by CÆSAR (650000 states and 2 000 000 transitions). Due to the size of the latter, the comparison was carried out using the "on the fly" algorithm. The generation/reduction took less than 6 hours on an HP 9000 computer. Notice that the most recent version of CÆSAR generates, for the same example, a smaller graph (125 000 states and 430 000 transitions) in less time (30 minutes on a DEC Station 5000 with 24 Mbytes main memory), because it manages to identify classes of strongly-equivalent states at compile-time.

A second verification [Mou92] of the causality property was performed using compositional reduction (*cf.* section 4.3). The LOTOS specification was manually split into 5 communicating processes. CÆSAR was used to generate the 5 corresponding graphs (the largest one had only 10 000 states and 200 000 transitions). By applying parallel composition and reduction alternately, ALDÉBARAN produced a graph with 3 000 states and 10 000 transitions. The full verification took about 15 minutes on a SUN SparcStation.

The formal verification of the rel/REL_{fifo} protocol revealed ambiguities in the informal description of the protocol. Furthermore, it provided additional informations of interest to implementors: for instance, it has shown that messages queues are always of bounded size; the value of the upper-bound was discovered automatically.

Conclusion

This paper has presented a set of tools intended to the formal verification of systems described in LOTOS. Our contribution is motivated by the following ideas.

Formal verification tools are badly needed in software development. We believe that such tools should be "as automated as possible"; our work is oriented as to achieve such a goal. This distinguishes our approach from others based on axiomatic proofs, which require interaction with the user; their efficiency and effectiveness strongly relies upon users's skills and efforts.

We think that the actual state-of-the-art is not sufficiently advanced to allow rigorous software engineering methodology based on successive refinements with multiple formalisms. Indeed, approaches using different languages (one for specification, another one for implementation, possibly with additional intermediate languages) suffer from the lack of automatic translators between various refinement levels, therefore leading to inconsistencies or ambiguities.

On the contrary, we use a single high-level language at the different design steps. Our approach relies on the existence a compiler generating intermediate forms from which executable code can be produced and various analysis tools can be applied. This ensures that "what you prove is what you execute".

Our approach needs a language with a *formally defined operational semantics*, in order to allow automated verification. This language must also be *abstract* enough to be used in the early steps of design. Yet, it must be *executable*, in the sense that it can be implemented efficiently on existing computers (even at the expense of increased compiler complexity). From our experience, we believe that the ISO language LOTOS is a good choice according to these criteria.

In our approach, formal verification is not dissociated from other problems, e.g., implementation, testing, etc. All these problems are tackled in the same framework; formal verification is the merely the problem which sets the hardest performance constraints. Efficient techniques developed for verification find immediately applications in other areas: our verification toolbox constitutes a kernel on which we intend to build a complete and integrated CASE environment, with LOTOS as backbone, including verification, simulation, debugging, prototyping, code generation, and test generation.

Although our toolbox is currently limited in some aspects, it can be used on non-trivial examples and projects. We believe that the current limitations will be circumvented by using new techniques (such as the "on the fly" and "partial order" techniques) and also existing techniques (such as symbolic analysis, control- and data-flow analysis at compile-time) that have not been applied yet to process algebras.

The implementation of the toolbox is approximately 100 000 lines of C and C++ code. The toolbox is distributed free of charge to universities and public research centers. It can be obtained by sending an e-mail request to `caesar@imag.fr`. It has been installed in more than 40 sites.

References

[ACD+93] B. Algayres, V. Coelho, L. Doldi, H. Garavel, Y. Lejeune, and C. Rodríguez. VESAR: A Pragmatic Approach to Formal Specification and Verification. *Computer Networks and ISDN Systems*, 25(7):779–790, February 1993.

[ADV90] Pierre Azema, Khalil Drira, and François Vernadat. A Bus Instrumentation Protocol Specified in LOTOS. In Juan Quemada, José Manas, and Enrique Vázquez, editors, *Proceedings of the 3rd International Conference on Formal Description Techniques FORTE'90 (Madrid, Spain)*. North-Holland, November 1990.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. SRC 29, Digital Equipment Corporation, August 1988.

[BFG+91] Ahmed Bouajjani, Jean-Claude Fernandez, Susanne Graf, Carlos Rodríguez, and Joseph Sifakis. Safety for Branching Time Semantics. In *Proceedings of 18th ICALP*. Springer Verlag, July 1991.

[BFH90] Ahmed Bouajjani, Jean-Claude Fernandez, and Nicolas Halbwachs. Minimal Model Generation. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 85–92. AMS-ACM, June 1990.

[BGR+90] M. Baptista, S. Graf, J.-L. Richier, L. Rodrigues, C. Rodríguez, P. Verissimo, and J. Voiron. Formal Specification and Verification of a Network Independent Atomic Multicast Protocol. In Juan Quemada, José Manas, and Enrique Vázquez, editors, *Proceedings of the 3rd International Conference on Formal Description Techniques FORTE'90 (Madrid, Spain)*. North-Holland, November 1990.

[BM91] Simon Bainbridge and Laurent Mounier. Specification and Verification of a Reliable Multicast Protocol. Technical Report HPL-91-163, Hewlett-Packard Laboratories, Bristol, U.K., October 1991.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[CES83] E. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic. In *10th Annual Symposium on Principles of Programming Languages*. ACM, 1983.

