CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes*

Hubert Garavel, Frédéric Lang, Radu Mateescu, Wendelin Serwe

INRIA / Laboratoire d'Informatique de Grenoble, VASY team, 655, avenue de l'Europe, 38330 Montbonnot St Martin, France
e-mail: {Hubert.Garavel,Frederic.Lang,Radu.Mateescu,Wendelin.Serwe}@inria.fr

Received: July 6, 2012

Abstract. CADP (*Construction and Analysis of Distributed Processes*) is a comprehensive software toolbox that implements the results of concurrency theory. Started in the mid 80s, CADP has been continuously developed by adding new tools and enhancing existing ones. Today, CADP benefits from a worldwide user community, both in academia and industry. This paper presents the latest release, CADP 2011, which is the result of a considerable development effort spanning the last five years. The paper first describes the theoretical principles and the modular architecture of CADP, which has inspired several other recent model checkers. The paper then reviews the main features of CADP 2011, including compilers for various formal specification languages, equivalence checkers, model checkers, compositional verification tools, performance evaluation tools, and parallel verification tools running on clusters and grids. Finally, the paper surveys some significant case studies.

Key words: asynchronous concurrency, equivalence checking, formal methods, model checking, performance evaluation, process calculus, verification.

1 Introduction

Among all the scientific issues related to the reliability of computer systems, concurrency has a major place, because the design of parallel systems is a complex, error-prone, and largely unmastered activity. Thirty years after the first attempts at building automated verification tools for concurrent systems, the problem is still there; it has even gained in relevance because system complexity has increased, and because concurrency is now ubiquitous, from multicore microprocessors to massively parallel supercomputers.

To ensure the reliability of a concurrent system under design, it is understood that the first step consists in establishing a precise model of the system behavior, this model usually consisting of several concurrent processes, together with a description of the data types, constants, variables, and functions manipulated by these processes. This opens the debate on the most appropriate languages to express system models, with a large choice of candidates ranging from semi-formal to formal languages.

Once a precise, if not formal, model is available, one needs automated methods to prove the correctness of the system with respect to its specification or, at least, to search for the presence of certain mistakes. Without neglecting recent progresses in theorem proving and static analysis, state space exploration techniques (among which reachability analysis and model checking) remain the most successful approaches for dealing with complex concurrent systems, especially during the design phase, when system specifications are evolving frequently.

State space exploration techniques are usually grouped in two classes: *enumerative* (or *explicit state*) techniques consider each state of the system separately, whereas *symbolic* (or *implicit state*) techniques manipulate sets of states represented using either decision diagrams (BDDs and their variants) or logical formulas whose satisfiability is determined using SAT and SMT solvers. In this paper, we will use the term enumerative instead of explicit-state in order to avoid possible confusions with the terminology about explicit and implicit models (see Section 2). Enumerative techniques, which historically were introduced first [124], are based on a forward exploration of the transition relation be-

* This work has been partly funded by Bull, by the French National Agency for Research (project OpenEmbedd), by the French Ministry of Economics and Industry (Aerospace Valley project Topcased), and by the *Conseil Général de l'Isère* (Minalogic project Multival).

tween states (*post* function), making them suitable for the on-the-fly verification of specifications written in languages with arbitrary data types. Although they enable exploration of *a priori* fewer states than their symbolic counterparts, enumerative techniques prove to be adequate for the analysis of asynchronous parallel systems containing complex data structures.

CADP (*Construction and Analysis of Distributed Processes*)[1] is a toolbox for verifying asynchronous concurrent systems. The toolbox, whose development started in 1986, is at the crossroads between several branches of computer science: concurrency theory, formal methods, and computer-aided verification. Initially, CADP consisted of only two tools: CÆSAR [38], a compiler and explicit state space generator for the LOTOS language, and ALDÉBARAN [32,35], an equivalence checker based on bisimulation minimization. Over the past 25 years, CADP has been continuously improved and extended [33,47,48]. This paper is an extended version of [49] and presents the latest release, CADP 2011 "*Zurich*", which currently contains 45 tools.

CADP now offers a comprehensive set of functionalities covering the entire design cycle of asynchronous systems: specification, interactive simulation, rapid prototyping, verification, testing, and performance evaluation. For verification, it supports the three essential approaches existing in the field: model checking, equivalence checking, and visual checking. To deal with complex systems, CADP implements a wide range of verification techniques (reachability analysis, on-the-fly verification, compositional verification, distributed verification, static analysis) and provides a scripting language for describing elaborate verification scenarios. In addition, CADP 2011 brings deep changes with respect to previous releases, especially the support for many different specification languages.

Related Work. There are many model checkers developed in the world. By taking the Wikipedia list of model checking tools[2] and complementing it with the list of verification tools established at the University of Brno[3], one obtains a cumulated list of 64 tools (as of January 2012). By considering only those tools that are still actively maintained (i.e., those for which at least one new version was released in 2010, 2011, or early 2012), one can restrict this list to 32 tools only, but this is still quite a number of tools that should be compared with CADP. Moreover, there are plenty of possible criteria for such comparison. We can mention the following ones.

From an historical perspective, CADP and SPIN [74] seem to be the two oldest model checkers still available. The development of both tools was undertaken in the 80s and is still going on with, e.g., support for 64-bit computing platforms, as well as application in recent industrial case-studies.

From a functionality perspective, CADP provides a unique combination of features that no other tool presently offers. This can be seen by formulating four essential criteria:

(C1) Does the tool support not only model-checking but also equivalence checking which, beyond being standard practice in hardware verification, plays a crucial role for component-based systems and compositional verification?
(C2) Does the tool support distributed verification, i.e. can it use the computing power and memories of a cluster of machines, rather than a single machine?
(C3) Does the modeling language of the tool support concurrency, i.e. does it have some builtin notion of asynchronous parallel composition?
(C4) Does the modeling language support user-defined (possibly unbounded) data types such as records, unions, lists, etc. (and not only boolean, integers, and enumerated types)?

tool name	(C1)	(C2)	(C3)	(C4)
ARC	no			
Alpina	no			
Cadence SMV	no			
CADP	yes	yes	yes	yes
CBMC	no			
CPAchecker	no			
EmbeddedValidator	no			
DIVINE	no			
FDR	yes	no		
HSolver	no			
ImProve	no			
JPF	no			
LTSMIN	yes	yes	yes	(*)
MCMAS	no			
mCRL2	yes	no		
MRMC	yes	no		
νSMV	no			
PAT	no			
PRISM	no			
PVS	yes	no		
Reactis Tester	no			
RED/OMPCA	yes	no		
Roméo	no			
SATABS	no			
SPIN	no			
TAPAAL	no			
TAPAS	yes	no		
Temporal Rover	no			
TLC	no			
UPPAAL	no			
YMER	no			
[mc]square	no			

(*) not applicable

[1] http://cadp.inria.fr
[2] http://en.wikipedia.org/wiki/List_of_Model_Checking_Tools
[3] http://anna.fi.muni.cz/yahoda

To the best of our knowledge, CADP is the only tool to satisfy criteria (C1) to (C4). It is followed by the LTSMIN [7] that is interconnected to CADP. Let us mention that another distinctive trait of CADP is the inclusion of performance evaluation tools, in addition to functional verification tools.

A detailed comparison of CADP with other explicit-state model checkers will be given in Section 2.

Outline. This paper gives an overview of CADP 2011, highlighting new tools and recent enhancements. It is organized as follows. Section 2 presents the core semantic models of CADP — namely (explicit and implicit) labeled transition systems (LTSs), Markov chains, and (parameterized) Boolean equation systems (PBESs). Section 3 describes the four languages now supported by CADP — namely LOTOS, FSP, EXP, and LOTOS NT — and lists translations developed for other languages. Section 4 presents the visual checking features of CADP. Section 5 presents the model checking features of CADP, namely the XTL language, the EVALUATOR 3.6 model checker for regular alternation-free μ-calculus, and the EVALUATOR 4.0 model checker for MCL, an extension of modal μ-calculus with typed variables and data manipulation constructs. Section 6 is devoted to the equivalence checking features of CADP — namely the BCG_MIN tool for bisimulation minimization and the BISIMULATOR tool for on-the-fly comparison of LTSs. Section 7 presents the compositional verification features of CADP. Section 8 describes the tools of CADP for performance evaluation, in particular steady-state/transient analysis and simulation of Markov chains. Section 9 outlines the CADP tools for parallel and distributed verification. Section 10 surveys some significant case studies. Finally, Section 11 summarizes the achievements and indicates directions for future work.

