Int J STTT (1997) 1: 166–183

Protocol verification with the ALDÉBARAN toolset

Marius Bozga, Jean-Claude Fernandez, Alain Kerbrat*, Laurent Mounier

Verimag centre Equation**, 2 rue de Vignate, F-38610 Gières, France; http://www.imag.fr/VERIMAG

Abstract. The design of distributed systems is an increasingly complex task, yet competitiveness requires faster developments. Formal Description Techniques (FDT) are a way to deal with this requirement, as they come with tools allowing us to simulate and verify the behavior of a system without actually having to execute it, thus translating part of the costly testing effort to the design effort. In this article, we present the verification toolset ALDÉBARAN. This verification toolset is designed independently of any FDT, yet allows us to work with the two most used ones. It is implemented in a modular way, for easy use and integration with other system design tools. We present the technical principles of this toolset, the performances obtained, and the application domains through the presentation of some case studies.

Key words: Protocol engineering – Formal methods – Verification – Model checking

1 Introduction

Distributed systems in general and more particularly telecommunications systems are more and more complex, yet the time-to-market becomes shorter and shorter, as the competition between service manufacturers is increasing. It becomes crucial to develop systems rapidly, at the lowest cost, with a good quality level. To keep up with these various and conflicting goals, engineering teams need to evolve more and more towards reuse (integration of previously implemented software components in a new application) and concurrent engineering (development in parallel of different parts of the same application).

* Correspondence to: Alain.Kerbrat@imag.fr
** Verimag is a joint laboratory of Université Joseph Fourier, CNRS, and INPG.

Moreover, there is one crucial requirement which dominates the area of large scale distributed telecommunications. It is the requirement for *openness*, which means that a telecommunication system can be composed of products designed by different, often competing manufacturers. Openness imposes that the interactions between the products components are based on standard definitions, such as interfaces, services and protocols. These standard definitions should be as much as possible *implementation independent*, to allow for maximum freedom for each manufacturer. Yet it should be possible to derive actual implementations for it, and to check if the chosen implementation conforms to the standard it is derived from.

Setting such standards is the aim of Formal Description Techniques (FDT). An FDT is basically about founding a description language on a suitable mathematical model, to allow a designer to express a design unambiguously and to reason about it. This is the basis for a language whose aim is to allow different designers to give the same meaning to the same system's description. Furthermore, this language should be an *international standard*, recognized as such by an international association such as the ISO or the ITU.

Such Formal Description languages are currently three:

ESTELLE [36]: ESTELLE (extended finite state machine language) is an ISO standard. It is designed for the description of protocols, as a hierarchy of communicating extended state machines.

LOTOS [38]: LOTOS (language of temporal ordering sequences) is a language coming from process algebra theories. Its first application for the design of protocols dates back to 1984, but LOTOS became an ISO standard in 1989. It is based on the algebraic composition of elementary actions, the resulting complex sequential behaviours can be encapsulated into processes. These processes can communicate together through interaction ports. The communication mode is a multi-way rendezvous.

Lotos is based on two sub-languages, one similar to the CCS and CSP process algebras, for the description of the control; the other is the Abstract Data Type (ADT) definition language called Act-One, for the description of data.

Sdl [11]: Sdl (specification and description language) first appeared in a Ccitt recommendation in 1976. Sdl has since evolved from an informal graphical description technique to a full Formal Description Technique, published in the Ccitt recommendation Z-100. Sdl is subject to revision every four years. Object-orientation and other extensions were introduced in 1992.

Sdl is based on communicating extended state machines, communicating via bi-directional links. These links are connected to the state machines via interaction ports. Each state machine owns an input queue, which is common to all of its interaction ports.

A complete FDT is not only a description language, but is usually completed by the following parts:

A design methodology: a design can be obtained by many different, yet equivalent ways. Designers have to make frequent choices during the design process. Making the right choices to obtain the "best" design is a difficult task. Even defining what best design means is already hard. However, to enforce re-usability and communication between different designers, a common design methodology is mandatory. So most FDTs come with their specific design methodology [4, 12].

Tools: tools such as editors are of course necessary for any language. Compilers are usually also needed, however for FDTs, compilation for implementation purposes is not always possible or even desirable. In fact, the characteristic tool for FDTs is for taking advantage of the formal definition of the language; this formal definition allows us to build tools for reasoning about the program's behaviour without having to actually execute it. There are tools for symbolic simulation, invariant analysis, deadlock checking, logical properties verification, etc.

A coherent set of efficient tools is crucial for a real use of FDTs in any development environment. Commercial companies already provide well designed editors, simulators and C code generators for a language such as Sdl. Some similar, but academic works exist for Estelle and Lotos. The simulators of these toolsets use the formal definition of the language to render a correct view of the behaviour of a system in its real environment. Moreover, they allow some more in-depth analysis such as deadlock or assertions checking. However, most of these toolsets actually stop here, and do not offer more advanced analysis functionalities like the comparison of the system under development with a formal definition of its requirements, or with another more abstract description of its behaviour. This is what formal verification is about.

In this article, we present the Aldébaran toolset allowing us to perform such formal analysis activities. Aldébaran is itself part of the Cæsar-Aldébaran Distribution Package (Cadp). Cadp is a toolset developed jointly with the Vasy action of Inria, the Cæsar part being designed for working with the FDT Lotos. Another interconnection of Aldébaran is with the commercial environment *Object*Geode from Verilog. *Object*Geode is an environment for the design of systems with the FDT Sdl. In each case, Aldébaran brings the verification and static analysis functionalities that these two environments lack.

This article is structured as follows: in Sect. 2, we present the theoretical principles Aldébaran is based on. In particular, we present the model checking principle, and what are the critical points for an efficient application of this principle. In Sect. 3, we explain how Aldébaran implements these principles, what are the techniques employed and the performances obtained. One strong point of Aldébaran is its modular architecture, allowing us to efficiently and quickly evaluate new verification algorithms with new modelling techniques.

In Sect. 4, we present two examples of integration of Aldébaran modules for enhancing existing validation activities. Finally, Sect. 5 consists in the description of some significant case studies and of the benefits brought by using Aldébaran for their verification.

2 Principles of model based validation techniques

Model checking [16, 51] consists in building a finite model of the system under analysis and to check the desired requirements on this model. The check itself amounts to a partial or complete exploration of the model (see Fig. 1).

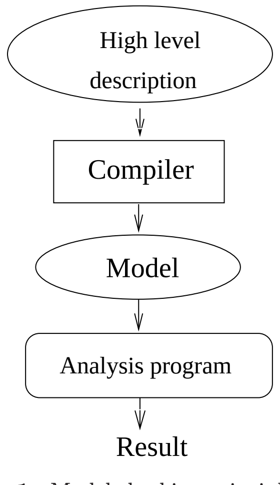

Fig. 1. Model checking principles

The main advantages of this technique are that it can be automated and is fast. Furthermore, model checking allows the easy production of counterexamples, when a requirement is not fulfilled. The main problem of model checking is the potential size of the model, which depends on the system complexity, and which can be huge. This problem is the *state explosion problem.*

Given this model, it is then possible to simulate step by step, or randomly the system. Furthermore, if the model is finite, we can also explore it exhaustively, thus providing the basis for formal verification.

2.1 What is the model?

The model considered in the case of the toolset presented below is a Labelled Transition System (LTS). An LTS is a state graph with anonymous states (no information except a distinguishing number) and transitions labelled with an identification of the actions performed during the states change. The notion of LTS is therefore quite similar to the usual notion of automaton (or state machine). Figure 2 is an example of LTS, with the state 0 being the initial state.

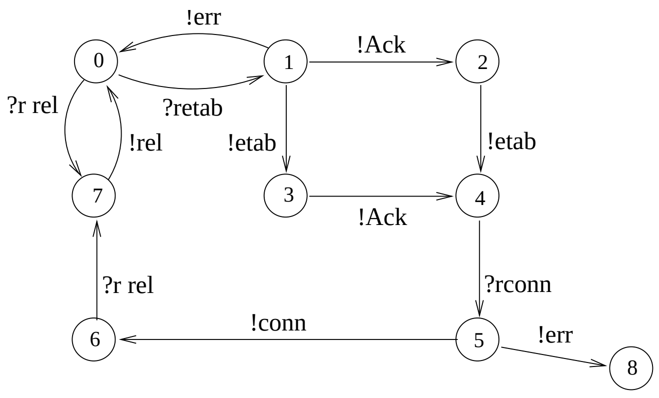

Fig. 2. Example of Labelled Transition System

2.2 How is the model generated?

FDTs usually come with a definition of their operational semantics: the consequences of the execution of any instruction are unambiguously defined by a set of mathematical rules. This set of rules is designed to be complete, coherent, and computable for all the instruction set.

More precisely, these rules rely on a abstract execution model. This execution model is defined as a sequential machine, where one state corresponds to a system's configuration (where is the control, what are the variables values). The execution of any instruction corresponds then to a transition from one state to another. Execution sequences can be given as sequences of transitions.

Given this set of rules, it is possible to derive *statically* and *automatically* any execution sequences; statically means that we do not have to actually execute the system under investigation in its operating environment (which is what testing is about). Automatically means that we can use a computer to derive these execution sequences.

Traditionally, the set of all possible execution sequences is built as an *execution tree*: as systems like communication protocols are designed to run indefinitely, this tree is usually of infinite depth. However, one system's state can occur many times in this tree: it is then possible to fold the tree by merging some of the identical states. The result is then an LTS, which is traditionally called the *model* of the system. The generation steps of this graph are resumed in Fig. 3.

