OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing

Hubert Garavel

INRIA Rhône-Alpes and DYADE / VASY group
655, avenue de l'Europe
38330 Montbonnot St Martin
France
Tel: +(33) 4 76 61 52 24   Fax: +(33) 4 76 61 52 52
E-mail: hubert.garavel@inria.fr
Web: http://www.inrialpes.fr/vasy

Abstract. This paper presents the OPEN/CÆSAR software architecture, which allows to integrate in a common framework different languages/formalisms for the description of concurrent systems, as well as tools with various functionalities, such as random execution, interactive simulation, on-the-fly and exhaustive verification, test generation, etc.. These principles have been fully implemented, leading to an open, extensible, and well-documented programming environment, which allows tools to be developed in a modular framework, independently from any particular description language.

Introduction

Research in the area of tools and algorithms for the construction and analysis of systems has been and remains particularly active. Despite this intense activity, end-users involved in actual system design do not always receive as much computer-aided assistance as they could expect. Among the many tools developed, only a few are robust enough to be applied to real-life problems. Moreover, in many cases, end-users cannot benefit from all these tools, because they are using a different language than the one supported by the tools.

It seems therefore that a significant part of the effort spent in developing tools is wasted, and that the global productivity of the research community in formal methods and verification could be increased through a better coordination.

A large part of these problems could probably be solved if the whole community and industry adopted a unique language for the specification and design of protocols and concurrent systems. However, this is not the case: even the standardization efforts undertaken within ISO and ITU-T led to three different standards (ESTELLE, LOTOS, and SDL), in addition to the many other formalisms that already exist: CCS, CSP, μCRL, PROMELA, etc. It seems therefore clear that different specification languages will continue to exist (as it is already the case for sequential programming languages); even if the Darwinian selection process exists, it is unlikely that one single language will emerge and remain.

Taking as a fact the coexistence of multiple languages, this paper attempts to lower the corresponding economic cost. It describes a generic architecture allowing the development of tools (e.g., simulation, verification, test generation tools, etc.) that can be applied to programs written in different languages (e.g., LOTOS, SDL, etc.). These ideas have been entirely implemented in a tool environment named OPEN/CÆSAR, which has been used for realistic case-studies [4, 7, 11, 19, 21, 22], some of them in an industrial context.

The work on OPEN/CÆSAR was initiated in 1992 as a follow-up to the design of the CÆSAR compiler [12], a model-checking tool for translating LOTOS programs into Labelled Transition Systems (LTSs or *graphs*, for short), a semantic model on which verification can been performed using behavioural equivalences and/or temporal logic formulas. Because the only functionality provided by CÆSAR was graph generation, and due to state explosion, its applicability was restricted to small-size systems.

The initial goal of the OPEN/CÆSAR project was to extend CÆSAR with additional functionalities (including random execution, interactive simulation, and partial, on-the-fly verification) needed to deal with larger systems. It is worth noticing that this goal was a radical departure from previous approaches, consisting either in:

- Providing an environment dedicated to a given language, by juxtaposition of separate tools, each tool providing a distinct functionality: graphical or syntax-driven editor, code generator, interactive simulator, debugger, on-the-fly property checker, test generator, etc. A certain degree of unification between these tools was often achieved by sharing a compiler front-end, using a common format for abstract syntax trees, and exchanging information via defined interfaces (e.g., files). However, the semantical processing parts remained duplicated between the different tools, possibly limiting interoperability, as each tool could have its own restrictions (accepting only a particular subset of the source language) or give a different interpretation of the source language semantics.

  On the contrary, the OPEN/CÆSAR project targeted at the greatest possible integration between the different functionalities by sharing, not only the compiler front-end, but also all semantic processings, the choice between simulation, verification, etc. being deferred as much as possible.
- Adapting an existing code generator or simulator in order to perform verification. In most cases, this approach faced architectural or performance issues: experience proved that it was very difficult to turn a simulation tool into an efficient verification tool, unless the simulator had been intentionally designed for this purpose from the beginning.

  On the contrary, the OPEN/CÆSAR project had to adapt the model-checking verification capabilities of CÆSAR to simulation and code generation. Not so surprisingly, we found out that going this direction was much easier than going the opposite way, exactly like turning a multi-user operating system into a single-user operating system is much easier than the opposite.

After completion of its initial goals, the aims of the OPEN/CÆSAR project were reviewed and extended toward a new target: the architecture was modified so that other languages/formalisms than LOTOS could be integrated into OPEN/CÆSAR.

This paper describes the technical solutions and achievements of the OPEN/CÆSAR project. It is organized as follows: Section 1 presents the principles of the OPEN/CÆSAR architecture, which is based upon a functional decomposition in three modules: the *graph module*, the *library module*, and the *exploration module*. These modules are described in Sections 2, 3, and 4 respectively. Finally, the conclusion summarizes the benefits of the OPEN/CÆSAR approach and discusses its limitations, leaving room for future research.

1 Architecture

The design of the OPEN/CÆSAR architecture takes its roots in the development of CÆSAR [12], the first model-checking tool for full LOTOS. It was also made possible by the author's prior experience in designing the architecture of VESAR [1], a protocol engineering tool for the ESTELLE language, probably the first commercial tool to integrate simulation, on-the-fly analysis and model-checking capabilities. OPEN/CÆSAR also benefited from ideas implemented in other verification tools, especially XESAR [13] and SPIN [14].