2 Architecture and Verification Technology

Compared to other explicit-state model checkers (especially SPIN [75]), CADP has the following principles and distinctive features (some of which were already present in precursory tools rooted in concurrency theory, such as CWB [24] and CWB-NC [23]):

- CADP supports both high-level languages with a formal semantics (process calculi) and lower level formalisms (networks of communicating automata); it also accepts connections from informal or semi-formal languages that have a means to compute the *post* transition function.
- Contrary to most model checkers supporting only scalar types, CADP has from the outset supported concurrent programs with complex and/or dynamic data structures (records, unions, lists, trees, etc.) provided that these data structures are not shared between concurrent processes.
- CADP relies on *action-based* (rather than *state-based*) semantic models inherited from concurrency theory, in which one can only refer to the observable communication actions performed by a system instead of the internal contents of states, which are supposed to be hidden and implementation dependent, and thus are not abstract enough. This encompasses the classical concepts of LTSs (for verification), discrete- and continuous-time Markov chains (for performance evaluation), and extended Markovian models, such as *Interactive Markov Chains* (IMCs) [70], which combine LTSs and Markov chains.
- Relying on action-based models enables equivalence checking, i.e., the comparison of specifications for equality or inclusion; this corresponds to the notions of bisimulations for LTSs and aggregation/lumpability for Markov chains. Also, the possibility of replacing a state space by an equivalent but smaller one is fundamental in compositional verification.
- As a consequence, the model checkers of CADP are based on branching-time (rather than linear-time) logics, which are adequate with bisimulation reductions.
- CADP is equipped with an original software architecture designed to enable modularity in model checking tools. Early model checkers (such as CESAR [110, 36], EMC [21,22], XESAR [62], and SPIN [74]) were "monolithic" in the sense that they tightly combined (1) the source language used to specify the concurrent system under verification and the compiling algorithms used to generate/explore the state space of the concurrent system, and (2) the temporal logic language used to specify correctness formulas and the verification algorithms that evaluate these formulas over the state space. CADP took a different approach and adopted a modular architecture with a clear separation between language-dependent and language-independent aspects. Different verification functionalities are implemented in different tools, which can be reused for several languages and which are built upon well-specified interfaces that enable code factoring.
- CADP 2011 can manage state spaces as large as 10^{10} explicit states; by employing compositional verification techniques on individual processes, much larger state spaces can be handled, up to sizes comparable to those reached using symbolic techniques, such as BDDs.

CADP can be seen as a rich set of powerful, interoperating software components for manipulating automata and Markov chains. All these tools are integrated in two ways: for interactive use, a graphical user-interface (named EUCALYPTUS) with contextual menus is provided; for batch use, a user-friendly scripting language, SVL, was designed, providing powerful verification strategies — to the best of our knowledge, the SVL

feature of CADP is unique and there is no equivalent in competing toolboxes.

Explicit state spaces. In the terminology of CADP, an *explicit* state space is a state-transition graph defined *extensively*, meaning that the sets of states and transitions are entirely known, because they have been already computed.

In the early 90s, most verification tools represented explicit state spaces using textual file formats, which were only adequate for small graphs but would not scale satisfactorily, e.g., to millions of states. To solve this issue, CADP was equipped in 1994 with BCG (*Binary-Coded Graphs*), a portable file format for storing LTSs (with provisions for storing also Kripke structures and Kripke Transition Systems although these two models are not used in the setting of CADP). BCG is a binary format, which was designed to handle large state spaces (up to 10^8 states and transitions initially — this limit was raised to 10^{13} in CADP 2011 to take into account 64-bit machines). Because the BCG format is not human readable, it comes with a collection of code libraries and utility programs for handling BCG files.

Two key design goals for BCG are file compactness and the possibility to encode/decode files quickly and dynamically (i.e., without requiring knowledge of the entire state space in advance); these goals are achieved using dedicated compression techniques that give significant results: usually, two bytes per transition on average, as observed on VLTS (*Very Large Transition Systems*)[4], a benchmark suite used in many scientific publications. A third design goal is the need to preserve in BCG files the source-level information (identifiers, line numbers, types, etc.) present in the source programs from which BCG files are generated, keeping in mind that these programs could be written in a variety of languages.

Implicit state spaces. In the terminology of CADP, an *implicit* state space is a state-transition graph defined *comprehensively*, meaning that only the initial state and the *post* transition function are given, such that (a fragment of) the graph is progressively explored and discovered on demand, depending on the verification goals. Handling implicit state spaces properly is a prerequisite for on-the-fly verification.

In addition to BCG, which only applies to explicit state spaces, CADP provides OPEN/CÆSAR [41], a software framework for implicit state spaces, which enforces modularity by clearly separating language-dependent aspects (i.e., compiler algorithms) from language-independent aspects (i.e., state-space exploration algorithms). OPEN/CÆSAR is organized around three components: the *graph module* (which encapsulates all language-dependent aspects, typically code generated from a high-level source program to compute states and transitions), the *library module* (which provides useful generic data structures, e.g., stacks, tables, hash functions, etc.), and the *exploration module* (which gathers language-independent aspects, typically simulation, test-case generation, and verification algorithms). All the internal details of the graph module are hidden behind a programming interface, which provides an abstraction for states and transition labels (making them available as opaque types) and implements the transition relation by means of a higher-order iterator.

Since the introduction of the OPEN/CÆSAR architecture in 1992, each of its three modules has been progressively extended. Regarding the graph module, only LOTOS was supported at first, but support for more languages (FSP [90], LOTOS NT [13], μCRL [66], SDL [78], and SYSTEMC/TLM [112]), more automata-based formalisms (BCG, EXP [84], FC2 [9], and SEQ [50]) and more external tools (IF [11], KRONOS [128], LTSMIN [7], TorX [3]) has been added, either by our or other research teams. Regarding the library module, its data structures and algorithms have been continuously optimized and enriched. Regarding the exploration module, many OPEN/CÆSAR tools have been developed for simulation, random execution, model checking, equivalence checking, and test case generation. The merits of the OPEN/CÆSAR modular architecture are recognized (e.g., in [7]: "*the* OPEN/CÆSAR *interface has been underlying the success of the* CADP *toolkit*") and a similar architecture can be found in several recent model checkers for concurrent systems, such as DIVINE[5] [2], LTSMIN[6] [7], and PAT[7] [88].

Boolean equation systems (BESs [89])**.** These are a useful low-level formalism for expressing analysis problems on LTSs, i.e., model checking, equivalence checking, partial order reductions, test case generation, and behavioral adaptation. A BES is a collection of equation blocks, each defining a set of Boolean variables (left-hand sides) by propositional formulas (right-hand sides). All equations in a block have the same fixed point sign: either minimal (μ) or maximal (ν). BESs can be represented as *Boolean graphs* [1] and are closely related to game graphs [116] and parity games [115]. Below is an example of BES in which block B_0 depends on variable Y_0 defined in block B_1:

$$(B_0)\ \left\{\begin{array}{l} X_0 =_\nu X_1 \wedge X_2 \\ X_1 =_\nu X_0 \vee X_1 \vee X_2 \\ X_2 =_\nu Y_0 \wedge X_3 \\ X_3 =_\nu \mathsf{true} \end{array}\right\}$$

$$(B_1)\ \left\{\begin{array}{l} Y_0 =_\mu Y_1 \vee Y_2 \\ Y_1 =_\mu \mathsf{false} \\ Y_2 =_\mu Y_2 \wedge Y_3 \\ Y_3 =_\mu Y_0 \vee Y_1 \vee Y_3 \end{array}\right\}$$

[4] http://cadp.inria.fr/resources/benchmark_bcg.html

[5] http://divine.fi.muni.cz/

[6] http://fmt.cs.utwente.nl/tools/ltsmin

[7] http://www.comp.nus.edu.sg/~pat

The CÆSAR_SOLVE library [94,95] of OPEN/CÆSAR contains a collection of linear-time algorithms for solving alternation-free BESs using various exploration strategies of its underlying Boolean graph (depth-first search, breadth-first search, etc.). The resolution works on the fly, the BES being constructed (e.g., from the evaluation of a temporal logic formula on an LTS, or from the comparison of two LTSs) at the same time it is solved, new equations being added to the BES and solved as soon as they are discovered. All the algorithms of CÆSAR_SOLVE can generate diagnostics, i.e., compute a minimal (in the sense of graph inclusion) Boolean subgraph explaining why a given Boolean variable is true or false [93].