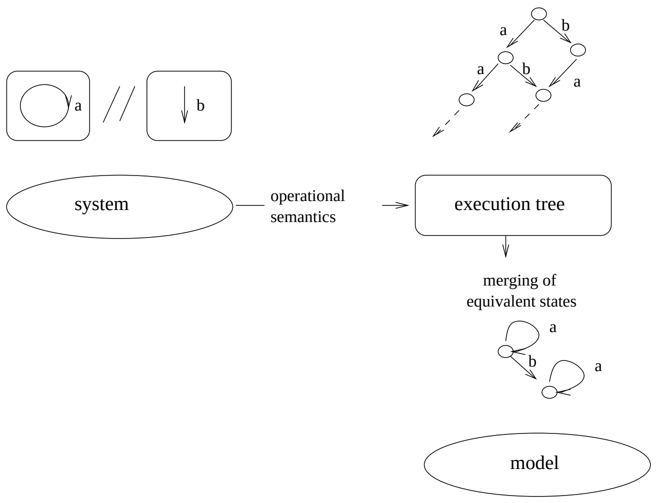

Fig. 3. Generation of the model of a parallel system

In this figure, we consider a first process s_1 able to perform indefinitely the action a, and a second process s_2 able to perform the action b once. s_1 and s_2 work asynchronously (they do not communicate with each other). If we apply usual operational semantics rules on this system description (e.g., those of LOTOS or SDL), then we obtain an infinite execution tree. However, many states of this tree are identical, in the sense that they correspond to the same control states and variables values. Then we can merge these identical states and fold the execution tree into a graph.

If we consider states with a bounded size (no unbounded dynamic creation of processes) and variables with a bounded domain, then the resulting graph (the model) is finite. However, its size is exponential with respect to the number of processes, and therefore usually huge.

2.3 ALDÉBARAN *presentation*

ALDÉBARAN is a formal verification tool. It has been developed for 10 years and integrates state-of-the-art techniques as well as less recent, but intensively tested and applied techniques. It is distributed as a part of the CÆSAR-ALDÉBARAN [19] toolbox.

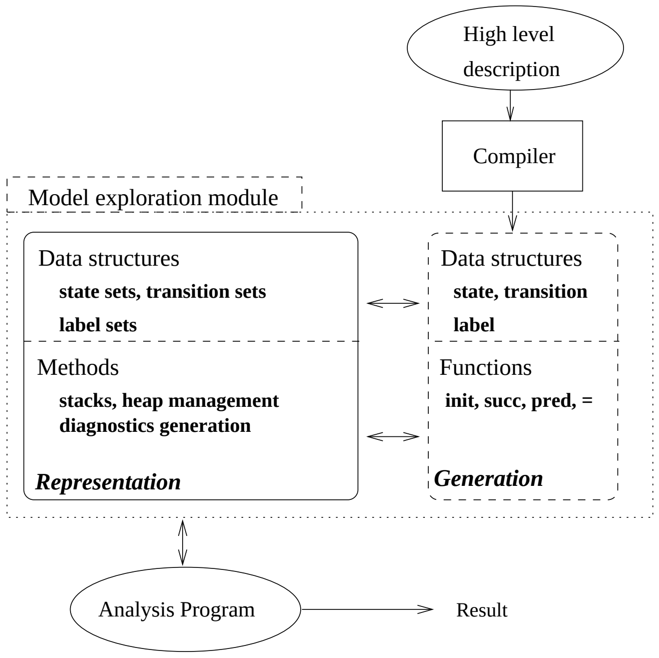

Fig. 4. ALDÉBARAN principles

The architecture of this toolset is centered on the model. More precisely, three main issues are addressed (see Fig. 4):

Model Generation: use or design compilers from high level languages to generate the functions needed for the model generation.

Model Representation: data structures and methods to store and explore efficiently this model.

Analysis Program: use or design algorithms and tools to explore this model for simulation and verification purposes.

ALDÉBARAN is designed with respect to this decomposition. So the *model exploration module* presented in Fig. 4 is a generic box providing what is necessary for the exploration of the model. Once completed by a compiler with the generation functions, this module can be coupled to an analysis program. We detail each of these issues, and present the available analysis programs in the next paragraphs.

2.4 Various techniques for efficient representation

In order to deal efficiently with the state explosion problem, ALDÉBARAN uses two different, but complementary techniques to represent the model.

Enumerative representation: this kind of representation is the most classical one; sets of states or transitions are represented by the complete enumeration of their elements. So the size of the computer representation of these sets is proportional to the number of their elements. However, the performances obtained with this representation, especially in memory terms, depend greatly on the exploration technique used (see Sect. 2.6).

Symbolic representation: in this kind of representation, a set is no longer represented by the enumeration of its elements, but by a formula. Thus, the size of the representation of a set is not proportional to its number of elements. For example, if we consider the set of all even integers between 0 and 10, we can either represent it by the enumeration {0,2,4,6,8,10} or by the formula $\{x \in N \mid \exists k \in N, x = 2k \text{ and } 0 \leq x \leq 10\}$.
So symbolic techniques can lead to a very concise representation of huge (even infinite) sets, and can allow us to perform analysis on models untractable by enumerative methods. On the other hand, symbolic methods involve costly computations and can also explode in memory size depending on the system characteristics.

Enumerative and symbolic representations are complementary. Depending on the system from which the model is built, one of the representation can be more efficient than the other. So it is important to be able to work with both.

2.5 Generation from higher-level languages

ALDÉBARAN is independent of any language, as the model it works with is low level and sufficiently general. However, to ease the description of complex systems, we usually use compilers from high-level languages to produce the functions needed for the model exploration. Generally, the functions `init` (returning the initial state), `succ` (computing the firable transitions from any state) and `=` (comparing two states) are required. Depending on the kind of exploration to perform, it is sometimes necessary to have the function `pred` (computing the transitions leading to a state). We use also the functions `pre` (resp. $\texttt{pre}_a$) computing the transitions (resp. the transitions labelled by a) leading to a set of states. The compiler usually also indicates the exact structure of a state, a transition and a label.

ALDÉBARAN works with several different compilers. The main compilers are CÆSAR which is an efficient compiler of LOTOS description and *Object*GEODE, for dealing with SDL description. Another ad-hoc compiler allows the generation of the model from a set of Extended Labelled Transition Systems (ELTSS).

An ELTS is an LTS extended with some local variables, and whose transitions are decorated by a label, by a guard (allowing the firing of the transition), and by a set of assignments performed on its variables. This kind of ELTS can be viewed as a Guarded Command language, extended with an implicit notion of control variable. Finally, we consider several ELTSs working in parallel. The structure of the composition and the communications between these ELTSs is given as an algebraic composition expression, based either on the binary rendezvous and restriction operators of CSP [32], or the n-ary rendezvous and abstraction operators of CCS [48].

This language is internal to ALDÉBARAN and serves for research and experimentation purposes. It is also used as an intermediate form for transformations from higher level descriptions (see Sect. 4.1).

2.6 Analysis programs

The analysis program is the algorithm which pilots the exploration of the model to check some properties, compute some information or allow the user to simulate the system. It is coupled more or less tightly with the exploration module, depending on the kind of representation chosen. When we leave aside any interaction of the user, the efficiency of the exploration becomes a crucial factor for successful verification. This efficiency depends in part on the representation of the model and the data structures used for its exploration, which is discussed in the previous paragraph. It depends also on the quality of the algorithm and on the optimizations one can bring to the model's size.

When we consider an enumerative representation of the model, we can choose two different approaches:

Working with the explicit representation: this consists in using the exploration module to fully compute the reachable parts of the model and store all the transitions on the way, then to apply the analysis program. In that case, the analysis program can be implemented rather independently of the exploration module, and have its own way of retrieving and storing the model. This is obviously limited to models of "reasonable" size, however it is sometimes necessary for algorithms needing a global knowledge of a model to work on it, like some minimization algorithms.

Working with the implicit representation: this consists in designing analysis algorithms which directly interact with the exploration module, to pilot the exploration of the model according to their own strategies. This allows to implement the so-called "on-the-fly" verification. One of the benefits of this approach is in terms of memory, as it is not usually necessary to record the transitions of the graph. For some algorithms, it is not even necessary to keep an exact idea of the explored states, keeping either partial information (like in Holzmann's bit-state hashing) or only some states (at least the stack in a depth-first search). So the savings in memory can be considerable.

Finally, working with the symbolic representation requires the analysis algorithm to be implemented with respect to the API of the exploration module. The algorithms working with this representation tend to be very different, as they operate directly on sets of states, so in a breadth-first like mode. Algorithms working on enumerative representations work with individual states, generally depth-first.

2.7 Available analysis programs

The available analysis programs of ALDÉBARAN include tools for locating deadlocks, livelocks, or some execution sequences loading to a given state.

On the verification side, ALDÉBARAN capabilities belong to two main categories, behavioural verification and logical verification.

Behavioural verification

Behavioural verification consists in comparing two different descriptions of the behaviour of a system. One should be the system's description, the other is usually a formalization of the system's requirements. It can be a set of execution sequences, MSCs, or another Labelled Transition System. This LTS can itself be produced from another high-level description, possibly more abstract or from another point of view. Two examples of such behavioural requirements are shown in Fig. 5.

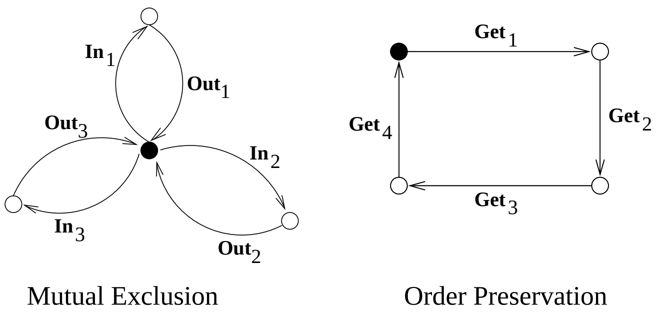

Fig. 5. Examples of behavioural requirements

A crucial point with behavioural verification is the definition of an adequate comparison relation. As we want to compare behaviours, a good candidate for a comparison relation should satisfy most or all of the following criteria:

Preservation of execution sequences: this is the basis of behavioural comparison, it ensures that two LTSs said to be equal represent the same sets of execution sequences (equivalence of language or `trace` equivalence).

Abstraction: we would like to compare two different descriptions of the same system, at different levels of abstraction. This means for example that a given event can be present in the more detailed description, and absent from the more abstract one, or an event of the most abstract one can be refined into a sequence of events in the more detailed description. So the comparison relation should take into account some abstraction/observation criteria to allow the comparison of the descriptions at the same abstraction level. This is usually done by defining a set of *observable* events, and by considering other events as internal, thus anonymous or invisible.