[EFJ90] Patrik Ernberg, Lars-åke Fredlund, and Bengt Jonsson. Specification and Validation of a Simple Overtaking Protocol using LOTOS. T 90006, Swedish Institute of Computer Science, Kista, Sweden, October 1990.

[EFT91] Reinhard Enders, Thomas Filkorn, and Dirk Taubner. Generating BDDs for Symbolic Model Checking in CCS. In K. G. Larsen and A. Skou, editors, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, Berlin, July 1991. Springer Verlag.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[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.

[FM90] Jean-Claude Fernandez and Laurent Mounier. Verifying Bisimulations "On the Fly". In Juan Quemada, José Manas, and Enrique Vázquez, editors, *Proceedings of the 3rd International Conference on Formal Description Techniques FORTE'90 (Madrid, Spain)*. North-Holland, November 1990.

[FM91a] Jean-Claude Fernandez and Laurent Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In K. G. Larsen and A. Skou, editors, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, Berlin, July 1991. Springer Verlag.

[FM91b] Jean-Claude Fernandez and Laurent Mounier. A Tool Set for Deciding Behavioral Equivalences. In *Proceedings of CONCUR'91 (Amsterdam, The Netherlands)*, August 1991.

[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.

[GHHL88] R. Guillemot, R. Haj-Hussein, and L. Logrippo. Executing Large LOTOS Specifications. In S. Aggarwal and K. Sabnani, editors, *Proceedings of the 8th International Workshop on Protocol Specification, Testing and Verification (Atlantic City, NJ, USA)*, pages 399–410. IFIP, North-Holland, 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.

[GS86] Susanne Graf and Joseph Sifakis. Readiness Semantics for Processes with Silent Actions. Rapport SPECTRE C3, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, November 1986.

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

[GW91a] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS 91), Amsterdam*. IEEE Computer Society Press, July 1991.

[GW91b] P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In K. G. Larsen and A. Skou, editors, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, Berlin, July 1991. Springer Verlag.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol89] Gerard J. Holzmann. Algorithms for Automated Protocol Validation. In Joseph Sifakis, editor, *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, June 1989.

[ISO88a] ISO/IEC. ESTELLE — A Formal Description Technique Based on an Extended State Transition Model. International Standard 9074, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[ISO88b] 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.

[ISO92] ISO/IEC. Approved Algorithms for Message Authentication — Part 2: Message Authenticator Algorithm. International Standard 8731-2, International Organization for Standardization — Banking, Genève, 1992.

[JGM88] Claude Jard, Roland Groz, and Jean-François Monin. Development of VEDA: A Prototyping Tool for Distributed Systems. *IEEE Transactions on Software Engineering*, 14(3), March 1988.

[JJ89] Claude Jard and Thierry Jeron. On-Line Model-Checking for Finite Linear Temporal Logic Specifications. 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 189–196. Springer Verlag, June 1989.

[JJ91] Claude Jard and Thierry Jéron. Bounded-Memory Algorithms for Verification On-the-Fly. In K. G. Larsen and A. Skou, editors, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, Berlin, July 1991. Springer Verlag.

[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.

[Mou92] Laurent Mounier. *Méthodes de vérification de spécifications comportementales : étude et mise en œuvre*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), January 1992.

[Mun91] Harold B. Munster. LOTOS Specification of the MAA Standard, with an Evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[NMV90] Rocco De Nicola, Ugo Montanari, and Frits Vaandrager. Back and Forth Bisimulations. CS R9021, Centrum voor Wiskunde en Informatica, Amsterdam, May 1990.

[NS90] Xavier Nicollin and Joseph Sifakis. The Algebra of Timed Processes ATP: Theory and Application. Rapport SPECTRE C26, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, December 1990.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, March 1981.

[PT87] Robert Paige and Robert E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

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

[QS83] Jean-Pierre Queille and Joseph Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[Ras90] Anne Rasse. *CLEO : diagnostic des erreurs en XESAR.* Thèse de Doctorat, Institut National Polytechnique de Grenoble, June 1990.

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

[Sch88] Philippe Schnoebelen. Refined Compilation of Pattern-Matching for Functional Languages. *Science of Computer Programming*, 11:133–159, 1988.

[SE90] Santosh K. Shrivastava and Paul. D. Ezhilchelvan. rel/REL: A Family of Reliable Multicast Protocol for High-Speed Networks. Technical Report (in preparation), University of Newcastle, Dept. of Computer Science, U.K, 1990.

[Sif86] Joseph Sifakis. A Response to Amir Pnueli's "Specification and Development of Reactive Systems". In *IFIP (Dublin, Ireland)*, pages 1183–1187, 1986. Invited conference.

[Val90] A. Valmari. A Stubborn Attack on State Explosion. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 25–42. AMS-ACM, June 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.

[vGW89] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[VT91] A. Valmari and M. Tienari. An Improved Failure Equivalence for Finite-State Systems with a Reduction Algorithm. In Bengt Jonsson, Joachim Parrow, and Björn Pehrson, editors, *Proceedings of the 11th IFIP International Workshop on Protocol Specification, Testing and Verification (Stockholm, Sweden).* IFIP, North-Holland, June 1991.

[Wes86] Colin H. West. Protocol Validation by Random State Explosion. In *Proceedings of the 6th IFIP International Workshop on Protocol Specification, Testing and Verification (Montréal, Canada).* IFIP, North-Holland, June 1986.