New strategies have been added to CADP 2011 for solving conjunctive BESs (arising from equivalence checking) and disjunctive BESs (arising from model checking), keeping in memory only the vertices (and not the edges) of the Boolean graphs. Currently, CADP 2011 offers nine resolution strategies, which can solve BESs containing 10^7 variables in ten minutes on a machine with a 2 GHz CPU. Recently, a new linear-time algorithm generalizing the detection of accepting cycles in Büchi automata was added [101], which serves for model checking fairness properties. For testing and benchmarking purposes, CADP 2011 provides the new BES_SOLVE tool, which can evaluate BESs entirely constructed and stored in (gzipped) files, or built on the fly randomly according to fourteen parameters (number of variables, equation length, percentage of disjunctive and conjunctive operators, etc.).

It is known [87] that BESs can encode HORNSAT, a particular case of the SAT problem. Thus, CÆSAR_SOLVE can be used to solve large instances of HORNSAT (containing billions of variables) in linear-time.

Parameterized Boolean equation systems. CADP 2011 also uses internally the PBES (Parameterized BES) model [92], which extends the BES model by adding typed data parameters and arbitrary Boolean expressions over these parameters. For instance, the following PBES checks whether $factorial(m) = n$:

$$X(m,n) =_\mu \begin{pmatrix} (n > 0) & \wedge \\ \big((m = 0) \Rightarrow (n = 1)\big) & \wedge \\ \big((m > 0) \Rightarrow n\%m = 0 \wedge X(n/m, m-1)\big) & \end{pmatrix}$$

The PBES model was originally invented as a means of representing the model checking of MCL formulas (μ-calculus extended with typed data), implemented in the EVALUATOR 4.0 model checker now available in CADP 2011 (see Section 5). Recently, this model received much attention from the model checking community [67], which investigates two approaches: symbolic resolution or instantiations towards BESs followed by on-the-fly resolution, the latter being somehow close to SAT-solving. Beyond verification, PBESs can express other problems such as evaluation of parameterized Horn-clauses or DATALOG queries over data-bases [87].

3 Specification languages

A major addition to CADP 2011 compared with earlier versions is the support for several specification languages, while previously only LOTOS was supported.

3.1 Support for the LOTOS language

LOTOS [76] is a formal specification language standardized by ISO to describe communication protocols. It is composed of two different languages in one: a data part, based on algebraic abstract data types, and a control part, which is a process calculus combining the best features of CCS [104], CSP [73], and CIRCAL [103]. For this reason, CADP provides two LOTOS compilers, both sharing a common front-end.

Compiling the data part. The CÆSAR.ADT compiler [39,59] translates the data part of a LOTOS program (i.e., a collection of sorts, constructors, and functions defined by algebraic equations) into executable C code. The translation aims at verification efficiency, by first optimizing memory (which is essential for state space exploration, where every bit counts), then time. The compiler automatically recognizes certain classes of the usual types (natural numbers, enumerations, tuples, etc.), which are implemented optimally. The algebraic equations of LOTOS are translated using a pattern-matching compilation algorithm for rewrite systems with priority. This required deviations from standard LOTOS (e.g., introducing a distinction between constructor and non-constructor operations, and turning the algebraic equations into a system of rewrite rules with priorities), which were found to be acceptable in practice. Amusingly, most of the compiler is itself written using LOTOS abstract data types, so CÆSAR.ADT is used to bootstrap itself.

The version of CÆSAR.ADT included in CADP 2011 enables values of complex types (such as tuples, unions, lists, trees, strings, sets, etc.) to be represented "canonically", meaning that these values are stored in tables, represented in normal form as table indexes and thus are stored only once in memory. A technical challenge was to make this feature optional: the user can selectively store certain types in tables, while other types remain implemented as before.

Compiling the control part. The CÆSAR compiler [55,54] translates an entire LOTOS program (reusing the C code generated by CÆSAR.ADT) into C code that can be used either for generating an explicit LTS (encoded in the BCG format) or an implicit LTS (represented using the OPEN/CÆSAR programming interface), or for rapid prototyping (using the EXEC/CÆSAR programming interface, which enables the connection with a real-world environment). The subset of LOTOS accepted by CÆSAR

must obey certain constraints, which forbid unbounded dynamic creation of processes and non-terminal recursion in process calls; practically, these constraints are acceptable in most cases.

The translation is done using several intermediate steps, so as to perform, for efficiency reasons, as many computations as possible at compile-time. The LOTOS program is first translated into a simplified language named SUBLOTOS, then into a (hierarchical) Petri net extended with atomic transitions, typed local/global variables, and arbitrary combinations of conditions and actions attached to Petri net transitions. This Petri net is then simplified by applying a collection of optimizations on its control and data flows, and finally transformed into C code. CÆSAR can generate C code for different purposes, namely an OPEN/CÆSAR graph module, an optimized stand-alone program for LTS generation, or C code for co-simulation and rapid prototyping according to the EXEC/CÆSAR framework [60].

In addition to various bug fixes, the version of CÆSAR included in CADP 2011 delivers increased performance, particularly by introducing dynamically resizable state tables and by optimizing the generated C code for the amount of physical memory available. Also, the reduction techniques based on data flow analysis [54], which typically reduce state spaces by several orders of magnitude, have been enhanced by applying data-flow optimizations iteratively, following the hierarchical structure of the Petri net: for 22% of the benchmarks, the number of states is divided by 2.4 on average (on certain benchmarks, it is divided by 25).

3.2 Support for the FSP language

FSP (*Finite State Process*) is a concise algebraic notation for concurrent processes [90], supported by the LTSA (*Labelled Transition System Analyser*)[8] verification tool designed at Imperial College (London, United Kingdom). FSP and LTSA are particularly suited for students to practice with academic examples.

Although FSP and LOTOS share many fundamental concepts, they differ slightly in their expressiveness. On the one hand, FSP provides a priority operator that has no equivalent in LOTOS. On the other hand, LOTOS enables abstract data types to be defined by the user, while FSP provides Booleans, integers, labels, and predefined numeric functions only. Also, LOTOS enables sequential and parallel composition operators to be combined with only few restrictions, while FSP imposes a strict separation between sequential and parallel processes, so that parallel processes cannot be composed in sequence.

CADP 2011 supports the FSP language, following the translation approach of [85], implemented in two new tools. The FSP2LOTOS tool translates each sequential FSP process into a LOTOS process, and each parallel FSP process into an EXP [84] network of communicating processes with priorities. The FSP.OPEN tool provides a transparent interface between FSP and the OPEN/CÆSAR environment, which thus enables every FSP specification to be explored using any tool in the OPEN/CÆSAR exploration module. These tools have been validated on more than 1000 FSP examples, for which the LTSs generated by CADP were found to be strongly bisimilar to those generated by LTSA.

For the FSP user community, CADP 2011 brings the following advantages: it can handle FSP programs with non-guarded process recursion; it can handle larger FSP programs than LTSA, due to the particular attention to performance issues in CADP and to the support of 64-bit architectures, whereas LTSA suffers from Java's 32-bit limitations; finally, CADP offers many tools that complement the functionalities provided by LTSA.