Preservation of the branching structure: two LTSs can represent the same language, yet be different in their structure. This structure reflects the internal choices made in the corresponding system. These internal choices can often influence the interactions one system has with another. So it is important to take them into account when verifying some properties.

Thus, the comparison relation is either a preorder relation, checking the behaviour inclusion, or a equivalence relation, checking the behaviour equality.

The relations considered in ALDÉBARAN belong to the class of the *simulation* and *bisimulation* [50] relations. Simulation relations are preorder relations, whereas bisimulation relations are equivalence relations. These relations respect all the criteria defined above. By variation of the abstraction criteria, and of how internal actions are considered, we obtain a lattice of relations (see Fig. 6)

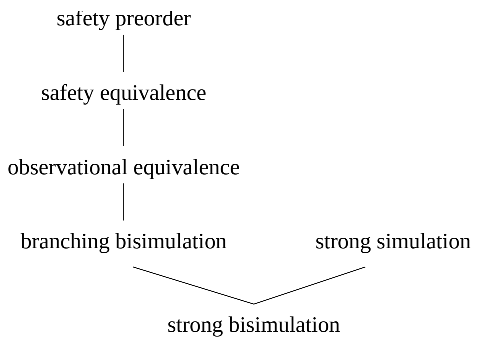

Fig. 6. Lattice of bisimulation relations

from the weakest, *safety equivalence*, to the strongest, *strong bisimulation*, an interesting compromise being the *branching bisimulation*. Strongest here means the relation which distinguishes more LTSs. The strength of one relation is directly related to the kind of properties it preserves. For example, the strong bisimulation preserves both properties (safety or liveness), where the safety equivalence preserves only safety properties. So choosing the right relation depends first on what is to be verified.

Minimization

If we have two LTSs equivalent for one of these bisimulation relations, obviously it is better to work with the smallest one (in terms of number of transitions), as verifying properties on it is equivalent to verifying properties on the other. As these equivalence relations define equivalence classes for LTSs, we would like to be able to pick one of the smallest in this class, and to continue to work with it. Some of the algorithms for behavioural comparison do in fact compute this minimal equivalent LTS. Two of these algorithms [5, 49] are implemented in ALDÉBARAN, and are in effect used also to produce this minimal LTS.

Logical verification

Logical verification consists in checking if the system verifies a property expressed as a *temporal logic formula*. Temporal logics allow us to express overall properties of a program execution, such as liveness, mutual exclusion, fairness, etc. These logics can be interpreted over LTSs: each formula exactly represents a class of LTSs, and the verification problem consists in deciding whether the LTS associated to a given program belongs or not to this class. In particular, whenever the LTS is finite, this kind of verification can be fully automated, which gave rise to the actual *model-checking* activity [16, 51].

Numerous temporal logics have been proposed in the literature [16, 46, 51] to express program properties. Basically, most of them are built upon propositional calculus or first-order logic (interpreted over program states, i.e., variables and control points), extended with a set of temporal operators to reason about program execution. Two family of logics are usually distinguished: *linear-time* logics, expressing program execution as a set of execution sequences; and *branching-time* logics, expressing program execution as an execution tree.

The logic we consider within ALDÉBARAN is the so-called alternation-free μ-calculus [43]. This is a branching-time logic, based upon the propositional calculus with fix-point operators. More formally, its formulae are described by the following grammar:

$$\begin{aligned}\varphi ::= & \; T \mid X \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \\ & <a> \varphi \mid [\, a\,]\, \varphi \mid \mu X.\varphi \mid \nu X.\varphi\end{aligned}$$

The intuitive semantics of these formulae is defined on the state-space of an LTS S as follows:

– T (true) is true in any state of S
– $<a>\ \varphi$ is true in state p if there exists an a-transition from p leading to a state satisfying φ
– $[\ a\ ]\varphi$ is true in state p if each a-transition from p leads to a state satisfying φ
– $\mu X.\varphi$ and $\nu X.\varphi$ denote the usual greatest fix-point and least fix-point operators (where X is a free variable of φ representing a set of states of S)
– $\neg$, $\vee$ and $\wedge$ denote the usual boolean operators: negation, disjunction, and conjunction

From this low-level specification formalism, several CTL-like macros operators are proposed. These macros allow us to express many usual program requirements, such as: "there is no deadlock", "any a action is eventually followed by a b action", "It is not possible to perform an a action followed by a c action without performing a b action between them", etc.

3 Implementation of these techniques within ALDÉBARAN

We present in this section how the principles described in the previous section are implemented in ALDÉBARAN. In particular, we take the point of view of the model's representation, and present for each different type of representation what technique is used and what kind of analysis is possible.

3.1 *Explicit* ALDÉBARAN

Historically the first model-checking tools worked according to Fig. 1:

- First, a compiler is used to produce *explicitly* the model
- Then, the model-checking tool gets back this model and builds its own explicit representation
- Finally, analysis is performed on this internal representation

Obviously, such an approach suffers directly from the state explosion problem, but it allows us to apply nearly all of the analysis programs that are interesting for verification.

With explicit representation, ALDÉBARAN allows us to minimize an LTS or to compare two LTSS *efficiently*, using various equivalence or preorder relations. The key to this efficiency is the implementation of an algorithm, which is an adapted version of the Paige and Tarjan algorithm [21, 49]. Moreover, as an LTS is represented explicitly by its transition relation (a transition is no more than a number, a label, and a number coding respectively the source state, the identification of the transition action, the target state), the tool may be interfaced very quickly with compilers translating a high level description to the LTS, e.g., compilers for LOTOS, SDL.

3.1.1 The Algorithm Principle

Intuitively, *Strong bisimulation* puts together in a single class states which have the same behaviour in terms of elementary steps, i.e., a transition. This means that two states are in the same class if they reach the same classes via a transition. The algorithm solves the *relational coarsest partition problem* which is an instance of the partition refinement paradigm. A *partition* P of a set S is a set of pairwise disjoint subsets of S whose union is all of S. In the context of formal verification, the relational coarsest partition problem may be expressed as follows: *given an* LTS *and an initial partition P_{init} of the set of states, find the coarsest refinement P of P_{init} such that P is* compatible with the transition relation. The last property is another characterization of bisimulation relations.

The algorithm uses a primitive refinement operation that generalizes the one used in Hopcroft's algorithm: if C and C' are two classes of the current partition, then, using the function `pre`, C is split into two subsets: one whose states have a successor in C' and the other whose states have no successor in C'.

A naive implementation of this idea leads to an algorithm proportional in the size of the product between the number of states and the size of the transition relation. Paige and Tarjan proposed an optimization which keeps track, for each state, of the number of successors in a reachable class.

3.1.2 Explicit model minimization

For bisimulation equivalence, the LTS is preprocessed, following the abstract criterion which parametrizes the equivalence relation, and then minimized using the Paige and Tarjan algorithm with the universal partition as initial partition. For example, considering the observational equivalence, the preprocessing consists in:

- Detecting maximal strongly connected component of the relation labelled with the internal action τ
- Computing the transitive closure of the transition relation labelled with τ
- Minimizing the result using the strong bisimulation

3.1.3 Explicit model comparison

Given two LTSS, each of them is minimized following the equivalence relation under consideration, and the LTS defined as the union of the two resulting LTSS is minimized. If the two initial states are in the same class, then the two LTSS are equivalent.

3.1.4 Performance

Let m, n and c be respectively the number of states, transitions and the maximal number of successors by an action. The theoretical complexity of the partition algorithm, implemented in ALDÉBARAN is in $O(c * m \log n)$. This allow the minimization of LTSs of about a few thousands of states and a few millions of transitions on a SPARC 20 with 128 MB of memory.

This point is particularly interesting using a compiler for high level language. For example, strong bisimulation usually reduces the size of *Object*GEODE LTS by 2 to 10 factor, with very good performances: for example, the LTS of a satellite control protocol was reduced from 147 007 states and 555 877 transitions to 66 695 states and 254 030 transitions in 6 min on a SPARC 20 with 128 MB of memory.

Minimization is useful especially for two purposes, first the possibility of visualizing the resulting minimal model, second the possibility of speeding up the verification process.

Visualization of the minimal model
During the design of a distributed system, it becomes rapidly difficult to understand how the different interactions occur. Trying to understand the behaviour of the system by interactive simulation does not always help. Too often the key events are bogged down in too many insignificant events. However, once the LTS has been minimized, it is possible to draw it to grasp its structure, and sometimes discover and understand some behaviours which were hidden in the description's complexity. It is particularly effective when choosing a handful (usually 2

or 3) of key events and hiding everything else. The resulting minimized model is usually very small, yet often more complex than one expects. The Cadp toolbox integrates a tool for the automatic drawing and interactive edition of LTS. It allows us to picture rapidly and easily the structure of these small LTSs (at most 10 to 20 states). Even if this LTS remains too big to be drawn, the Cæsar-Aldébaran simulator still allows us to explore it interactively. So, combining the abstraction, minimization and visualization allows us to do what is sometimes called "visual verification", that is verify "on sight" on a small enough model that a property is correct.

Speeding-up the verification process
Another interest of minimization is purely a performance aspect. Usually, the verification process includes the checking of many properties, so it involves exploration of many models. Some of the properties will need only a simple exploration (e.g., for deadlocks), others the checks of liveness conditions (e.g., for livelocks), some others the use of elaborate algorithms involving the computation of bisimulations. So if before performing all these checks, it is possible to generate a minimized model with respect to a suitable equivalence, the whole verification process will be sped up by at least the reduction factor (and usually much more, as many of the verification algorithms involved are not linear).

3.2 Implicit Aldébaran

As already mentioned in Sect. 2, one of the main motivations for using an implicit model representation is to partially avoid the state explosion problem occurring when using an explicit one. However, to benefit from this advantage, program analysis has to be performed *on-the-fly*, which means that the corresponding algorithms are based on a forward traversal of the underlying LTS.

We present here the common interface shared by the components of Cæsar-Aldébaran to access an implicit representation of an LTS, and then the main verification capabilities offered by this toolset on such a representation.