Although these tools support different source languages (LOTOS, ESTELLE, and PROMELA), they offer similar functionalities, among which verification by reachability analysis. The basic idea of the OPEN/CÆSAR architecture was to identify the common functionalities shared by these tools and to organize them into three distinct modules. The OPEN/CÆSAR architecture improves previous tools by enforcing a clear separation between these modules using well-defined APIS (Application Programming Interfaces). The OPEN/CÆSAR architecture is depicted on Figure 1.

The graph module is responsible for encapsulating and hiding all language-dependent aspects. From the outside of this module, the source program can be seen only as an LTS, whatever the source language used. The graph module exports a representation for the states and the labels of the transition system, as well as primitives to handle states and labels. It also provides primitives to compute the transition relation (i.e., the initial state and the successors of a given state). These features are accessed through an API named "`caesar_graph.h`", which does not depend on any particular source program, nor any particular language.

The mapping between a given source program and this interface is achieved using an OPEN/CÆSAR-*compliant compiler*, which translates the source program into a C program implementing this interface. This C program is compiled separately and linked with the other OPEN/CÆSAR modules. In this approach, the

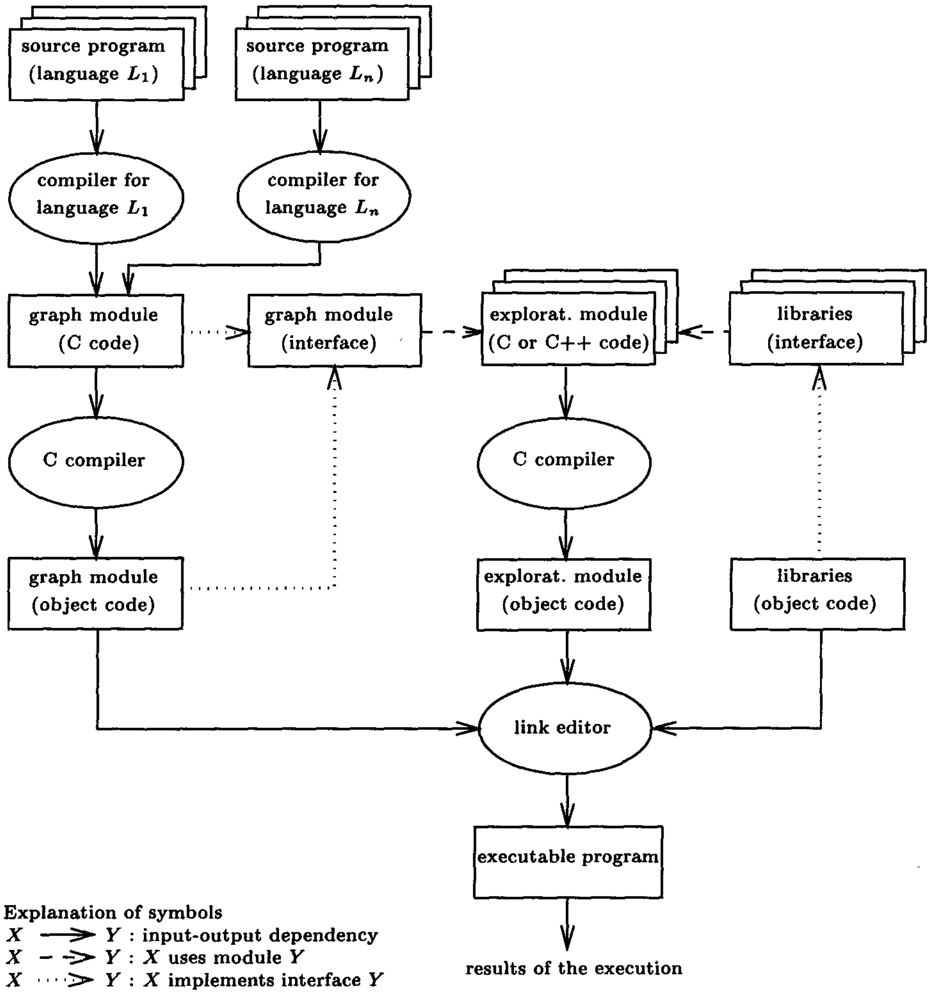
source program (language L_1)
source program (language L_n)
compiler for language L_1
compiler for language L_n
graph module (C code)
graph module (interface)
explorat. module (C or C++ code)
libraries (interface)
C compiler
C compiler
graph module (object code)
explorat. module (object code)
libraries (object code)
link editor
executable program
results of the execution
Explanation of symbols
X ⟶ Y : input-output dependency
X - -> Y : X uses module Y
X ···> Y : X implements interface Y

Fig. 1. OPEN/CÆSAR architecture

C language was chosen because efficient compilers for this language are available (C plays the role of a portable assembly language).

A list of available OPEN/CÆSAR-compliant compilers is given in Section 2. The functioning principles and internal details of these compilers are not constrained by the OPEN/CÆSAR architecture, provided that implement the "`caesar_graph.h`" interface properly.

The library module consists in a set of libraries. Each library provides a coherent set of data structures and associated primitives for handling transition lists, storing visited states during graph traversals (e.g., stacks, tables, bitmap tables), computing hash functions, displaying diagnostics, etc. These libraries are independent from any source program and source language. The available libraries (written in C and pre-compiled) are presented in Section 3, but users can add new libraries to fit their specific needs.