3.3 Support for the LOTOS NT language

A major new feature of CADP 2011 is the support of LOTOS NT [13], a specification language derived from the ISO standard E-LOTOS [77]. LOTOS NT is an attempt [56,42,43] at merging the most salient features of process calculi (concurrency, abstraction, congruence results) into mainstream programming languages (imperative and functional languages for sequential programming). Contrary to LOTOS, which gathers two different languages into one, LOTOS NT exhibits a single unified language, in which the data part can be seen as a subset of the control part (i.e., functions are a particular case of processes): absence of such a nice symmetry in LOTOS is a drawback and a cause of its steep learning curve.

LOTOS NT has convenient features that LOTOS is lacking: it has a set of predefined data types (Booleans, natural numbers, integers, reals, characters, and strings); it provides short-hand notations for lists, sets, arrays, intervals, and predicate types; it eases the definition of inductive types by automatically generating common operations (equality and order relations, field accessors, etc.); it enables typing of communication channels [40]; it introduces the notion of modules. Similar to the LOTOS compilers of CADP, LOTOS NT can import hand-written, external C code that implements LOTOS NT types and functions; under some conditions, it is also possible to combine LOTOS and LOTOS NT code into the same specification.

As an example, Figures 1 and 2 give an excerpt of a LOTOS NT model of the MCS queue lock [102, Figure 5] for N processes trying to access a common resource; this model is part of a study of mutual exclusion protocols [99]. The MCS queue lock ensures mutual exclusion by means of a queue of waiting processes implemented in shared memory. A global variable (modeled as process "Lock") contains the index of the last element of the queue (or "nil" if the resource is free); this variable is

[8] http://www.doc.ic.ac.uk/ltsa

```
type Index is                                   -- interval type
  range 0 .. N of Nat with "==", "!="
end type

function nil: Index is
  return Index (0)
end function

type Pid is          -- valid Process IDentifier (predicate type)
  pid: Index where pid != nil
end type

type Qnode is
  Qnode (next: Index, locked: Bool) with "get", "set"
end type

type Memory is                  -- shared memory (array type)
  array [ 1 .. N ] of Qnode
end type

type Operation is        -- memory access (enumerated type)
  Read_next, Read_locked, Write_next, Write_locked,
  Fetch_and_Store, Compare_and_Swap
end type

channel CS_Access is      -- channel for Pid communication
  (Pid)
end channel

channel Memory_Access is        -- access to shared memory
  (Operation, Pid, Index, Pid),          -- read/write field next
  (Operation, Pid, Bool, Pid)          -- read/write field locked
end channel

channel Lock_Access is             -- access to global variable
  (Operation, Index, Index, Pid),              -- fetch-and-store
  (Operation, Index, Index, Bool, Pid)  -- compare-and-swap
end channel

process P [NCS, ENTER, LEAVE: CS_Access,
           L: Lock_Access, M: Memory_Access]
          (pid: Pid) is
  loop
    NCS (pid);
    acquire [L, M] (pid);
    ENTER (pid); LEAVE (pid);
    release [L, M] (pid)
  end loop
end process

process acquire [L: Lock_Access, M: Memory_Access]
                (pid: Pid) is
  var predecessor: Index, locked: Bool in
    M (Write_next, pid, nil, pid);
    L (Fetch_and_Store, ?predecessor, Index (pid), pid);
    if (predecessor != nil) then
      M (Write_locked, pid, true, pid);
      M (Write_next, Pid (predecessor), Index (pid), pid);
      loop L in
        M (Read_locked, pid, ?locked, pid);
        if not (locked) then break L end if
      end loop
    end if
  end var
end process

process release [L: Lock_Access, M: Memory_Access]
                (pid: Pid) is
  var next: Index, swap: Bool in
    M (Read_next, pid, ?next, pid);
    if next == nil then
      L (Compare_and_Swap, Index (pid), nil, ?swap, pid);
      if swap == false then
        loop L in
          M (Read_next, pid, ?next, pid);
          if next != nil then break L end if
        end loop;
        M (Write_locked, Pid (next), false, pid)
      end if
    else
      M (Write_locked, Pid (next), false, pid)
    end if
  end var
end process

process Lock [L: Lock_Access] is
  var i, new_i, j: Index in
    i := nil;
    loop
      select
        L (Fetch_and_Store, i, ?new_i, ?any Pid);
        i := new_i
      []
        L (Compare_and_Swap, ?j, ?new_i, true, ?any Pid)
          where i == j;
        i := new_i
      []
        L (Compare_and_Swap, ?j, ?new_i, false, ?any Pid)
          where i != j
      end select
    end loop
  end var
end process

process Memory [M: Memory_Access] is
  var m: Memory, pid: Pid, next: Index, locked: Bool in
    m := Memory (Qnode (nil, false));
    loop
      select
        M (Read_next, ?pid, ?next, ?any Pid)
          where next == m[Nat (pid)].next
      []
        M (Read_locked, ?pid, ?locked, ?any Pid)
          where locked == m[Nat (pid)].locked
      []
        M (Write_next, ?pid, ?next, ?any Pid);
        m[Nat (pid)] := m[Nat (pid)].{next => next}
      []
        M (Write_locked, ?pid, ?locked, ?any Pid);
        m[Nat (pid)] := m[Nat (pid)].{locked => locked}
      end select
    end loop
  end var
end process
```

Fig. 1. LOTOS NT model of the MCS queue lock (excerpt — an instantation of the MCS queue lock for five processes is shown in Figure 2)

modifiable by atomic *fetch-and-store* and *compare-and-swap* operations. The elements of the queue are pairs of the index of the next element and a Boolean flag indicating whether the resource is still locked by some preceding process; all elements are modeled as a shared array "m", accesses to which are managed by process "Memory". To acquire the lock, process "P" with identifier "pid" modifies the index of the last element to point to "m[pid]", initialized to "(nil, false)", and then waits until the resource is released by the immediately preceding process (by setting the flag to true). Each access to a shared variable is modeled by a rendezvous communication on one of the gates "L" or "M", using four or five offers (i.e., data parameters): in all cases, the first offer defines the operation to be executed, and the last offer indicates the executing process.

The feedback received about LOTOS NT from both academia and industry is highly positive: it is observed that people quickly start writing meaningful LOTOS NT specifications without the need for a long prior training. As of January 2010, the VASY team has switched from LOTOS to LOTOS NT for all its modeling activities, and LOTOS NT is used internally in companies such as Bull, CEA/Leti, and STMicroelectronics.

CADP 2011 includes a set of tools (LPP preprocessor, LNT2LOTOS translator, and LNT.OPEN connector to OPEN/CÆSAR) that implement LOTOS NT by translation to LOTOS, which enables one to reuse the CÆSAR and CÆSAR.ADT compilers to analyze and execute LOTOS NT specifications. To reduce the translation complexity, many semantic checks are deferred to the CÆSAR.ADT and CÆSAR compilers that will run on the generated, possibly incorrect LOTOS code.

The translation of LOTOS NT data part into LOTOS (which is, to some extent, the reverse of the translation performed by CÆSAR.ADT) requires compilation of functions defined in imperative-style into rewrite systems with priorities. It reuses an existing algorithm [108] for translating a subset of the C language into Horn clauses, but largely extends this algorithm to handle reference-passing parameters, pattern matching ("**case**" statements), loop interruptions ("**break**" statements), multiple "**return**" statements within function bodies, uncatchable exceptions ("**raise**" statements), and overloading of function names.