3.2.1 A common interface to handle implicit LTSs

The purpose of this interface is to provide the verification algorithms with a unified access to an implicit LTS representation, independently of the compiler used to produce this LTS from a high-level source program. However, as a matter of fact, the internal architecture of the interface depends on this compiler, as described below and illustrated in Fig. 7.

Lotos *programs and* ELTSs *composition expressions*
Originally designed by H. Garavel to provide on-the-fly verification facilities for Lotos programs, the Open-Cæsar environment [26]) implements the Model Exploration Module presented in Sect. 2 as follows:

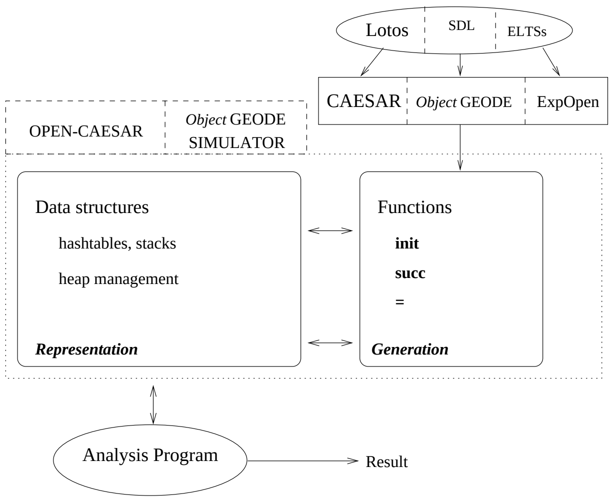

Fig. 7. A common interface to handle implicit LTSs

- Model generation is implemented by a *graph module*, namely a set of *C* data types to represent the states and labels of the LTS, and a set of *C* functions for computing on-the-fly its transition relation (functions `init` and `succ`, delivering respectively the initial state and the firable transitions from any given state). Of course, although the interfaces of this module can be "standardized", its body depends on the source program under consideration. Therefore, it has to be automatically generated from this source program.
- Model representation is implemented by a C library, the *storage module*, providing data structures and associated primitives to efficiently store the part of the state space of the LTS that has to be memorized during its exploration. Data structures currently available are the following: a state table with hash access to perform exhaustive simulation, a state queue and a state stack to manage breadth-first and depth-first exploration, a bitmap table to implement Holzmann's bit hashing technique [33, 34], etc. As it is based on an abstract representation of states and labels, the storage module is independent of the program under verification.

Practically, verification tools are built upon this environment as follows:

The kernel of the verification algorithm is provided as a C program, the so-called *analysis program.* This kernel is quite independent of the program under verification and accesses it only through the graph module interface. Depending on its storage policy, it can also use some of the data structures available within the storage module. A graph module is then generated from the source program under verification and the link edition of these three modules produces an analysis tool dedicated to this particular source program. Examples of such analysis tools include interactive or exhaustive simulation, deadlock detection, execution sequence search, etc.

More recently, the following developments have been performed to extend this initial environment:

- A graph module generator for ELTSs composition expressions
- Two analysis programs, respectively dedicated to behavioural and logical verification, and described in Sects. 3.2.2 and 3.2.3.

SDL *programs*

The successful results obtained with on-the-fly verification of LOTOS programs led us to extend implicit LTS representation within ALDÉBARAN to the SDL FDT. To this purpose, a similar environment of OPEN-CÆSAR has been designed upon the *Object*GEODE compiler [39]. More precisely, this environment is implemented through an Application Programming Interface (API) to the *Object*GEODE simulator, graciously provided by VERILOG.

This API is written in C and it makes available the functions used by the simulator to explore the underlying LTS (`init` and `succ`), as well as data structures for state storage (i.e., stacks, bitmap tables, etc.). Moreover, it also gives access to the scenario generation functions, thus diagnostic sequences computed by ALDÉBARAN can be played back by the simulator (see Sect. 3.2.4).

3.2.2 Behavioural verification using implicit LTSs

Verifying a behavioural specification consists in comparing two LTSs with respect to an equivalence or a preorder relation. To this purpose an efficient algorithm has been presented in Sect. 3.1.4, able to deal with explicit LTSs.

Unfortunately this algorithm is of no interest when one of these LTSs has to be accessed only through an implicit representation. Indeed, it is based on partition refinements of the state space of the two LTSs, which requires a global knowledge of their transition relations. This is therefore not compatible with an implicit representation, since a pre-computation of the transition relation would lose the memory gain induced by such a representation.

In this context, a new algorithm has been proposed for comparing two LTSs with respect to a simulation or a bisimulation relation [24]. This algorithm is based on a traversal of the LTSs, thus allowing on-the-fly behavioural verification from an implicit representation. More precisely, it relies on the fact that the existence of a bisimulation relation between two LTSs can be characterized by a criterion on the execution sequences of a synchronous product of these two LTSs. This criterion can be checked by exhaustive enumeration of these execution sequences, and therefore carried out during a depth-first exploration of the synchronous product, without requiring us to store the transition relations of the two LTSs (only their state space has to be stored). When one of the two LTSs under comparison is deterministic (i.e., when its labelled transition relation is in fact a function), then this check can be reduced to a simple reachability problem on the synchronous product. This happens to be often the case in practice, since the LTS describing the property to be verified is usually deterministic.

This algorithm has been implemented for several simulation and bisimulation relations, the most interesting of which in practice are strong (bi)simulation, branching bisimulation, safety equivalence, and safety preorder.

The worst case time complexity of this algorithm is in $O(m_1.n_1.m_2.n_2)$ in the general case (resp. $O(m_1.m_2)$ when one LTS is deterministic) where n_1, n_2 and m_1, m_2 denote respectively the number of states and transitions of each LTS. However, it appears in practice that this theoretical complexity is far from being reached, and that the comparison times are close to the ones obtained on explicit LTSs with the Paige & Tarjan algorithm. Moreover, as transition relations are never stored, this algorithm could be applied to LTSs with a large size transition relation (a few millions of transitions), that could not be handled using an explicit representation. Finally, thanks to the on-the-fly approach, this algorithm is particularly efficient when the two LTSs are not related (i.e., when the behavioural specification under check happens to be false) since in this case only a small part of the state space of their synchronous product has to be explored and memorized.

3.2.3 Logical verification using implicit LTSs

The good results obtained in practice when verifying behavioural specifications on-the-fly naturally led us to investigate how this same kind of algorithm could be extended to the model-checking of logical specifications.

It turned out that *boolean equation systems* (BES, for short), with mixed fix-point equations, are a suitable framework to formalize such algorithms [1, 2, 60]. More precisely, we proposed a general algorithm for computing the solution $\delta(X_{\text{init}})$ of a given BES $\mathcal{E}$, where X_{init} is a distinguished variable of $\mathcal{E}$. This algorithm can be viewed as a generalization of the one described in Sect. 3.2.2: it relies on depth-first traversals of the dependency graph of the BES, starting from variable X_{init}. During these traversals a solution is computed for each variable of $\mathcal{E}$ following a postfixed order.

This general algorithm has been implemented within ALDÉBARAN and applied to the verification of alternation-free μ-calculus formulae on implicit LTSs. The results obtained from this implementation are quite similar to the ones obtained when verifying a behavioural specification: the worst-case time complexity is rarely reached in practice, and, since the transition relation of the LTS is never stored, large size LTSs can be dealt with (a few millions of states and transitions). Moreover, here again this algorithm is particularly efficient when the logical formula under check happens to be false, since only a small part of the LTS has to be computed in such a case.

3.2.4 Diagnostic computation

As shown in this section, performing on-the-fly verification on an implicit LTS is particularly attractive when the specification under check happens to be false. Thus, this kind of representation is very useful in the early stages of the verification process, when the program under check usually still contains several errors. In this situation the verification tools are mainly used for debugging purposes, and therefore they have to produce accurate diagnostic elements.

However, since the verification algorithms are based on a depth-first exploration of the corresponding LTS, whenever an incorrect state is encountered, an execution sequence leading to this state is available in the execution stack of the algorithm. Although not sufficient from theoretical point of view, this diagnostic sequence usually provides enough information to identify the error (possibly by using a simulation tool to replay it).

3.3 Symbolic ALDÉBARAN

Using symbolic techiques for the representation of the model can allow us to push the limits of the state explosion problem. However, this implies the design of special verification algorithms and of encoding functions to obtain the symbolic representation of a model from the system's description.

Binary decision diagrams (BDDs) [9] have proved to be very efficient for representing and manipulating boolean functions symbolically in many application domains. Their success relies on two important properties: they are *canonical* representations and allow *efficient* (graph-based) computations with finite functions.

We have implemented an efficient procedure to build a symbolic model representation using different types of decision diagrams for systems described by communicating ELTSs. We also propose a suitable interface to this representation allowing the rapid development of complex symbolic verification tools.

3.3.1 The Symbolic Model Interface

The Symbolic Model Interface (SMI) is a library which provides for the efficient construction and manipulation of symbolic representations (with decision diagrams) for finite state systems described as networks of communicating ELTSs. The SMI components, illustrated in Fig. 8, are briefly described below.

Decision diagrams module

The *decision diagrams module* provides a uniform framework for the use of DD in symbolic verification. It consists of a set of C++ generic classes handling decision diagrams, variables, variable lists, substitution lists, etc.