The exploration module can be considered as the "main" program. It contains the core of the verification, simulation, or testing algorithm, and determines how the LTS is to be explored. In most cases, the exploration module is independent from any source program and source language (however, dedicated exploration modules are possible, for instance, to verify a specific property). The exploration module uses the primitives exported by the graph and library modules. It is usually written in C or C++. It can be distributed either in source code form, or in object code form if its algorithms must be kept private. The available exploration modules are listed in Section 4.

Figure 1 illustrates the compiling and linking steps needed to merge the different code fragments (user-written code, library code, and automatically generated code) into a single executable program. Of course, some programming conventions have to be enforced in order to avoid identifier clashes between the different modules. Also, shell-scripts are available for chaining all these steps in a simple, user-friendly way and avoiding unnecessary recompilations.

2 The graph module

As said above, the graph module encapsulates all language-dependent aspects and gives access to them through a language-independent interface. Therefore, the design of such an interface is subject to antagonistic constraints:

- It should be general and abstract enough to accommodate a variety of source languages. This implies not to retain all the particular features of a given language, but to select characteristics shared by several languages. Therefore, the design of the interface relies on the existence of a common semantic model into which different source programs, written in different source languages, can be translated.
- The interface should keep track of the relationship between the semantic model and the corresponding source program, so as to provide enough information for diagnosis: when an error is detected using simulation or verification, the user should be able to understand the reason of the error in terms

of the source program. This is not always easy to implement, especially if the compiler uses sophisticated translation algorithms, involving intermediate forms and optimization techniques.

– As verification algorithms have strong efficiency requirements, the interface should be close enough to existing compilers in order not to introduce unacceptable run-time overhead.

Since 1992, the OPEN/CÆSAR graph module interface has undergone successive revisions to match these constraints. The latest version (September 1996) can be seen as a good compromise between conflicting requirements. We briefly present the main design choices:

– The interface is based upon *interleaving semantics* (which reduces concurrency to sequential composition and non-deterministic choice). Its underlying semantic model is a combination of Labelled Transition Systems and Kripke structures, which was found to be appropriate for various languages and formalisms. This model consists in a set of *states* (with an *initial state*) and a set of *transitions* between states. Depending on the source language considered, there can be additional information attached to each state (these attributes are called *state vectors*) and/or to each transition (these attributes are called *labels*).
– The interface follows the principles of abstract data type specification. It exports two "opaque" types, the *state type* and the *label type*, whose internal representations are left undefined (i.e., up to the compiler) and which can be handled using a set of primitives. There are 13 functions dealing with states and 18 functions dealing with labels.
– As regards states, OPEN/CÆSAR makes the assumption that, for a given system, state vectors can be stored in a fixed-length byte string. This restriction is meant for handling states efficiently [12], for instance when storing them in tables. However, it can be relaxed as the state vector can contain pointers to dynamic data structures (lists, FIFO queues, etc.) allocated in the heap.
  The functions exported by the interface allow to obtain the size and the alignment (in bytes) of a state, to create and delete a memory cell to store a state, to copy a state to another one, to compare two states, to compute a hash-value on a state, to print a state to a text file, to print the "differences" between two states, etc.
  By doing so, the interface provides a somehow "restricted" access to state vectors by converting them to character strings, which can be obtained when printing a state to a file or printing the differences between two states. This approach is justified by the fact that state vectors are highly language- and compiler-dependent (in particular, they often rely upon user-defined types in the source program, for which character strings provide the simplest portable interface).
  Notice that, in the case where state vectors contain pointers or, more generally, if their binary representation is not a normal form, some primitives (e.g., comparison and hashing) cannot be simply implemented as bitwise op-

erations between two memory cells. These problems are addressed in the OPEN/CÆSAR architecture.

- Similarly, labels are assumed to be fixed-length byte strings (possibly containing pointers to the heap) and functions are available for obtaining the size and the alignment (in bytes) of a label, for creating and deleting a memory cell to store a label, for copying a label to another one, for comparing two labels, for computing a hash-value on a label, etc.
  State vector restrictions also apply to labels: although label contents depend on the source language, the source program and the compiler, they can be accessed using conversion to character strings. Additional functions are available to decide whether a label is visible or not[1], in how many fields a label is subdivided, from which line of source program a label comes from, etc.
- As regards the transition relation, which is the crucial point of the graph module, OPEN/CÆSAR makes minimal requirements intentionally, in order to give maximal freedom to compiler implementors. Compilers are only required to generate C code for computing the initial state and for enumerating the successors of a given state. For the latter purpose, there are many possible approaches, many of which do not match efficiency or language-independence criteria. The OPEN/CÆSAR interface solves this problem elegantly, by introducing the concept of *callback mechanism.*
  When generating the graph module, an OPEN/CÆSAR-compliant compiler has to produce a *successor enumeration function* $\mathcal{F}$, which iterates over all the successors of a given state S_1. For each transition $S_1 \xrightarrow{L} S_2$, where L is a label and S_2 a successor state, the successor enumeration function $\mathcal{F}$ will make a function call of the form $F(S_1, L, S_2)$, where function F is passed as a parameter to $\mathcal{F}$ and is referred to as the *callback function.* Function F can perform any action, e.g., printing the transition $S_1 \xrightarrow{L} S_2$ to a file, storing S_2 in a state table, etc. It can be either defined by the user in the exploration module, or imported from the library module, which provides several predefined callback functions of general interest. The enumeration of transitions going out of state S_1 is sequential: no direct access to the i^{th} successor is required. The order in which function $\mathcal{F}$ enumerates the successors is left to the compiler. Function F can do side-effects, but it should not invoke function $\mathcal{F}$ recursively (therefore, the iteration mechanism needs not be reentrant). On top of this primitive (but general and efficient) callback mechanism, more elaborated facilities can be developed (see the EDGE library in Section 3).