The translation of the LOTOS NT control part into LOTOS process algebraic terms borrows from a prior translation of CHP into LOTOS [53], which was adapted and optimized for LOTOS NT. The translation is tricky because LOTOS is much less "regular" than LOTOS NT for certain aspects (sequential composition, functionality typing for process termination) and because LOTOS lacks certain concepts (graphical parallel composition [57], type checking for communication channels). Surprisingly, the state spaces generated from LOTOS NT programs are in general not larger than those generated from "equivalent" LOTOS programs, due to the precise analysis and sharing of program continuations during the translation.

```
process Protocol [NCS, ENTER, LEAVE: CS_Access,
                  L: Lock_Access, M: Memory_Access] is
  par M, L in
    par      -- processes trying to access the critical section
      P [NCS, ENTER, LEAVE, L, M] (Pid (1))
    ||
      P [NCS, ENTER, LEAVE, L, M] (Pid (2))
    ||
      P [NCS, ENTER, LEAVE, L, M] (Pid (3))
    ||
      P [NCS, ENTER, LEAVE, L, M] (Pid (4))
    ||
      P [NCS, ENTER, LEAVE, L, M] (Pid (5))
    end par
  ||
    par                                  -- shared variables
      Lock [L] || Memory [M]
    end par
  end par
end process
```

Fig. 2. Instantiation of the MCS queue lock for five processes

3.4 Support for the EXP language

EXP is a language for describing a network of communicating LTSs, represented as a set of BCG files. In its current version, 2.0 [84], EXP offers an expressive syntax for handling compositions of processes in various languages. It provides the parallel composition, hiding, renaming, and cutting operators of CCS [104], CSP [111], LOTOS [76], E-LOTOS [77], LOTOS NT, and μCRL [66], as well as synchronization vectors and priorities. Hiding, renaming, and cutting operators are generalized, so that gate and/or label sets can be represented using regular expressions.

Figure 3 illustrates an EXP network of LTSs corresponding to an instance of the MCS queue lock for five processes, where only the gates "ENTER" and "LEAVE" are made observable. Each of "Pi.bcg" ($i \in$ 1..5), "Lock.bcg", and "Memory.bcg" is a BCG file encoding the LTS corresponding respectively to one of the five concurrent processes or to a shared variable.

3.5 Support for other languages

Numerous other languages have been connected to CADP 2011. Figure 4 gives a global picture; dark grey boxes indicate the languages and software components included in CADP 2011; light grey boxes indicate the languages for which VASY has developed translators and connections to CADP 2011, these translators being distributed separately from CADP 2011; arcs are labeled

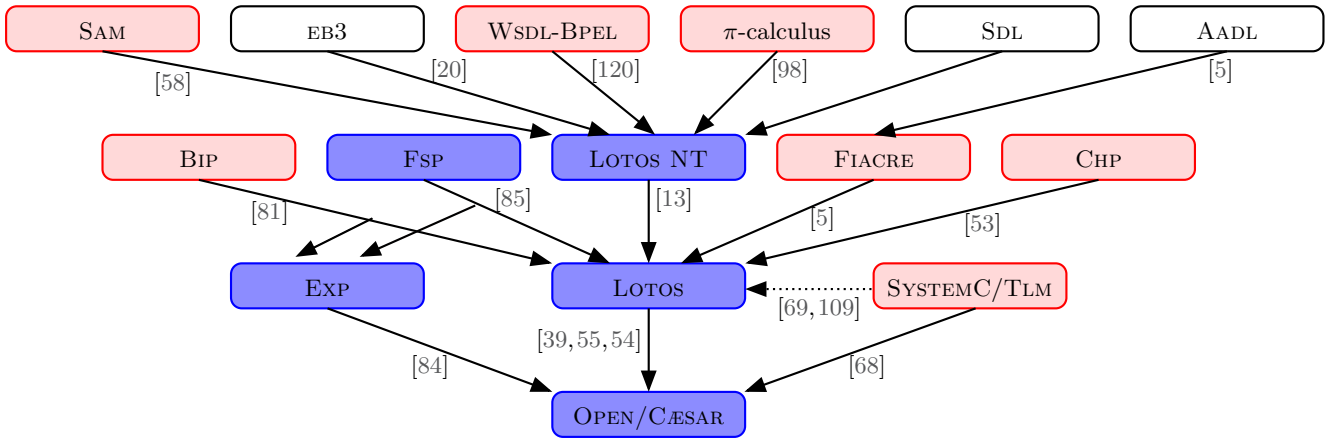

Fig. 4. Connection of the input languages of CADP 2011

```
hide all but "ENTER", "LEAVE" in
 par M, L in
  par in
   "P1.bcg" || "P2.bcg" || "P3.bcg" || "P4.bcg" || "P5.bcg"
  end par
 ||
  par in
   "Lock.bcg" || "Memory.bcg"
  end par
 end par
end hide
```

Fig. 3. MCS queue lock for five processes in the EXP 2.0 language

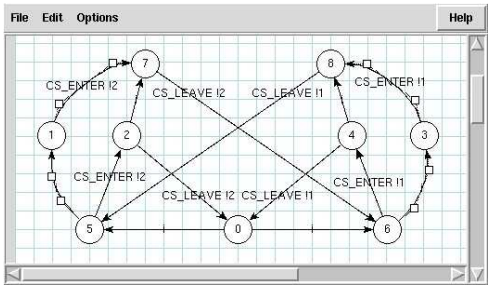

Fig. 5. BCG_EDIT

with bibliographic references; arcs without labels correspond to work in progress; dotted arcs correspond to manual translations.

These translation approaches have the merit of unifying the landscape of concurrent languages, which is currently scattered among multiple, incompatible proposals. Also, they significantly help to progress the development of the LOTOS and LOTOS NT compilers, by providing VASY with many programs, different from those that humans usually write; these programs reveal compiler mistakes and suggest new compiler optimizations for state space reduction and/or better performance.

4 Visual Checking

CADP contains tools to visualize explicit and implicit state spaces.

For explicit state spaces, the BCG_DRAW and BCG_EDIT (see Figure 5) tools enable LTSs encoded in the BCG format to be visualized and edited interactively.

For implicit state spaces, the OCIS (*Open/Cæsar Interactive Simulator*, see Figure 6) tool enables step-by-step simulation with backtracking. Simulation scenarios, which are trees describing the execution paths followed by the user, can be stored in the BCG format and reloaded for further exploration.

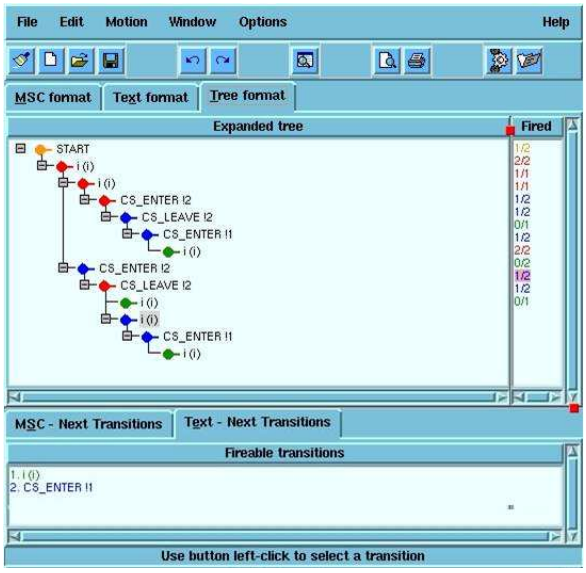

Fig. 6. Open/Cæsar Interactive Simulator

5 Model Checking

CADP contains three model checkers operating on explicit and implicit LTSs.

XTL (*eXecutable Temporal Language*) [96] is a functional language dedicated to the exploration and querying of an explicit LTS encoded in the BCG format. XTL handles (sets of) states, labels, and transitions as basic data types, enabling temporal logic operators to be implemented using their fixed point characterizations. Temporal logic operators can be mixed with non-standard properties (e.g., counting states, transitions, etc.) and, more generally, with arbitrary computations described as recursive functions exploring the LTS. XTL specifications can include reusable libraries of operators (15 such libraries are available in CADP) and can also be interfaced with external C code for specific computations.