These classes can be easily instantiated with any particular DD implementation. We have already done this for

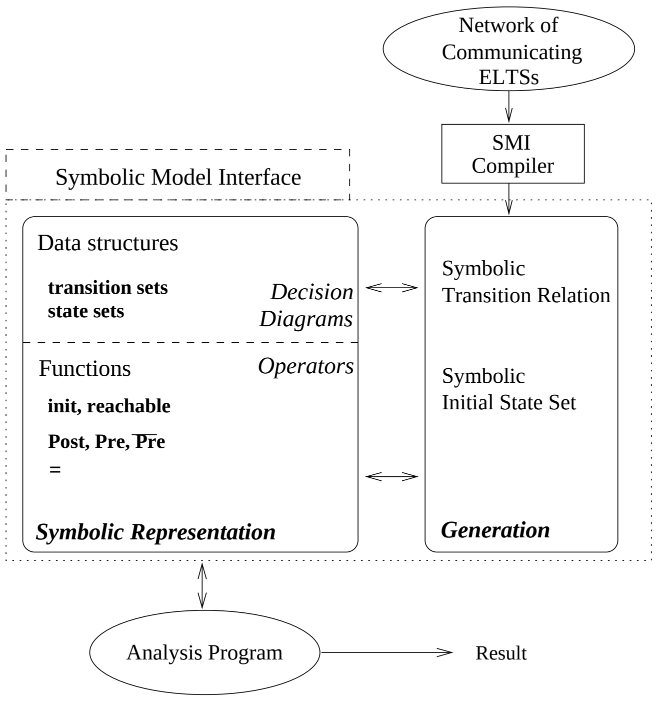

Fig. 8. The SMI architecture

some efficient BDD implementations: TiGeR BDDs [17], Colorado University CUDDs [56], Berkeley BDDs [8] and Verimag BDDs [52]. We have also experimented with a particular version of *multivalued decision diagrams* (MDDs), the MDDs with binary branching, developed at Verimag [6].

Symbolic model generation module

The *generation module* provides functions to build the symbolic model representation. It uses the decision diagram module, i.e., a symbolic representation consisting of a set of decision diagrams which encode the model transitions and the model initial state set.

The generation follows a *compositional* approach. First, a representation is built independently for each ELTS and then all these representations are composed to obtain the whole symbolic representation.

The symbolic model generation is parameterized with various options. We can use two different semantics for ELTSs and composition operators: the first one, with binary rendezvous and restriction operator CSP-like [32], and the second with n-ary rendezvous and abstraction operator CCS-like [48]. We can *a priori* compute the reachable states using different strategies (taking all previously reached states or only the frontier states to compute the next states). This computation can be improved using the simultaneous composition of asynchronous transitions. We allow partitioned transition representations. We can also explicitly specify the DD variables order corresponding to the system variables.

The module consists of a set of C++ classes modeling all the ELTSs concepts: processes, variables, expressions, assignments, transitions, etc. Specific methods to build corresponding decision diagrams are given for any of them. Finally, the whole model is represented as an instance of the *SmiModel* class, whose interface is:

```
class SmiModel {
  ...
public:
  // basic sets
  SmiDDStateSet GetInitialStateSet();
  SmiDDStateSet GetReachableStateSet();
  // successors / predecessors computation
  SmiDDStateSet GetPost(SmiDDStateSet aSet,
                        char* aLabel);
  SmiDDStateSet GetPre(SmiDDStateSet aSet,
                        char* aLabel);
  SmiDDStateSet GetPreTilda(SmiDDStateSet aSet,
                        char* aLabel);
  // transitions
  SmiDDTransition GetTransition(char* aLabel);
  SmiDDTransition GetTauTransition();
  SmiDDTransition GetGlobalTransition();
  // model initialization
  void Initialize(...);
}
```

Symbolic model analysis module
The symbolic model *analysis module* can be considered as the main program. Usually it is written by hand and contains the implementation of the verification algorithm. This module determines how the model is explored, what kind of analysis is performed (forward, backward), which states are stored, etc.

Basic operations on sets, such as union, intersection, or complementation are directly mapped to DDs functions. The inclusion or the equality test are straightforward using DDs. Some specialized functions which perform the model exploration, e.g., to compute the initial/reachable state set (`init`, `reachable`) or the successors/predecessors for a given state set (`post`, `pre`) are also provided.

For example, consider a simple algorithm which computes the reachable states for a given model. The implementation using the SMI library appears as follows:

```
SmiModelManager manager; // the manager
SmiModel* model = NULL; // the model
SmiDDStateSet reach, prev; // two state sets
// create the model from the “example” file
model = manager.CreateModel(“example”);
// build the symbolic representation
model->Initialize();
// the reachable state computation
reach = model->GetInitialStateSet();
do {
    prev = reach;
    // get and store next states
    reach = SmiDDOr(prev, model->GetPost(prev));
} while (prev != reach)
// print the reachable states number
printf(“%lf reachable states”,model->Cardinal(reach));
```

3.3.2 Symbolic analysis tools

Using the SMI library we have implemented two verification algorithms: a μ-calculus model checking algorithm and a minimal model generation algorithm with respect to various equivalence relations. The algorithms' principles and their performance are briefly described in the rest of this section.

μ-calculus model checker
The model checker performs the backward evaluation of μ-calculus formulae over symbolic model representations. The algorithm for a formula φ_0 works in two steps:

– Initially, the set $[[\varphi_0]]$ of model states satisfying the formula φ_0 *is constructed*. All needed operations are straightforward to implement using the SMI functions. The basic boolean expressions are directly evaluated over the system variables and a DD for the satisfying states is obtained. The next state formulae ($<a>\ \varphi$, $[\ a\ ]\varphi$) are evaluated using the primitive `pre`. The fixed point formulae are successively iterated until a stabilized state set is obtained. Finally, any boolean combination of formulae is reduced to the corresponding set operation (complementation, intersection, etc.).
– After this stage, one out of three different *decision procedures* can be invoked. The *standard evaluation* procedure tests if the initial state set `init` is included in $[[\varphi_0]]$. The *forward analysis* procedure checks if some reachable states exist satisfying the formula, if the intersection $[[\varphi_0]]\cap$`reachable` is not empty. If such states exist, a shortest sequence to one of them is also extracted. Finally, the *invariance checking* procedure tests if the formula is satisfied by the initial states and if it is always preserved by one transition step.

Minimal model generator
Classical minimization tools (see Sect. 3.1.4) usually dissociate two tasks for computing the minimal model. One task is to compute the partition refinement which actually corresponds to the minimization, the other is to compute reachability from the initial states. The result is a *minimal* and *reachable* model. The computation of reachability (which amounts to the production of the explicit model as in Sect. 3.1.4) suffers from the state explosion problem. It would be better to be able to compute the partition refinement and reachability at the same time.

This is the aim of the Minimal Model Generation (MMG) algorithm [5]. Given a transition relation and an initial partition, this algorithm allows us to compute the minimal and reachable model up to bisimulation equivalence (currently, strong, weak, and branching bisimulation). As is the case for the Paige and Tarjan algorithm, this same algorithm can be used to compare two models, again without having to compute reachability beforehand.

This algorithm relies on a symbolic representation of the transition relation. It is for example also used for the analysis of timed automata with a symbolic representation based on linear inequalities [57]. We adapted it for a use with decision diagrams in ALDÉBARAN, with interesting results [20].

3.3.3 Performance

These implementations have been successfully tested on several protocols. For example the verification of the mutual exclusion property in a model of the token ring protocol [27] with more than 5×10^8 states takes 1 hour and 16 minutes using the symbolic model checker. The same model can be minimized by the symbolic MMG with respect to the branching bisimulation in less than 10 minutes. Good results can be mentioned also for Fischer's mutual exclusion protocol: in a discrete time version with 12 processes (the model having more than 10^{13} states!) the mutual exclusion property was instantaneously verified (less than 3 seconds). Further results obtained using the symbolic ALDÉBARAN can be found in [7] where an efficient approach for the symbolic verification of asynchronous circuits was proposed.

4 ALDÉBARAN at work

4.1 Compositional generation

One of the possible approaches to overcome the state explosion problem inherent to model-based verification methods relies on the following observation: instead of considering the initial LTS S obtained from the program description, verification can be performed on its quotient S/R, where R is an equivalence relation preserving the properties under check. However, the main difficulty remains to obtain this quotient without having first to explicitly generate the whole LTS S. In particular, a first solution to this problem has already been proposed in Sect. 3.3.2, the *Minimal Model Generation* algorithm, based on a symbolic representation of S.

4.1.1 Compositional LTS generation

We present here an alternative solution, when the program under consideration is described by a composition expression between communicating LTSS. More precisely, provided that R is a congruence with respect to the operators of this expression, the quotient S/R can be obtained following a *compositional approach* [59]: it consists in (repeatedly) generating the LTS S' associated to a given sub-expression, and replacing this sub-expression in the initial one by the quotient S'/R.

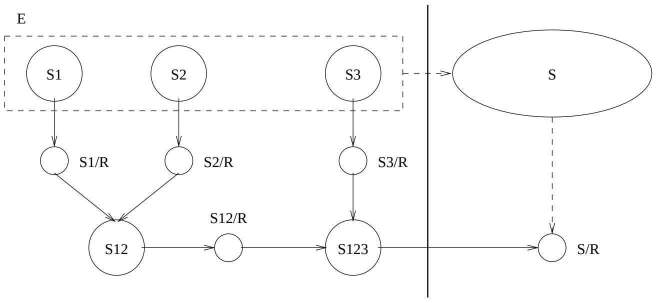

Fig. 9. Compositional generation

Figure 9 illustrates this approach for a composition expression E built from LTSS S_1, S_2 and S_3 (in the left-hand side of the bold line):

1. Each LTS S_i is replaced by its quotient S_i/R
2. S_{12} is generated by composition of S_1/R and S_2/R, then minimized into its quotient S_{12}/R
3. S_{12}/R is then composed with $S3/R$, leading to LTS S_{123}
4. Finally, S_{123} is minimized itself to produce the quotient S/R associated to E.

Thus, the whole LTS S never has to be generated.

Unfortunately, this straightforward technique is not always so appealing in practice. In particular, intermediate LTSS (like S_{12} in the above example) may often contain lots of unnecessary execution sequences, forbidden by the synchronizations expected by its *environment* (the rest of the composition expression). In the worst cases, the size of these LTSS may even exceed that of S, leading to a failure of this approach.

4.1.2 Reducing the size of intermediate LTSS

An appealing solution was proposed by [29, 31] and [14, 15] to reduce the size of the intermediate LTSS produced during a compositional generation. Intuitively, it consists in expressing the environment of a sub-expression as an *interface*, i.e., an LTS representing a set of "authorized" execution sequences that can be performed by this sub-expression. Thus, using a *projection* operator, only a restricted LTS associated to a sub-expression is generated, in which useless execution sequences have been cut off according to its corresponding interface.