At the time being, there exist 5 different implementations of OPEN/CÆSAR-compliant compilers, which we briefly review:

1. After designing the OPEN/CÆSAR architecture, the author adapted the CÆSAR compiler [12] accordingly. The core of CÆSAR's compiling algorithms (based upon the translation of LOTOS to an intermediate extended Petri net

[1] i.e., the concept of τ-transitions in process algebras

model) was kept unchanged; only the back-end of CÆSAR was modified for compliance with the "caesar_graph.h" interface. In this implementation, each state is a pair $\langle M, C\rangle$ where M is a marking of the Petri net, and C is a context mapping state variables to their values; each transition is generated by the firing of a corresponding transition in the Petri net; each label consists of a gate name followed by a list of exchanged values. The algebraic data types contained in the source LOTOS program can be either translated in C code by the CÆSAR.ADT compiler or implemented manually by the user; in both cases, data structures dynamically allocated in the heap are supported.

2. In 1994, Renaud Ruffiot and the author connected the BCG format for the representation of Labelled Transition Systems [10] to the OPEN/CÆSAR environment. The resulting BCG_OPEN tool enabled the application of all OPEN/CÆSAR tools (see Section 4) to graphs entirely generated and represented in the BCG format. In this implementation (700 lines of C code), each OPEN/CÆSAR state (*resp.* label, transition) is directly mapped to the corresponding BCG state (*resp.* label, transition). The development of BCG_OPEN led to a modification of the "caesar_graph.h" in order to remove some LOTOS-specific aspects.
3. In 1995, Marius Bozga, Jean-Claude Fernandez and Laurent Mounier (VERIMAG, France) developed the EXP.OPEN compiler, which allows to use OPEN/CÆSAR for the compositional verification of networks of communicating automata. The input language accepted by EXP.OPEN consists in a set of automata (entirely generated) connected together using the parallel composition and hiding operators of LOTOS. In their implementation (3,000 lines of C code, including a LEX scanner and a YACC parser), each OPEN/CÆSAR state is a tuple of the individual states of the automata, and transitions are obtained by applying the LOTOS semantics rules for parallel composition and hiding.
4. In 1997, Khalid Laksiouar and Amar Bouali (INRIA Sophia-Antipolis, France) developed the FC2OPEN compiler to connect the FC2 toolset [3] to OPEN/CÆSAR. FC2OPEN takes as input FC2 models, which are either automata or networks of communicating automata connected together by means of so-called *synchronization vectors.* In their implementation (3,000 lines of C++ code), each OPEN/CÆSAR state is either an automaton state or a tuple of local states, and transitions are determined according to the semantics of synchronization vectors.
5. In 1997, Alain Kerbrat, Carlos Rodriguez, and Yves Lejeune (VERIMAG/VERILOG, France) connected VERILOG's OBJECTGEODE tool [2] for SDL to the CÆSAR/ALDÉBARAN toolbox. One aspect of this connection was the developement of a gateway between OBJECTGEODE and OPEN/CÆSAR [17].

3 The library module

OPEN/CÆSAR provides a library of useful, generic facilities. We give an overview of them:

- The EDGE library is built on top of the graph module. It extends the callback mechanism described in Section 2 with higher-level functionalities:
    - The callback mechanism enumerates sequentially the successors of a given state S_1, but does not store them in memory. Moreover, as the callback mechanism is not supposed to be reentrant, it does not allow depth-first traversals algorithms to be programmed recursively. The EDGE library solves this problem by building transition lists, which can be used for programming depth-first traversals. Transition lists are linked lists of tuples $\langle S_1, L, S_2, M\rangle$, where L is a label, S_2 a successor state and M a byte-string in which users can put any information they want. All fields S_1, L, S_2, and M are optional and can be omitted if not relevant to the exploration algorithm under consideration.
    - The order in which the callback mechanism enumerates the successors is left unspecified, but the EDGE library can sort transition lists according to various criteria (e.g., lexicographic order over the L fields). This can be useful, for instance, in an interactive simulator, for displaying to the user an alphabetically-sorted list of transitions.
    - The EDGE library also exports many primitives to create, delete, copy, print, and reverse transition lists; to compute the length and access directly the i^{th} element of transition lists; to access the different tuple fields S_1, L, S_2, and M of a given element. It also provides iterators over transition lists and automatically truncates the transition lists if the available memory is unsufficient to store all successors of a given state.
- The HASH library provides various predefined hash-functions which can be applied to states and labels considered as byte strings. These functions are needed for accessing hash-tables and for Holzmann's algorithm [14]. OPEN/CÆSAR users can add their own hash-functions: see for instance [6], where OPEN/CÆSAR is used for a comparative analysis of various hashing techniques.
- The STACK_1 library is built on top of the EDGE library and provides primitives for managing one or several stacks[2]. Depth-first search algorithms rely on stacks to store the execution path taken from the initial state. These stacks are not merely stacks of states: it is also necessary to store the transitions between states for characterizing an execution path entirely. Also, depth-first search algorithms require to store the list of states remaining to be explored at each stack depth. Therefore, each element in a stack consists of three fields: a *state field* S, a *label field* containing the label of the last transition performed before reaching state S (or a null pointer if S is the initial state), and an *edge field* containing the list of transitions going out of state S that have not been explored yet. For a given stack, the label and edge fields are