EVALUATOR 3.x [100] evaluates formulas of RAFMC (the regular alternation-free μ-calculus) on an implicit LTS on the fly. RAFMC incorporates the PDL [37] modalities containing regular formulas over transition sequences, which are much more concise and intuitive than their fixed point counterparts: for instance, safety properties are simply expressed using the modality "[R] **false**", which forbids the transition sequences characterized by the regular formula R. The tool works by reformulating the model checking problem as a BES resolution, which is performed using the linear-time local algorithms of the CÆSAR_SOLVE library [95]. According to the shape of the formula, the most memory-efficient algorithm of the library is selected automatically: e.g., all formulas of ACTL [105] or PDL are evaluated using algorithms optimized to store only states, and not transitions, of the LTS. The tool produces examples and counterexamples, which are general LTS subgraphs (i.e., may contain branches and/or cycles), and also enables the definition of reusable libraries of property patterns, such as [30][9].

EVALUATOR 4.0 [101] is a new model checker handling formulas written in MCL (*Model Checking Language*), which conservatively extends RAFMC with two kinds of features. First, MCL adds data-handling mechanisms to parse and exploit structured transition labels (containing a channel/gate name and a list of values exchanged), generated from value-passing specification languages. MCL contains action predicates with value extraction, fixed point operators parameterized with data values, quantifiers over finite data domains, regular formulas extended with counters, and constructs inspired from functional programming languages ("**let**", "**if-then-else**", "**case**", "**while**", "**repeat**", etc.).

Second, MCL adds fairness operators, inspired from those of PDL-Δ [117], which characterize complex, unfair cycles consisting of infinite repetitions of regular subsequences. These operators belong to $L\mu_2$, the μ-calculus fragment of alternation depth two and were shown to subsume CTL* [125]. Although $L\mu_2$ has, in the worst case, a quadratic model checking complexity [31], the fairness operators of MCL are evaluated in linear-time using an enhanced resolution algorithm of CÆSAR_SOLVE [101].

We illustrate below the features of MCL by means of two data-based temporal properties related to the MCS protocol described in Figure 1. The first property expresses that processes access the shared resource in mutual exclusion, i.e., after a process i entered its critical section, it is impossible that another process j also enters its critical section as long as process i does not leave its critical section. This can be expressed in MCL using a necessity modality that forbids the occurrence of undesirable execution sequences:

```
[ true* .
  { ENTER ?i:Nat } .
  ( not { LEAVE !i } )* .
  { ENTER ?j:Nat where j <> i }
] false
```

Note how the process identifier i is captured by the first action predicate and is reused later in the formula in the "**where**" clause. This formula is fully parametric, in the sense that it does not depend on the number N of processes, and therefore it can be reused as it is for verifying any instantiation of the MCS.

The second property specifies the absence of starvation for each process i, i.e., the absence of cycles on which every process executes at least one action, but process i does not enter its critical section. This can be expressed in MCL by using the saturation operator "[R]**-**|", which forbids the infinite repetition of a transition sequence characterized by the regular formula R:

```
[ true* ] forall i:Nat among {1...N} .
  [ for j:Nat from 1 to N do
      ( not { ENTER ... !i } )* .
      { ?G:String ... !j
        where (j = i) implies (G <> "ENTER") }
    end for
  ]-|
```

Note the usage of the "**for**" regular formula to describe the concatenation of N subsequences corresponding to each process j. The fact that the formula uses a precise order of the occurrence of actions executed by processes j (from 1 to N) does not restrict its generality, because the absence of a cycle containing a particular ordering of the actions is equivalent to the absence of cycles containing *any* ordering of these actions (this is similar to the counting construction used for transforming a generalized Büchi automaton into a standard one).

[9] http://cadp.inria.fr/resources/evaluator/rafmc.html

6 Equivalence checking

Equivalence checking is useful to guarantee that some properties verified on one graph are also satisfied by another. Alternatively, equivalence checking can be used to minimize a graph by collapsing its equivalent states. Concurrency theory produced many graph equivalence relations, including strong bisimulation [104] and branching bisimulation [123], as well as stochastic/probabilistic extensions of strong and branching bisimulations (which take into account the notion of *lumpability* [80]) for models combining features from LTSs and Markov chains. From the beginning, equivalence checking has been a key feature of CADP, first with the ALDÉBARAN tool [32,35] and, since 1999, with the BCG_MIN 1.0 tool for minimization of explicit graphs using various partition-refinement algorithms ([79] for strong bisimulation, [65] for branching bisimulation, and [72] for stochastic/probabilistic extensions). The functionalities of these two tools have been progressively subsumed by improved tools, namely BCG_MIN 2.0 and BISIMULATOR, available in CADP 2011.

BCG_MIN 2.0 enables an explicit LTS to be minimized according to various equivalence relations. It implements partition-refinement algorithms based on the notion of state signature, originally designed for strong and branching bisimulations [6]. Intuitively, the signature of a state is the set of all couples "(transition label, block of the target state)" of the outgoing transitions (possibly following some compressed sequence of internal transitions in the case of branching bisimulation). Refinement of the state partition consists in dispatching states with different signatures to different blocks until the fixpoint has been reached, each block thus corresponding to a class of equivalent states. BCG_MIN 2.0 extends this algorithm to the stochastic/probabilistic extensions of strong and branching bisimulations, by incorporating lumpability in the computation of signatures.

For strong and branching bisimulations, tests on more than 8000 BCG graphs show that BCG_MIN 2.0 is 20 times faster and uses 1.3 times less memory than BCG_MIN 1.0. For stochastic/probabilistic bisimulations, BCG_MIN 2.0 is more than 500 (occasionally, 8500) times faster and uses 4 times less memory. Large graphs of more than 10^8 states and 10^9 transitions have been minimized in just a few hours, using less than 100 Gbytes RAM.

BISIMULATOR [4,95,97] compares an implicit LTS (usually, describing a *protocol*) with an explicit LTS (usually, describing the expected *service*) on the fly, by encoding the problem as a BES, which is solved using the linear-time local algorithms of the CÆSAR_SOLVE [95] library of CADP. This encoding generalizes and, because of optimizations applied on the fly depending on the LTS structure, outperforms the pioneering on-the-fly equivalence checking algorithms [35]. For typical cases, e.g., when the service LTS is deterministic and/or free of hidden/invisible (τ) actions, the tool automatically chooses an appropriate memory-efficient BES resolution algorithm, which stores only the states, and not the transitions.

BISIMULATOR implements seven equivalence relations (strong, observational, branching, $\tau^*.a$ [35], safety [8], trace, and weak trace [12]) and their associated preorders, thus being one of the richest on-the-fly equivalence checkers available. For non-equivalent LTSs, the tool can generate a counterexample, i.e., a directed acyclic graph containing the minimal set of transition sequences that, if traversed simultaneously in the two LTSs, lead to couples of non-equivalent states. Minimal-depth counterexamples can be obtained using breadth-first strategies for BES resolution. The tool is also equipped with on-the-fly reductions modulo τ-compression (collapse of τ-cycles) and τ-confluence (elimination of redundant interleavings), which preserve branching equivalence and can improve performance by several orders of magnitude.

Besides the classical usage as equivalence checker, BISIMULATOR is also employed by the OCIS graphical simulator of CADP in order to check on the fly whether a given execution scenario (represented as an LTS) is accepted or not by the specification under simulation.

7 Compositional Verification

Compositional verification is a way to avoid state explosion for the enumerative verification of complex concurrent systems which can be expressed as collections of sequential processes composed in parallel, either in a flat or hierarchical manner.

In its simplest form [32,91,113,127,118,119,122], compositional verification consists in replacing each sequential process by an *abstraction*, simpler than the original process but still preserving the properties to be verified on the whole system. In CADP, abstracting a process is done by minimizing its corresponding LTS modulo an appropriate equivalence or preorder relation (e.g., a bisimulation relation, such as strong or branching equivalence). If the system has a hierarchical structure, minimization can also be applied at every intermediate level in the hierarchy. This approach is possible as long as the equivalence or preorder relation is a congruence for the parallel composition operator.

Although this simple form of compositional verification has been applied successfully to some complex systems (e.g., [34,14] in the case of the LOTOS language), it may be counter-productive in some other cases: generating the LTS of each process separately may lead to state explosion, whereas the generation of the whole system of concurrent processes might succeed if processes constrain each other when composed in parallel.