These results obtained by [31] and [14] lead us to generalize this approach to the composition expressions used within CÆSAR-ALDÉBARAN [44]. To this purpose, a suitable projection operator has been defined and implemented upon the implicit LTS interface (see Sect. 3.2.4), thus allowing us to generate on-the-fly the restricted LTS associated to a sub-expression of a composition expression. Two kinds of interface can be handled by this operator:

– "exact" interfaces, that are automatically computed from the environment of the sub-expression

– "user-given" interfaces, that can be supplied by the user when the computed ones are not sufficient (i.e., they do not restrict the sub-expression enough).

Note that in this latter case, the correctness of these interfaces can be automatically checked at the end of the compositional generation process.

Finally, a compositional generation tool has also been implemented within CÆSAR-ALDÉBARAN. This tool takes as inputs a composition expression E (extended with projection operators) between LTSs, and one of the bisimulation relation R accepted by ALDÉBARAN. Then, it automatically generates the LTS quotient S/R associated to E by performing corresponding calls to the components of CÆSAR-ALDÉBARAN.

4.1.3 Practical results

The practical results obtained so far on large size case-studies demonstrated the interest of compositional generation, in particular when symbolic representations are too large to efficiently work on. As an example, compositional generation has been successfully applied to the verification of an atomic multicast protocol (the rel/REL protocol [55]): an LTS quotient of about 1 million states was generated in a few hours on a SUN SS 20 workstation, whereas several days of computation on the same workstation were necessary to produce the symbolic representation of the whole LTS (containing about 200 million states).

4.2 Support for automatic conformance test generation

Another example of application of the ALDÉBARAN modularity is the tool TGV (for Test Generation with Verification technology). TGV is a prototype for the automatic generation of conformance test suites [22, 23].

It is being developed jointly by VERIMAG and the INRIA project Pampa.

4.2.1 Working principles

TGV takes as input the *model* of a formal specification, a *test purpose* and a *test architecture*. TGV outputs test suites either as an LTS or in the standard language for conformance test suites TTCN. The external view of the TGV package is illustrated in Fig. 10. The generation of a test case can be decomposed in several functional parts which are performed by different tools. We present in more detail each of these parts.

Test architecture
The Test Architecture describes how the Implementation Under Test (IUT) is placed in its testing environment and how the tester communicate with the IUT. The communications with the IUT usually go through Points of

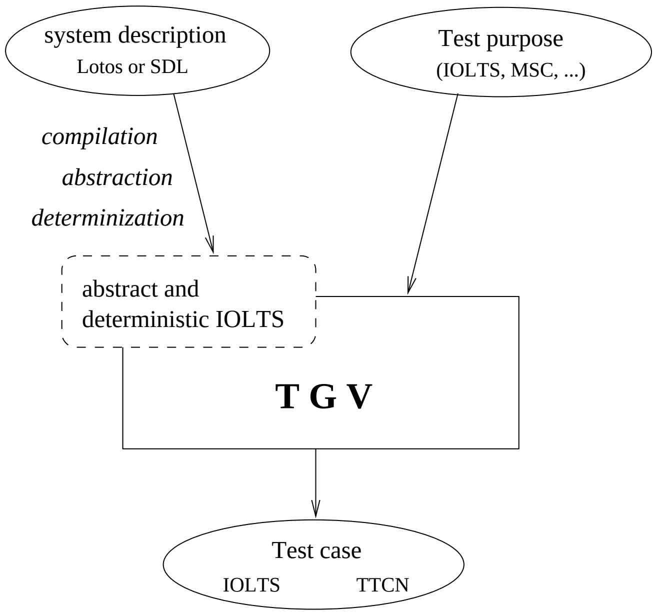

Fig. 10. External view of TGV

Control and Observation(PCO). A test architecture can be quite complex, depending on the tester (which can be itself composed of several coordinated testers) and the PCOs (which can be separated from the IUT by some layers of the testing environment). More details can be found in the ITU-T recommendation ISO 9646 [37].

The model of the implementation
The tester has an external, *black box* view of the IUT. It can only send outputs to and receive inputs from the IUT, it cannot observe any internal actions, much less the state of the IUT. Outputs are *controllable* actions initiated by the tester and sent to the IUT whereas inputs are *observable* actions, initiated by the IUT and received by the tester.

In contrast, a formal specification generally models the *internal view* of the system, i.e., the behaviour of the system with its internal actions and states changes.

To produce a model of the IUT in its testing environment, we replace internal actions by invisible transitions denoted by τ. Then these τs are abstracted away with respect to the weak bisimulation and the result is determined. Other specification transformations are also performed to take care of asynchronous communications between the IUT and the tester.

The result is an Input-Output LTS (IOLTS), abstract (no more reference to internal action) and deterministic.

Test purpose
A test purpose defines a pattern on some particular interactions between the IUT and the tester. This pattern is usually extracted from the system's requirements. So a test purpose describes the desired test cases for testing the conformance of the system with respect to a given requirement.

A test purpose is modeled as a direct acyclic graph with a set of distinguished accepting states (indicating success of the test). In the current version of TGV it is given as an LTS, extended with attributes on states, for distinguishing accepting states.

4.2.2 Test generation process

The current version of TGV works either with the implicit or explicit representation of the model. In the explicit case, we use ALDÉBARAN to produce the abstract and deterministic IOLTS modeling the implementation. In the implicit case, we apply a specially designed algorithm which performs abstraction and determinization on the fly. The rest of TGV processing is done on this explicit or implicit abstract and deterministic IOLTS.

Test graph generation
This is the kernel of the tool. The algorithm is based on a depth-first traversal of a kind of synchronous product between the IOLTS of the model and the LTS representing the test purpose.

During the traversal, two different things are done:

– The algorithm checks that the test purpose is at least feasible.
– Meanwhile, a **skeleton graph** of the test case is synthesized.

Some transitions are decorated with the verdicts (PASS), PASS, FAIL, and INCONCLUSIVE.

Finally, timer management is added to the test case. Timers are used in test cases when a reaction of the implementation is expected but one does not want to wait for an unbounded time because an error that is not observable may have occurred. The difficulty in the management of timers is that concurrency and non-determinism should not be treated in the same way. Timers are managed by TGV in the following way. A timer **tm** is associated to each possible reception of the message **m** by the tester. The timer **tm** is started in the last transition which necessarily precedes the reception of **m**. When the reception of message **m** is expected, the expiration of **tm** may occur. Thus there is always a choice between reception of **m** and reception of **tm** i.e., timeout of **tm**. In each transition sequence, a timer **tm** is cancelled in a transition following its start and as soon as the reception of **m** is no longer possible in the future.

Once completed with timer management, the resulting LTS can be translated into TTCN or be kept in the ALDÉBARAN format for translation in other proprietary test cases formats.

4.2.3 Advantages in using TGV

Formal techniques developed in the area of verification could be useful and profitable for the automatic generation of test suites. The main gains are qualitative and quantitative:

– *Qualitative.* The comparison between test suites automatically generated with hand-written test suites has shown some errors to be detected in the manual test
– *Quantitative.* TGV already provides a productivity (of test cases) improvement of +25 %.

4.3 Conclusion

The first prototype of TGV was developed in a few weeks, by reusing some modules of CADP and by adapting some efficient verification techniques, such as on-the-fly verification, synchronous product of LTSs and behavioural equivalence [23]. The actual version continues to evolve with ALDÉBARAN, in particular for the implicit interconnection with the *Object*GEODE tool. This version should be the basis for the industrial transfer of TGV in the future version of *Object*GEODE.

5 Practical use of ALDÉBARAN

ALDÉBARAN is actually distributed as a part of the CÆSAR-ALDÉBARAN toolset in more than 130 sites. It is used in many places for teaching purposes. ALDÉBARAN is also used in several significant case-studies. We present in this section some of these case studies.

5.1 Telecommunications

Generation of test suites for the DREX *protocol* [22]
The DREX protocol is part of an industrial contract sponsored by the DGA *Direction Générale pour l'Armement* of the French Army. Partners of this contract are CNET (Centre National d'Etude des Télécommunications), Cap Sesa Régions, VERILOG, the Pampa team from IRISA, and VERIMAG.

The goal of this contract was to find out if the automatic generation of test sequences is feasible and profit-earning in industrial contexts. Three tools have been studied and/or developed, TVéda (CNET), Topic (VERILOG) and TGV (IRISA/VERIMAG). In order to compare the methods and the tools, these three tools had to generate test suites, starting from the same specification SDL specification of the DREX protocol and test purposes in natural languages, and to compare the results with handwritten test suites. It appears that finally the consortium agrees on the different components of a realistic test generator, and that TGV represents a good demonstrator of these ideas.

The DREX protocol runs on a network called SOCRATE and connects several MTBX (Telecommunication Means of Air-Bases). Only a subset of the services offered by the DREX protocol has been specified in SDL: priority, roving user, call forwarding, implicit partitioning of users, safety path and user to user signalling. A generic SDL specification of around 2000 lines has been written and instantiated for each service.

Results obtained using ALDÉBARAN *and* TGV
The time needed for the generation of a test case has to be separated into two parts: the time needed for the graph generation with GÉODE which took between 3.5 s and 400 s and the test case generation with TGV which took between 1 s and 2 s.

As we have already mentioned, we have discovered errors in the hand written test suites, and we have proved that automatic generation provides a productivity improvement.

Other recent telecommunication case studies include: feature interactions in telephony systems [42], ISDN User Part protocol [41], SSCOP protocol (ongoing work), VIRES protocol (ongoing work).

5.2 *Hardware protocols*

Verification of the Powerscale bus arbiter protocol
PowerScale is the multiprocessor, PowerPC-based architecture used by BULL in its Escala series of workstations and servers. In this case-study [13], the main components of this architecture (processors, memory controller and bus arbiter) were described by 760 lines of LOTOS.

Results obtained using ALDÉBARAN
This case study is a good illustration of the power of compositional generation. It was not possible to generate the whole model, and on the fly verification techniques failed due to lack of memory. Using compositional generation, it was possible to break the system into three main parts, generate the corresponding LTS, and after minimization of these LTSs, generate a bisimulation-equivalent LTS of the whole system. This resulting LTS was 52 320 states and 176 284 transitions, so it became easy to perform all needed verifications.