[2] The number "1" occurring at the end of the name STACK_1 denotes the fact that this library is a particular implementation of the stack, and that alternative implementation could be offered in future versions of OPEN/CÆSAR; this is also the case for the other data structures presented below.

optional: if none of them are present, the stack behaves as a simple stack of states.

The STACK_1 library provides a set of classical primitives for dealing with stacks. These primitives allow to create, delete, copy, and print stacks; to erase the contents of a stack; to check whether a stack is empty; to compute the depth of a stack; to access the fields of the element on top of a stack; to push or pop an element on top of a stack; etc.

There are also specific primitives for depth-first search. They allow to deal with the list of successors of the state on top of the stack and, more specifically to create or delete this list; to check whether it is empty; to compute its length; to remove its first element; to extract its first element and to push it on top of the stack; etc.

The STACK_1 library provides additional features suitable for on-the-fly verification. For instance, when creating a stack, one can specify a maximal depth not to be exceeded, as well as the action to be taken if this maximal depth is reached or when the stack overflows because of a memory shortage (e.g., stopping the exploration, backtracking to the previous state, etc.).

- The TABLE_1 library for managing one or several *state tables*, i.e., tables for storing the states of a program which are visited during a graph traversal. Each element in a given table is a byte string, subdivided into two fields, the "*base*" field and the "*mark*" field. The sizes of these fields is specified when the table is created and it is the same for all the elements in the table. More often than not, base fields contain states of the graph being explored (these states are those produced by the graph module). However, the base fields can be used to store other data. The contents of mark fields (possibly empty) are determined by the user. During graph traversals, mark fields are generally used to store additional attributes attached to states (i.e., base fields). To allow fast access, each table is equipped with an auxiliary hash table that allows to retrieve an element having a given base field.

  The primitives offered by the TABLE_1 library allow to create and delete tables; to erase their contents and to print it under various formats; to access the base and mark fields of an element given its index; to put or get an element; to search an element given its base field, optionally inserting it in the table if not already present; to determine if a table is empty or full; etc.
- The BITMAP library provides primitives for managing one or several *bitmap tables* (i.e., large bit arrays) such as those used in Holzmann's bit-space algorithm [14]. In addition to the basic test-and-set primitives, it provides convenient features, such as automatical dimensioning of the table size to the greatest prime number less than the requested size, dump of the bitmap table to a text file under various formats, usage statistics recording and display, etc.
- The DIAGNOSTIC_1 is built on top of the STACK_1 library and provides primitives for dealing with diagnostic sequences (e.g., execution sequences leading to deadlock states). It allows to specify which diagnostic sequences will be displayed to the user and to control the exploration algorithm according to

various strategies (for instance, to find the shortest possible diagnostic sequence).

4 The exploration module

On top of the graph and library modules, the exploration module plays the role of the main program: it determines how and why the graph will be explored. There are many different possibilities for exploring a graph $\mathcal{G}$. In an attempt to establish a taxonomy, we list below the essential parameters ("degrees of freedom") that can be tuned by the exploration module:

Definition of states and transitions: it is often the definition exported by the graph module; however, in some cases, this definition has to be modified. For instance, when evaluating temporal logic or μ-calculus formulas on $\mathcal{G}$, or when checking behavioural equivalences (e.g., bisimulation equivalences, preorders, trace inclusion, etc.) between $\mathcal{G}$ and some other graph, one often uses "*product states*" of the form (S, S'), where S is a state of $\mathcal{G}$ and S' a state of another graph (or observer, or Büchi automaton, or linear trace, etc.) noted $\mathcal{G}'$; the transition relation must also be extended to reflect concurrency and synchronization constraints between $\mathcal{G}$ and $\mathcal{G}'$.

Selection of successor states: when at a given state S of $\mathcal{G}$, the exploration program must decide how many successors of S (if any), and which, should be visited. There are several possible answers: *none of them* (i.e., backtracking if some boolean condition is false or if the available memory is exhausted); *one of them* (e.g., chosen randomly in the case of random execution, or by prompting the user in the case of interactive simulation); *all of them* (in the case of reachability analysis); *some of them*, according to various heuristics related to the level of coverage expected. If several successors of S are selected, the order in which they should be enumerated must also be specified.

Storage policy: the exploration program must also decide how many states, and which, should be stored in memory. There are many choices: *only the current state* (e.g., in random execution), *only the states on the path leading from the initial state to the current state* (e.g., in interactive simulation allowing unbounded backtracking facilities), *all the states* (e.g., when constructing the entire graph to perform model-checking at a later stage), *all the states up to a maximal number*, etc.

There are even more sophisticated strategies [16] allowing to discard states stored in memory when some upper limit on the number of states is reached, or when no more memory is available, with again a choice between various replacement policies inspired from garbage collecting (e.g., discarding first the most recent states, the oldest ones, etc.).