This issue has been addressed by refined compositional verification approaches [63,17,126,18,19,64,82,16,61], which enable the LTS of each separate process to be generated by taking into account *interface constraints* (also known as *environment constraints* or *context constraints*). These constraints express the behavioral restrictions imposed on each process by synchronization with its neighbor processes. Taking into account the environment of each process enables the elimination of states and transitions that are not reachable in the LTS of the whole system. Depending on the approach, interface constraints can be either written by the user or generated automatically.

The CADP toolbox contains various tools dedicated to compositional verification, among which EXP.OPEN 2.0, PROJECTOR 3.0, and SVL play a central role (in addition to BCG_MIN already presented in Section 6).

EXP.OPEN 2.0 is a compiler that takes a network of communicating LTSs in the EXP language, compiles it into an intermediate model based on synchronization vectors, and finally produces an OPEN/CÆSAR graph module. EXP.OPEN 2.0 implements partial order reductions preserving either (stochastic) branching bisimulation, weak traces, or deadlocks.

PROJECTOR 3.0 implements behaviour abstraction [64, 82] of a process given as an OPEN/CÆSAR graph module, by taking into account interface constraints provided in the form of an LTS (the environment of the process) and a set of labels (on which the graph module interacts with its environment). The output is an LTS in the BCG format. If the interface constraints are written by the user, then validation predicates are generated in the LTS and checked afterwards by EXP.OPEN, when composing the resulting LTS with its environment.

SVL (*Script Verification Language*) [46,83] is both a high level language for expressing complex verification scenarios and a compiler dedicated to this language. SVL can be seen as a process algebra extended with operations on LTSs, e.g., minimization (also called reduction), abstraction, comparison, deadlock/livelock detection, etc., which orchestrates calls to the CADP tools.

The order in which processes are composed and minimized influences the efficiency of compositional verification. SVL thus implements various compositional verification strategies, in which the order can be either specified by the user (explicitly, or implicitly through the hierarchy of the concurrent processes), or determined automatically using heuristics such as "*smart reduction*" [28] (also enabling orders that are independent of the hierarchy of concurrent processes).

Figure 7 illustrates the compositional LTS generation of "mcs.bcg", an instance of the MCS queue lock for five processes where only the gates "ENTER" and "LEAVE" are made observable, followed by its comparison with the expected service modulo branching bisimulation. A diagnostic of the comparison is stored in file "mcs_diag_branching.bcg". The file "mcs.lnt" contains the code shown in Figure 1, where in addition each process "Pi [NCS, ENTER, LEAVE, L, M]" ($i \in 1..5$) is defined as the instance "P [NCS, ENTER, LEAVE, L, M] (Pid (i))" and where the process "Service" models the expected service. The value 7 of variable DEFAULT_SMART_LIMIT indicates that any combination of no more than seven processes can be selected at each step of compositional LTS generation. In this example, all seven processes are first minimized and then composed all together in a single step, producing an LTS with 408,762 states and 2,043,810 transitions. By contrast, direct generation produces an LTS that is about ten times larger (4,231,969 states and 21,159,845 transitions). The LTS "mcs.bcg" reduces finally to 651 states and 1295 transitions.

```
% DEFAULT_PROCESS_FILE="mcs.lnt"
% DEFAULT_SMART_LIMIT=7

"mcs.bcg" = smart branching reduction of
  hide all but ENTER, LEAVE in
    par M, L in
      par in
        P1 [NCS, ENTER, LEAVE, L, M]
      ||
        P2 [NCS, ENTER, LEAVE, L, M]
      ||
        P3 [NCS, ENTER, LEAVE, L, M]
      ||
        P4 [NCS, ENTER, LEAVE, L, M]
      ||
        P5 [NCS, ENTER, LEAVE, L, M]
      end par
    ||
      par in
        Lock [L]
      ||
        Memory [M]
      end par
    end par;

  "mcs_diag_branching.bcg" = branching comparison
    "mcs.bcg" == Service;
```

Fig. 7. Compositional generation and verification with SVL

8 Performance Evaluation

During the last decade, CADP has been enhanced for performance evaluation operating on extended Markovian models encoded in the BCG format (see details in [26]). In the example of the MCS queue lock, an interactive Markov chain [70] can be obtained by first inserting symbolic delay transitions "Lambda", "Mu", and "Nu" in the LOTOS NT model, for instance by parallel composition with a dedicated process "Latency" shown in

```
channel Latency is (Pid), (Operation) end channel

process Latency [NCS, ENTER, LEAVE: CS_Access,
                 L: Lock_Access, M: Memory_Access,
                 Lambda, Mu, Nu: Latency] is
  var pid: Pid, op: Operation in
    loop
      select
        NCS (?pid);
        Lambda (pid)
      []
        L (?op, ?any Index, ?any Index, ?pid);
        Mu (op)
      []
        L (?op, ?any Index, ?any Index, ?any Bool, ?pid);
        Mu (op)
      []
        M (?op, ?any Pid, ?any Index, ?pid);
        Mu (op)
      []
        M (?op, ?any Pid, ?any Bool, ?pid);
        Mu (op)
      []
        ENTER (?pid);
        Nu (pid)
      []
        LEAVE (?any Pid)                    -- no delay
      end select
    end loop
  end var
end process

process Main [NCS, ENTER, LEAVE: CS_Access,
              L: Lock_Access, M: Memory_Access,
              Lambda, Mu, Nu: Latency] is
  par NCS, ENTER, LEAVE, L, M in
    Protocol [NCS, ENTER, LEAVE, L, M]
  ||
    Latency [NCS, ENTER, LEAVE, L, M,
             Lambda, Mu, Nu]
  end par
end process
```

Fig. 8. Delay insertion for the Mcs queue lock

Figure 8, and subsequently instantiating these symbolic transitions with concrete rates by renaming the corresponding transitions in the generated Lts.

Besides Bcg_Min (which supports stochastic/probabilistic extensions of strong and branching bisimulations), the Exp.Open tool [84] now supports also the parallel composition of extended Markovian models, implementing maximal progress of internal transitions in choice with stochastic transitions. New tools have been added, namely Determinator [71], which eliminates stochastic nondeterminism in extended Markovian models on the fly using a variant of the algorithm presented in [29], and the Bcg_Steady and Bcg_Transient tools, which compute, for each state s of an extended Markovian model, the probability of being in s either on the long run (i.e., in the "steady state") or at each time instant t in a discrete set provided by the user. These tools can also compute the throughputs of transitions chosen by the user.

More recently, the new Cunctator on-the-fly steady-state simulator for extended Markovian models has been added to Cadp. The tool explores a random execution sequence in the model until a non-Markovian transition or a deadlock state is found, or the sequence length or virtual time (obtained by summing up the Markovian information present on transitions) reaches a maximum value specified by the user, or the user interactively halts the simulation. Upon termination, the throughputs of labeled transitions chosen by the user are displayed, together with information such as the number of τ-transitions encountered and the presence of nondeterminism (i.e., states with more than one outgoing τ-transition). The context of a simulation can be saved and restored for starting subsequent simulations, enabling one to implement convergence criteria (e.g., based on confidence intervals) by executing series of increasingly long simulations in linear time. For nondeterministic models, Cunctator selects between conflicting τ-transitions according to one of three scheduling policies (the first, the last, or a randomly chosen transition). Thus, launching simulations using different scheduling policies provides more insight about the stochastic behavior of the model. Compared to Bcg_Steady, which computes exact throughputs, Cunctator consumes less memory but achieving the same accuracy may require more time.

In the Mcs example, Bcg_Steady or Cunctator can be used to study the performance by computing the throughput of the transitions labeled with "ENTER" (see [99] for more details).

9 Parallel and Distributed Methods

Verification algorithms based on state space exploration have high computing and memory requirements and, thus, are often limited by the capabilities of one single sequential machine. However, the limits can be pushed out by new algorithms capable of exploiting processing resources offered by networks of workstations, clusters, grids, etc.