5.3 *Embedded systems*

DMS *Design Validation* (DDV)
DDV [3] is a case study sponsored by ESA-ESTEC and developed in collaboration Matra-Marconi Space (MMS) and Dornier. DMS (Data Management System) is the control system of a satellite. Is is responsible for the detection and treatment of failures.

One of the goals of this study was to define a methodological framework for specifying and validating fault tolerant systems. It is based on the combined use of SDL for the specification of the system and of the Fault Detection, Isolation and Recovery (FDIR) methodology.

Results obtained using ALDÉBARAN
For this case study, ALDÉBARAN was used in combination with GÉODE (*Object*GEODE was not available at the time), with only the explicit connection. Two SDL descriptions were produced: one *functional description* written by MMS whose main aim was the verification of the requirements, and one *architectural description* written by DORNIER, whose main aim was code generation. The requirements themselves were established during the Failure Mode Effects and Criticality Analysis (FMECA). The aims of the study was to verify that both descriptions were correct with respect to the requirements. Each SDL specification was about 4000 lines of comment-free SDL-88. We were able to find some errors and test some requirements on the first one. We present more detailed results on the second one, namely the Architectural description with Fault Injection, which proved to be the most difficult to verify.

The complete model was too large to be generated. Instead, a classical partial generation method was applied: using the GÉODE simulator, we produced an execution sequence leading the system into an interesting state for verification. In that case, this sequence corresponded to the firing of all initialization procedures of the system. The model was generated from this state, with the injection of one particular fault. Then the stop conditions of the GÉODE simulator were applied to stop the generation of the model, when the system was able to come back to a stable state (i.e., the fault was treated) or a given depth was reached. The resulting model (depending on the fault) was up to 147 007 states and 555 877 transitions. Using minimization and visualization, we were then able to verify the properties corresponding to the correct treatment of faults.

Other recent embedded systems case studies include: railyard systems [25] and the satellite control system MSG [53].

5.4 *Security protocols*

Verification of the Equicrypt Protocol
The Equicrypt Trusted Third Party protocol is an authentification protocol for the conditional access to multimedia services. It is based on the use of a Trusted Third Party for authentification. The specification [45] consisted in around 2000 lines of LOTOS.

Results obtained with ALDÉBARAN
Model-checking is often viewed as being inadequate for this kind of protocols, due to the relative complexity of data manipulation with respect to the control. However, when applicable, model checking brings up the possibility to generate counter examples to unsatisfied properties. In this case study, this ability was crucial, as it allowed us to produce two possible attacks on the protocol. The production of attacks is generally up to the protocol verifier when using more theorem proving based methods. Finally, the use of efficient minimization algorithms proved essential, as the model was 786 681 states and 4 161 795 transitions and took 20 hours to be generated. It was minimized by ALDÉBARAN for strong bisimulation in 20 minutes using an Ultra-Sparc 2 with 800 MB of RAM.

The resulting model was 69 754 states and 520 633 transitions, which allowed the authors of [45] to perform all needed verifications.

5.5 Network protocols

Some network protocols case studies include a bounded retransmission protocol [47], an Internet transport protocol TCP [54], and some distributed leader election algorithms [27].

6 Conclusion

The ALDÉBARAN toolset we have presented in this article is devoted to the formal verification of distributed systems. More precisely, it consists in an integrated set of tools, closely interconnected, allowing us to address several program validation issues, such as symbolic debugging, formal verification of behaviour requirements, and automatic test case generation.

This toolset relies on the so-called *model-based* approach: from a formal description of the program under consideration a model is generated, then program analysis is performed on this model. Several model representations are available within ALDÉBARAN, each of them offering particular advantages in terms of efficiency, and each of them leading to different kinds of program analysis algorithms.

The ALDÉBARAN toolset has now been developing for 10 years, with an important concern to keep it *open* and *evolutive*, achieved through an architecture based on clear-cut modules. In particular:

– It is open, as it is already connected to two FDT-based development environments : the LOTOS compiler CÆSAR, from the INRIA action VASY, and the commercial SDL environment *Object*GEODE, from the VERILOG company. Conversely, parts of ALDÉBARAN are routinely used in conjunction with other tools, like in the TGV environment presented in Sect. 4.2. For example, it is used to perform time abstraction on timed-automata with the KRONOS tool [57], or to minimize abstract state graphs produced by a theorem prover [30]. Finally, ALDÉBARAN is also open from an internal point of view. For instance, its symbolic LTS representation interface allows us to use several existing DD packages (Sect. 3.3.3).
– It is evolutive, as it allows an easy prototyping of new analysis algorithms or new verification strategies. Moreover, some of its underlying algorithms have been adapted to other contexts: in particular, the on-the-fly algorithm used within ALDÉBARAN for bisimulation checking could be re-used both for the TGV kernel and within an optimizer for synchronous code distribution [10].

The ALDÉBARAN toolset has been already distributed as a part of the CÆSAR-ALDÉBARAN package to more than 130 sites, and thus used in numerous case studies, some of them being of industrial origin. Therefore, thanks to this user feedback, its components achieved a relative robustness.

We conclude by an overview of the development perspectives of ALDÉBARAN, according to the three main issues of model-based verification addressed by this toolset, namely model generation, model representation, and program analysis.

– Regarding model generation, it is clear that the verification capabilities offered by ALDÉBARAN can be applied to any formalism whose (operational) semantics can be expressed in terms of LTSS, provided that there already exists a compiler able to produce this LTS. If such a connection is usually straightforward through an explicit LTS representation, it is more difficult to obtain it through an implicit representation, and much more difficult through a symbolic one (unless the compiler already produces such a representation).
  However, this latter kind of connection can be more easily achieved using a higher level program representation than the LTS, like the communicating ELTSS networks already existing within ALDÉBARAN. To this purpose, we plan to extend this intermediate program representation to other communication mechanisms than rendezvous and shared variables, such as fifo channels. Thus, formalisms like SDL or PROMELA could be translated in such a program representation, making it possible to generate symbolic representations that could be processed by ALDÉBARAN.
– Regarding model representations, most of the perspectives concern the definition of suitable symbolic representation. Indeed, if the BDD have been initially proposed for boolean program representation (and hence hardware verification), they are not necessarily well adapted for representing programs with more general data types, or asynchronous communication modes. Even though many extensions have been already proposed, none of them is quite satisfactory at the moment.
  Furthermore, other kinds of symbolic representations could be also considered when the program under consideration is expressed in terms of communicating ELTSS. In particular, representations based on convex polyhedra have already proved their interest for static analysis purposes [40].
– Finally, many directions remain to be explored regarding model-based analysis algorithms, and, more generally, model-based verification strategies. Here, one of the major concerns is clearly to push back the state explosion problem. Thus, a general approach is to perform the analysis on a “reduced” model, which preserves the properties under verification. At least two

directions could be thoroughly investigated within Aldébaran to compute such a reduced model:

– The compositional strategy presented in Sect. 4.1 could be improved and extended to other communication mechanisms, such as fifo channels. Thus, this strategy could be also applied to the Sdl Fdt.
– The implicit LTS representation available within Aldébaran is well adapted for implementing the model reduction methods based on partial orders [28, 58], or program symmetries [18, 35]. Moreover, these methods would be directly combined with the on-the-fly analysis algorithms already provided by Aldébaran.

Of course, all these perspectives will still have to be continuously validated through the confrontation with industrial case-studies. But reciprocally, it also seems obvious that the diffusion of formal methods within an industrial context necessarily goes through the development of verification toolsets like Aldébaran.

Aldébaran is available free of charge upon request addressed to `Alain.Kerbrat@imag.fr`.