Instead of storing states under the exact representation exported by the graph module, it is also possible to store only a "condensed" form of them by using some compression function mapping states to a smaller bit string (typical examples of such functions are hash-functions and cryptographic message digest functions). The classical approach is known as "bitstate hashing" [14], but more elaborate variants exist [15]. Of course, as the compression function is not injective, the exploration algorithm must take into account the fact that two different states may have the same condensed form. Again, the choice of the compression function is left open.

Traversal algorithm: the exploration module has also to decide which type of algorithm should be used (depth-first search, breadth-first search, etc.), as well as many other parameters (for instance, having a maximal exploration depth).

Many exploration modules have been developed within the OPEN/CÆSAR framework[3]; most of them are distributed within the CADP toolbox. We review them briefly:

- DECLARATOR is a debugging tool that exercises all the primitives exported by the "caesar_graph.h" interface. This tool is used to check and validate OPEN/CÆSAR-compliant compilers.
- EXECUTOR is a random execution tool, which produces a random trace starting from the initial state. Various options are available, e.g., to control the seed of the random number generator, to report non-deterministic choices, to display or not invisible transitions, to have an upper limit on the number of transitions fired, etc.
- SIMULATOR is an interactive simulator allowing step-by-step execution (with backtracking) controlled from a command-line interface. XSIMULATOR is a graphical, TCL/TK-based extension of SIMULATOR developed by Mark Jorgensen, Jean-Michel Frume and the author.
- GENERATOR performs reachability analysis to generate exhaustively the LTS (represented in the BCG format) corresponding to a source program. REDUCTOR is similar to GENERATOR, but performs on-the-fly reduction modulo the τ^*a equivalence (which preserves all safety properties).
- TERMINATOR is a deadlock detection tool implementing Holzmann's "bit-state" (or "supertrace") algorithm [14], with various improvements regarding the generation of diagnostic sequences.
- EXHIBITOR searches on-the-fly for execution sequences starting from the initial state and whose labels match a given "pattern". The language used to describe patterns combines boolean operators and (a subset of) regular expressions with an extension to characterize deadlock states. EXHIBITOR implements a depth-first search algorithm and a breadth-first search algorithm, the latter being able to find the shortest sequence(s) matching a given pattern.
- EVALUATOR [9] is an on-the-fly model-checking tool for branching-time μ-calculus developed by Marius Bozga, Jean-Claude Fernandez and Laurent Mounier. It implements two different model-checking algorithms: a global one and a local one.
- ALBATOR is a tool developed by Laurent Mounier and Laurent Aublet-Cuvelier to check on-the-fly whether two LTSs are equivalent modulo strong bisimulation.

[3] All these tools have been developed by the author, unless specified otherwise by bibliographic reference or explicit mention of the author(s)

- PROJECTOR [18] is a tool for compositional verification. For each process of the source program, PROJECTOR allows to generate the corresponding LTS in a constrained manner, by taking into account an *interface*, i.e., an LTS expressing (a superset of) the set of execution sequences permitted for this process by its environment.
- TGV [7] is a test generation tool based on verification technology. Given a source program and an automaton formalizing the behavioural part of a test purpose, TGV produces the behaviour description and constraints definitions of a test case in the standard TTCN format.

Concluding remarks and future work

In this paper, we have presented the motivations and achievements of a long-term project, which spanned over the last five years.

We have defined the principles of the OPEN/CÆSAR architecture, a software framework for developing tools that integrate simulation, verification and test generation functionalities in an coherent way. The main principles underlying this architecture are:

Modularity: a clear separation is established between the language-dependent part (definition of states and labels, and computation of the transition relation) and language-independent parts (exploration algorithms themselves). This separation is achieved using the OPEN/CÆSAR API, whose design has been continuously reviewed and improved during the past years. Technically, this interface realizes a good tradeoff between various (conflicting) requirements: expressiveness, language independence, efficiency, portability, genericity, etc.

Reusability: in addition to the modularity principle, the OPEN/CÆSAR also promotes reusability, by providing a library of predefined utilities (thus avoiding to users the tedious process of implementing and debugging data structures such as stacks, state tables, etc.). These libraries are accessible using well-defined interfaces and follow established software engineering methodologies (namely abstract data types and object-orientation).

Orthogonality: within the OPEN/CÆSAR framework, any verification or testing algorithm can be applied to any source language. Thus, in the specialized area of protocol engineering, OPEN/CÆSAR achieves goals similar to those of UNCOL [23], the universal intermediate language, a most inspiring paradigm, discussed but never implemented. This orthogonality property is especially of interest when designing user interfaces: in particular, the EUCALYPTUS graphical user-interface [10] takes advantage of it to present the available operation for each type of source program in a uniform, regular manner.

Openness: as the exploration module can be written in a general-purpose programming language (e.g., C or C++) and relies upon the link edition mechanism offered by the operating system, the user is free to write any possible algorithm. This approach strongly contrasts with more limited solutions in which the user is only given access to a few parameters for controlling the simulation and reachability analysis, but not to a full-fledged programming interface [1,2].

As time passed, the OPEN/CÆSAR approach proved to be superior, so that industrial tools recently switched to the OPEN/CÆSAR principles by developing a similar API, including a direct connection to OPEN/CÆSAR [17].

Extensibility: the OPEN/CÆSAR environment can be extended in three ways: by adding new connections to source languages, by adding new exploration algorithms, and by adding new libraries to fit specific needs.

The idea of integrating various techniques within a single tool is becoming increasingly popular. Prior to OPEN/CÆSAR, there have been many attempts at turning a simulation tool into a verification or test generation tool. In particular, the SPIN tool [14] allowed to combine simulation and model-checking several years before the first version of OPEN/CÆSAR; however, as SPIN is designed for a single language, PROMELA, its internal architecture remains rather monolithic. Also, some of the OPEN/CÆSAR principles were already present when the author designed the internal architecture of VESAR [1], but not in such a systematic way.

It was the intrinsic merits of OPEN/CÆSAR to formulate the principle of a radical separation between three modules (graph, exploration and libraries), to design and specify the corresponding APIs, and to prove the feasibility of these ideas by providing a complete implementation (the first version of the OPEN/CÆSAR environment was distributed in April 1992 as a part of version R of CADP).

As regards the development of verification tools applicable to different source languages, we can also mention the *Process Algebra Compiler* (PAC) [5], a compiler generation tool for process algebras specified by their BNF syntax and their SOS semantics. The main difference between PAC and OPEN/CÆSAR relies in the fact that PAC provides an implementation for a well-defined class of languages, whereas OPEN/CÆSAR leaves implementation matters to OPEN/CÆSAR compliant compilers. As OPEN/CÆSAR only assumes the existence of states, labels, and transitions, it raises less constraints on the source language, the way it is defined, and the way it is executed: thus, OPEN/CÆSAR can accomodate a wider class of languages (i.e., value-passing process algebras, imperative languages, etc.). Nevertheless, both approaches are not mutually exclusive and could interoperate, as the PAC approach could be used to generate OPEN/CÆSAR compliant compilers automatically.

We believe that the OPEN/CÆSAR environment should be of interest to several categories of people:

Language/compiler designers, who connect their compilers to the OPEN/CÆSAR API can immediately reuse for their language all the existing OPEN/CÆSAR tools available for simulation, verification, and testing. The connection task consists in providing an implementation for the primitives defined in the graph module, which should be straightforward, as all language-dependent features have been gradually lifted out from OPEN/CÆSAR's API. At the time being, five different formalisms are already connected to OPEN/CÆSAR: two standardized high-level languages (LOTOS and SDL), two formalisms for describing networks of communicating finite-state machines (EXP and FC2), and a formalism for representing LTSs (BCG). Our experience indicates that such a connection can be established in 4–6 weeks by a computer-science student without prior knowledge in verification theory.

Algorithm designers, who propose new algorithms for verification and testing will find in OPEN/CÆSAR a rapid prototyping platform for experimenting their ideas. At present, many tools have already been developed within OPEN/CÆSAR, which cover many aspects of protocol engineering (random execution, interactive simulation, reachability analysis, on-the-fly verification of bisimulation and μ-calculus, test generation, etc.) and demonstrate the applicability of OPEN/CÆSAR for a wide spectrum of problems. OPEN/CÆSAR allows to bridge the gap between theoretical research and practical applications by providing a "programming kit" to implement concisely, quickly, and efficiently new algorithms, under a form close to the way these algorithms are specified on paper. It is worth noticing that these algorithms can be written in a fully language-independent way, without the need to develop a compiler from scratch (nor to adapt the code of an existing compiler, if available); yet, they can still be applied to real-life examples, by simply using one of the existing OPEN/CÆSAR-compliant compilers. In this respect, OPEN/CÆSAR could play the role of a common framework for comparing and assessing the performances of different algorithms.

Protocol designers, who are concerned by applicative aspects (but are not interested in developing new languages, compilers, or algorithms) can benefit from a complete set of robust tools, covering almost all aspects of protocol engineering. These tools can easily be accessed from a graphical-user interface [10] and have been field-tested on several real-life applications [4, 11, 19, 21, 22].

Naturally, the design choices of OPEN/CÆSAR induce several limitations and drawbacks. Although these limitations are not considered to be crippling by OPEN/CÆSAR users (especially, industrial users), they leave room for further research and improvements. We briefly discuss the main ones:

- As a counterpart for modularity and reusability, there is a price to pay in terms of performance. For instance, when constructing the state graph of a LOTOS description, the OPEN/CÆSAR GENERATOR tool is slightly less efficient than the dedicated CÆSAR tool. However, this overhead is felt acceptable.
- To achieve language independence, OPEN/CÆSAR operates at the level of a Labelled Transition System model. This creates a gap between the source level program (usually written in a language involving some form of concurrency) and the model of this program, as it is made available by the graph module. There are already some "hook" primitives to keep track of the correspondence between the model and the source program, but they could be enhanced in several ways. For instance, the interface could give more information about the concurrent processes that exist at the source program level, e.g., by indicating in which state a given process is, which processes participate in a given transition, etc. This kind of information is needed by verification algorithms using partial orders and symmetries. Also, it would be desirable to have a more accurate access to the values contained in states and labels (at present, state contents and label contents are represented as

character strings). This would be useful for debugging purpose (for instance, to inspect the value of a variable). However, such facilities are often language-dependant, and require to keep track of the types and functions defined in the source program. A proper treatment of user-defined types and functions[4] would definitely make OPEN/CÆSAR a much more complex system.