References

1. H.R. Andersen. Model checking and boolean graphs. In: *Proceedings of ESOP'92*, Lecture Notes in Computer Science 582. Springer Verlag, 1992
2. H.R. Andersen, B. Vergauwen. Efficient checking of behavioural relations and modal assertions using fixed-point inversion. In: *Proceedings of CAV'95*, Lecture Notes in Computer Science 939. Berlin Heidelberg New York: Springer-Verlag, 1995
3. S. Ayache, E. Conquet, Ph. Humbert, C. Rodriguez, J. Sifakis, R. Gerlich. Formal methods for the validation of fault tolerance in autonomous spacecraft. In: *Proceedings of the FTCS Symposium*, Sendai, Japan, June 1996
4. Tommaso Bolognesi, Jeroen van de Lagemaat, Chris Vissers. *LOTOSphere: Software Development with LOTOS*. Kluwer Academic, 1995
5. Ahmed Bouajjani, Jean-Claude Fernandez, Nicolas Halbwachs. Minimal model generation. In: R.P. Kurshan, E.M. Clarke, (eds.), *Proceedings of the 2nd Workshop on Computer-Aided Verification, DIMACS Series in Discrete Mathematics and Theoretical Computer Science* vol. 3, pp. 85–92. AMS-ACM, June 1990
6. Dorel Marius Bozga. Vérification formelle de systèmes distribués : diagrammes de décision multivalués. Master's thesis, Université Joseph Fourier, 1996
7. Marius Bozga, Oded Maler, Amir Pnueli, Segio Yovine. Some progress in the symbolic verification of timed automata. In: *Proceedings of the 8th Conference on Computer-Aided Verification*, Haifa, Israel, June 1997. To appear
8. K.S. Brace, R.L. Rudell, R.E. Bryant. Efficient implementation of a bdd package. In: *27th ACM/IEEE Design Automation Conference*, pp. 40–45, 1990
9. Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992
10. P. Caspi, J.-C. Fernandez, A. Girault. An algorithm for reducing binary branchings. In: *FST & TCS*, Lecture Notes in Computer Science 1026. Berlin Heidelberg New York: Springer-Verlag, 1995
11. CCITT. Specification and description language. Technical Report Tome X, ITU, 1989
12. CCITT. Directives relatives à la méthodologie, pour le langage de description et de spécification. Technical Report Rapport Z100 - Annexes I et II, ITU, 1993
13. Ghassan Chehaibar, Hubert Garavel, Laurent Mounier, Nadia Tawbi, Ferruccio Zulian. Specification and verification of the powerscale bus arbitration protocol: An industrial experiment with lotos. In: Reinhard Gotzhein, Jan Bredereke, (eds.), *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96*. IFIP London Chapman & Hall, 1996. Full version available as INRIA Research Report 2958
14. S.C. Cheung, J. Kramer. Enhancing compositional reachability analysis with context constraints. In: *Proceedings of the 1st ACM International Symposium on the Foundations of Software Engineering*, pp. 115–125, Los Angeles, CA, December 1993
15. S.C. Cheung, J. Kramer. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In: *Proceedings of SIGSOFT'95*, 1995
16. E. Clarke, E. A. Emerson, A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. In: *10th Annual Symposium on Principles of Programming Languages*. ACM, 1983
17. Digital Equipment Corporation. *TiGeR Library Manual Reference*, 1994
18. E.A. Emerson, S. Jha, D. Peled. Combining partial order and symmetry reductions. In: Ed Brinksma, (ed.), *Proceedings of TACAS'97 (Tools and Algorithms for the Construction and Analysis of Systems)*, Enschede, The Netherlands, April 1997. Berlin Heidelberg New York: Springer-Verlag
19. J.C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, M. Sighireanu. Cadp: A protocol validation and verification toolbox. In: Rajeev Alur, Thomas A. Henzinger, (eds.), *Proceedings of the 8th Conference on Computer-Aided Verification*, August 1996
20. J.C. Fernandez, A. Kerbrat, L. Mounier. Symbolic equivalence checking. In: C. Courcoubetis, (ed.), *Proceedings of the 5th Workshop on Computer-Aided Verification (Heraklion, Greece)*, Lecture Notes in Computer Science 697. Berlin Heidelberg New York: Springer-Verlag, 1993
21. Jean-Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2–3): pp. 219–236, May 1990
22. Jean-Claude Fernandez, Claude Jard, Thierry Jéron, Laurence Nedelka, César Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programing*, 1996. Special issue on Industrially Relevant Applications of Formal Analysis Techniques. Also available as INRIA Research Report RR-2923
23. Jean-Claude Fernandez, Claude Jard, Thierry Jéron, Laurence Nedelka, César Viho. Using on-the-fly verification techniques for the generation of test suites. In: R. Alur, T. A. Henzinger, (eds.), *Proceedings of the 8th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 1102. Berlin Heidelberg New York: Springer-Verlag, 1996. Also available as INRIA Research Report RR-2987
24. Jean-Claude Fernandez, Laurent Mounier. "On the fly" verification of behavioural equivalences and preorders. In: K. G. Larsen, (ed.), *Proceedings of the 3rd Workshop on Computer-Aided Verification*, July 1991
25. Lars-åke Fredlund, Fredrik Orava. An experiment in formalizing and analysing railyard configurations. In: Z. Brezočnik, T. Kapus, (eds.), *Proceedings of COST 247 International Workshop on Applied Formal Methods in System Design*. University of Maribor, Slovenia, June 1996
26. Hubert Garavel. The open/cæsar reference manual. Rapport SPECTRE C33, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, May 1992
27. Hubert Garavel, Laurent Mounier. Specification and verification of various distributed leader election algorithms for unidirectional ring networks. *Science of Computer Programming*, 1996. Special issue on Industrially Relevant Applications of Formal Analysis Techniques. Full version available as INRIA Research Report 2986
28. Patrice Godefroid. Using partial orders to improve automatic verification methods. In: R. P. Kurshan, E. M. Clarke, (eds.),

Proceedings of the 2nd Workshop on Computer-Aided Verification,DIMACS Series in Discrete Mathematics and Theoretical Computer Science volume 3, pp. 321–340. AMS-ACM, June 1990

29. S. Graf, G. Lüttgen, B. Steffen. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computation*, 3, 1996. Appeared as Passauer Informatik Bericht MIP-9505
30. S. Graf, H. Saidi. Construction of abstract state graphs with pvs. In: *Conference on Computer Aided Verification CAV'97, Haifa*, Lecture Notes in Computer Science, June 1997
31. Susanne Graf, Bernhard Steffen. Compositional minimization of finite state processes. In: R.P. Kurshan, E.M. Clarke (eds.) *Workshop on Computer-Aided Verification*, Rutgers, USA, June 1990. DIMACS
32. C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978
33. Gerard J. Holzmann. Algorithms for automated protocol validation. In: Joseph Sifakis, (ed.), *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems*, June 1989
34. Gerard J. Holzmann. An analysis of bitstate hashing. In: Piotr Dembinski, Marek Sredniawa, (eds.), *Proceedings of the 15th IFIP International Workshop on Protocol Specification, Testing and Verification*. IFIP London, Chapman & Hall, June 1995
35. C.W. Ip, D. Dill. Better verification through symmetry. In: L. Claesen, (ed.), *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, April 1993
36. ISO. Estelle: a formal description technique based on an extended transition model. Technical Report TC97-SC21, ISO, 1985
37. OSI-Open Systems Interconnection, Information Technology – Open Systems Interconnection Conformance Testing Methodology and Framework Part 1: General Concept – part 2: Abstract Test Suite Specification – part 3 : The Tree and Tabular Combined Notation (TTCN). *International Standard ISO/IEC 9646-1/2/3*, 1992
38. 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
39. A. Kerbrat, C. Rodriguez, Y. Lejeune. Interconnecting the *Object*GEODE and CÆSAR-ALDÉBARAN toolsets. In: *To be published in the proceedings of SDL forum'97*. Elsevier Science (North-Holland), 1997
40. Alain Kerbrat. Reachable state space analysis of lotos programs. In: Dieter Hogrefe and Stefan Leue, (eds.), *Proceedings of the 7th International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols FORTE'94*, October 1994
41. Alain Kerbrat, Dave Penkler, Nicolas Raguideau. Using aldébaran and object-géode for the development of telecommunications services. Technical report, Hewlett Packard / Verimag, 1996
42. Henri Korver. Detecting feature interactions with CÆSAR/ ALDEBARAN. *Science of Computer Programming*, 1996. Special issue on Industrially Relevant Applications of Formal Analysis Techniques. To appear
43. D. Kozen. Results on the propositional μ-calculus. In: *Theoretical Computer Science*. North-Holland, 1983
44. Jean-Pierre Krimm, Laurent Mounier. Compositional state space generation from lotos programs. In: Ed Brinksma, (ed.), *Proceedings of TACAS'97 (Tools and Algorithms for the Construction and Analysis of Systems)*, Enschede, The Netherlands, April 1997. Berlin Heidelberg New York: Springer-Verlag. Extended version with proofs available as Research Report VERIMAG RR97-01
45. G. Leduc, O. Bonaventure, E. Koerner, L. Léonard, C. Pecheur, D. Zanetti. Specification and verification of a ttp protocol for the conditional access to services. In: *Proceedings of the 12th J. Cartier Workshop on Formal Methods and their Applications: Telecommunications, VLSI and Real-Time Computerized Control System*, Montreal, Canada, October 1996
46. Z. Manna, A. Pnueli. Verification of concurrent programs: the temporal framework. In: *The Correctness Problem in Computer Science*, London, 1982. International Lecture Series in Computer Science, Academic Press
47. R. Mateescu. Formal description and analysis of a bounded retransmission protocol. In: Z. Brezočnik, T. Kapus, (eds.), *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design*. University of Maribor, Slovenia, June 1996. Also available as INRIA Research Report RR-2965
48. Robin Milner. *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92. Berlin Heidelberg New York: Springer-Verlag, 1980
49. Robert Paige, Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6): pp. 973–989, December 1987
50. David Park. Concurrency and automata on infinite sequences. In: Peter Deussen, (ed.), *Theoretical Computer Science*, Lecture Notes in Computer Science 104, pp. 167–183. Berlin Heidelberg New York: Springer-Verlag, March 1981
51. Jean-Pierre Queille, Joseph Sifakis. Fairness and related properties in transition systems — a temporal logic to deal with fairness. *Acta Informatica*, 19: pp. 195–220, 1983
52. C. Ratel. *Définition et Réalisation d'un Outil de Vérification Formelle de Programmes LUSTRE: le Système LESAR*. PhD thesis, Université Joseph Fourier, Grenoble, 1992
53. Carlos Rodriguez. Modelling and verification of the asw for msg. Spectre report 97-04, VERIMAG, March 1997
54. Ina Schieferdecker. Abruptly-terminated connections in tcp – a verification example. In: Z. Brezočnik, T. Kapus, (eds.), *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design*. University of Maribor, Slovenia, June 1996
55. Santosh K. Shrivastava, Paul.D. Ezhilchelvan. rel/rel: A family of reliable multicast protocol for high-speed networks. Technical report, University of Newcastle, Dept. of Computer Science, UK, 1990
56. F. Somenzi. *Cudd Decision Diagram Package*. Release 1.0.6. Colorado University, 1995
57. S. Tripakis, S. Yovine. Analysis of timed systems based on time-abstracting bisimulations. In: *Proc. 8th Conference Computer-Aided Verification, CAV'96*, pp. 232–243, Rutgers, NJ, July 1996. Lecture Notes in Computer Science 1102, Berlin Heidelberg New York: Springer-Verlag
58. A. Valmari. A stubborn attack on state explosion. In: R. P. Kurshan, E. M. Clarke, (eds.), *Proceedings of the 2nd Workshop on Computer-Aided Verification, DIMACS Series in Discrete Mathematics and Theoretical Computer Science* volume 3, pp. 25–42. AMS-ACM, June 1990
59. Antti Valmari. *Compositionality in State Space Verification*, Lecture Notes in Computer Science 1091, pp. 29–56. Berlin Heidelberg New York: Springer-Verlag, June 1996
60. B. Vergauwen, J. Wauman, J. Levi. Efficient fixpoint computation. In: *Static Analysis Symposium (SAS'94)*, Lecture Notes in Computer Science 864. Berlin Heidelberg New York: Springer-Verlag, 1994