- At present, the interface of the graph module allows on-the-fly exploration for a single source program only. This interface could be extended to handle several graphs simultaneously. However, we have not found yet a practical situation where such an enhancement would be needed. Even algorithms for computing bisimulations on-the-fly between two graphs [8] assume that one graph is small enough for being generated exhaustively, so that only one graph remains to be explored on-the-fly.
- When computing the transition relation, OPEN/CÆSAR only gives access to the successors of a given state, but not to the predecessors. Although this is a limitation for some verification algorithms (e.g., [20]), it is justified, as the computation of state predecessors for high-level languages is undecidable in the general case (because of user-defined data types, user-defined functions over these data types, assignment to variables and boolean conditions).
- OPEN/CÆSAR's graph module interface deals with states one by one. This interface remains to be extended in order to deal with symbolic methods (e.g., methods based upon binary decision diagrams or polyhedra) that deal with sets of states, for which they often provide a more efficient representation than lists of isolated states.
- Finally, it is planned to extend the graph module interface with a notion of quantitative time. This is needed for applying OPEN/CÆSAR to timed languages (e.g., the forthcoming ISO standard Extended-LOTOS) that rely upon timed Labelled Transition Systems.

OPEN/CÆSAR can be obtained free of charge as a component of the CADP toolset. See the CADP Web page (`http://www.inrialpes.fr/vasy/cadp.html`) for further information.

References

1. B. Algayres, V. Coelho, L. Doldi, H. Garavel, Y. Lejeune, and C. Rodriguez. VESAR: A Pragmatic Approach to Formal Specification and Verification. *Computer Networks and ISDN Systems*, 25(7):779–790, February 1993.
2. B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL: Observing SDL behaviors with GEODE. In *Proc. 7th SDL Forum (Oslo, Norway)*, September 1995.
3. A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In *Proc. CAV '96*, LNCS 1102, 1996.
4. G. Chehaibar, H. Garavel, L. Mounier, N. Tawbi, and F. Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In *Proc. FORTE/PSTV'96*. Chapman & Hall, 1996. Full version available as INRIA Research Report RR-2958.

[4] Such an approach has already been investigated in the context of the BCG format.

5. R. Cleaveland, E. Madelaine, and S. Sims. A Front-End Generator for Verification Tools. In *Proc. TACAS'95 Tools and Algorithms for the Construction and Analysis of Systems (Aarhus, Denmark)*, May 1995. Also available as INRIA Research Report RR-2612.
6. B. Cousin and J. Helary. Performance Improvement of State Space Exploration by Regular and Differential Hashing Functions. In *Proc. CAV'94*, LNCS 818, 1994.
7. J-Cl. Fernandez, Cl. Jard, Th. Jéron, L. Nedelka, and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, 29(1–2):123–146, July 1997.
8. J-Cl. Fernandez and L. Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In *Proc. CAV'91*, July 1991.
9. J-Cl. Fernandez and L. Mounier. A Local Checking Algorithm for Boolean Equation Systems. Rapport SPECTRE 95-07, VERIMAG, Grenoble, March 1995.
10. H. Garavel. An Overview of the Eucalyptus Toolbox. In *Proc. COST 247 Int. Workshop on Applied Formal Methods in System Design (Maribor, Slovenia)*, 1996.
11. H. Garavel and L. Mounier. Specification and Verification of various Distributed Leader Election Algorithms for Unidirectional Ring Networks. *Science of Computer Programming*, 29(1–2):171–197, July 1997.
12. H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In *Proc. PSTV '90 (Ottawa, Canada)*. North-Holland, 1990.
13. S. Graf, J-L. Richier, C. Rodríguez, and J. Voiron. What are the Limits of Model Checking Methods for the Verification of Real Life Protocols? In *Proc. 1st Workshop on Automatic Verification Methods for Finite State Systems*, LNCS 407, 1989.
14. G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
15. G. J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proc. 3rd SPIN Workshop (Twente Univ., The Netherlands)*, 1997.
16. Cl. Jard and Th. Jéron. Bounded-Memory Algorithms for Verification On-the-Fly. In *Proc. CAV '91*, LNCS 575, July 1991.
17. A. Kerbrat, C. Rodriguez, and Y. Lejeune. Interconnecting the ObjectGEODE and CÆSAR/ALDEBARAN Toolsets. In *Proc. 8th SDL Forum*, 1997.
18. J-P. Krimm and L. Mounier. Compositional State Space Generation from Lotos Programs. In *Proc. TACAS'97*, LNCS 1217, 1997.
19. R. Mateescu. Formal Description and Analysis of a Bounded Retransmission Protocol. In *Proc. COST 247 Int. Workshop on Applied Formal Methods in System Design (Maribor, Slovenia)*, 1996. Also available as INRIA Research Report RR-2965.
20. R. Paige and R. E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.
21. Ch. Pecheur. Specification and Verification of the CO4 Distributed Knowledge System Using LOTOS. In *Proc. 12th IEEE Int. Conf. on Automated Software Engineering ASE-97*, 1997. Extended version available as INRIA Research Report RR-3259.
22. M. Sighireanu and R. Mateescu. Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS. In *Proc. 2nd COST 247 Int. Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, 1997. Full version available as INRIA Research Report RR-3172.
23. T. B. Steel. A First Version of UNCOL. In *Proc. Western Joint Computer Conf.*, pages 371–378, May 1961.