Cadp was among the first toolboxes to release tools for distributed model checking. The first step was to parallelize the state space construction, which is a bottleneck for verification because storing all reachable states requires a considerable amount of memory. For this purpose, the Distributor and Bcg_Merge tools [52,51] split the generation of an Lts across several machines, each machine building only a fragment of the entire Lts. Interestingly, certain essential Distributor features, such as the Pbg (*Partitioned* Bcg *Graph*) format and the graphical monitor that displays in real-time the

progress of generation across all the machines, have been replicated in competing verification toolsets.

The second step was the integration into CADP 2011 of a collection of new software tools (PBG_CP, PBG_MV, PBG_RM, and PBG_OPEN) to manipulate an LTS in the PBG format, and their connection to OPEN/CÆSAR.

The third step was the parallelization of on-the-fly verification itself. Therefore we designed a distributed version of the CÆSAR_SOLVE library to solve Boolean equation systems on the fly using several machines, thus enabling the development of parallel model and equivalence checkers.

10 Applications

As much as possible, we try to confront CADP with complex (often industrial) case studies in several domains.

Distributed systems. In collaboration with academic teams and industrial companies, we have used CADP for modeling and analysing various distributed middleware protocols and systems. We can mention the following:

- the consensus protocol used by the CO4 distributed knowledge system [106]
- a cluster file system built on top of ARIAS, a virtual shared memory across distributed machines running AIX [107]
- a dynamic reconfiguration protocol for agent-based applications [25]
- the deployment of the SCALAGENT platform for distributed JAVA agents [121], in the framework of a collaborative project aimed at the implementation of a safe, flexible architecture (based on JAVA components) enabling the remote management of uninterruptible power supplies
- the SYNERGY reconfiguration protocol [10]
- a self-configuration protocol for distributed applications in the cloud [114]

In addition to verification, the performance evaluation features of CADP have been applied to compare 27 mutual exclusion protocols in the shared memory setting [99].

High-performance computing. High-performance computing, particularly cache coherency protocols [14], has been a recurrent topic since 1996 in the framework of a collaboration with Bull, the European leader in this field. Notably, we have used CADP to verify critical parts of the TERA 10 super-computer (installed at CEA), which was Europe's most powerful computer in 2006.[10]

We have also used CADP to evaluate the performance of the SCSI-2 bus arbitration protocol [45] and of MPI implementations on CC-NUMA architectures [15].

Embedded systems on chip. The formal modeling, functional verification, and performance evaluation of hardware architectures with a high degree of asynchronous concurrency have been the subject of collaborations with Bull, STMicroelectronics, and the CEA/Leti laboratory (Grenoble, France) since 2004. A first step was the development of a translation from the CHP description language for asynchronous circuits into LOTOS, subsequently enabling verification with CADP [53]. We have used CADP to study the performance prediction of the XSTREAM architecture [27], a complex graphical processing unit described in SYSTEMC/TLM [44], and the dynamic task dispatcher of the Platform 2012 architecture [86].

Avionics. We have used CADP in the framework of collaborations with Airbus since 2005. In particular, we have developed a connection from the FIACRE intermediate language [5] to CADP. We have also verified several avionics systems using CADP, namely a ground/plane communication protocol based on TFTP [58], an equipment failure management protocol and an air-traffic control subsystem[11].

The CADP Web site also provides 40 properly packaged and documented case studies[12]. Other researchers from many institutions worldwide have also applied CADP to real case studies; a comprehensive database of 152 such case studies (from 1990 to 2011) published in the scientific literature is available on line[13]. The case studies cover application domains as diverse as the following (in alphabetical order): avionics, bioinformatics, business processes, cognitive systems, communication protocols, component-based systems, constraint programming, control systems, coordination architectures, critical infrastructures, cryptography, database protocols, distributed algorithms, distributed systems, e-commerce, e-democracy, embedded software, grid services, hardware design, hardware/software co-design, healthcare, human-computer interaction, industrial manufacturing systems, middleware, mobile agents, model-driven engineering, networks, object-oriented languages, performance evaluation, planning, radiotherapy equipment, real-time systems, security, sensor networks, service-oriented computing, software adaptation, software architectures, stochastic systems, systems on chip, telephony, transport safety, and Web services.

11 Conclusion

Concurrency theory is now 40-year old; formal methods are 35-year old; model checking verification is nearly 30-

[10] http://vasy.inria.fr/doc/inedit-69-en.pdf

[11] http://gforge.enseeiht.fr/docman/view.php/52/4316/B4-INRIA_VASY.pdf

[12] http://cadp.inria.fr/demos.html

[13] http://cadp.inria.fr/case-studies

year old. To push theoretical ideas into reality and to obtain new scientific results, significant effort must be put into software development and confrontation with industrial applications.

This was indeed the case with CADP 2011 which, besides all aforementioned new tools and major enhancements, also required large amounts of programming work: porting to various processors (Itanium, PowerPC, Sparc, x86, x64), operating systems (Linux, Mac OS X, Solaris, Windows) and C compilers (gcc 3, gcc 4, Intel, and Sun); careful code cleanup to remove all compiler and lint warnings, not only in the C code of the CADP tools themselves, but also in the C code that they may generate (this ensures that all compiler warnings received by end-users are related to some mistakes in their LOTOS or LOTOS NT code); significant documentation effort; intensive nonregression testing using thousands of LOTOS and LOTOS NT programs, BCG files, temporal logic formulas, Boolean equation systems, etc. together with a new tool named CONTRIBUTOR that will allow CADP users to send such test cases to the VASY team.

The relevance of these efforts and the maturity of CADP can be estimated from its dissemination and impact figures. CADP is distributed free of charge to universities and public research centers (academic licenses) and commercially to companies (professional licenses). As of July 2011, academic and commercial licences have been signed with more than 441 universities, public research institutes, and global corporations. Since July 2011, the academic distribution scheme was simplified: a signed license contract is no longer required, and CADP is now immediately available to (1) any scientist who can demonstrate employment in a university or a public research center by providing a valid professional e-mail address and Web page, and (2) any student who matches the above conditions or who is recommended by a professor.

As of January 2012, 152 case-studies have been tackled using CADP (see Section 10); 61 research software applications have been developed using CADP[14]; numerous academic courses are using CADP to teach concurrency[15]; the CADP user forum currently gathers more than 200 registered members with 1340 messages exchanged.

Regarding future work, we plan to develop a native LOTOS NT compiler, to connect even more concurrent languages to CADP, and add new verification tools that exploit massively parallel computing platforms. The latter research area is especially difficult, because it superposes the algorithmic complexities of verification and of distributed programming; yet this is the only way to exploit parallel computing resources, which are becoming pervasive.

[14] http://cadp.inria.fr/software

[15] http://cadp.inria.fr/training

Acknowledgments. The authors are grateful to H. Hermanns (Saarland University) and to I. Bellicot, S. Bouland, D. Champelovier, X. Clerc, N. Coste, J. Fereyre, Y. Genevois, Y. Guerte, C. Helmstetter, J. Henri, R. Hérilier, A. Kaufmann, R. Lacroix, N. Lépy, C. McKinty, S. Mériot, J. Merle, E. Oudot, L. Paternault, O. Ponsini, V. Powazny, S. Robert, G. Salaün, G. Smeding, J. Stöcker, D. Thivolle, M. Vidal, A. Wijs, and M. Zidouni (INRIA/VASY), who contributed during the past five years to software development, porting, testing, and/or maintenance of CADP. The authors are also grateful to the 40 scientists outside VASY who reported bugs and suggestions for enhancements during that period.

References

1. H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, Apr. 1994.
2. J. Barnat, L. Brim, M. Češka, and P. Ročkai". DiVinE: Parallel Distributed Model Checker (tool paper). In *Proceedings of Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology HiBi/PDMC 2010 (Twente, The Netherlands)*, pages 4–7. IEEE Computer Society Press, Sept. 2010.
3. A. Belinfante, J. Feenstra, R. G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal Test Automation: A Simple Experiment. In *Proceedings of the IFIP 12th International Workshop on Testing of Communicating Systems IWTCS'99 (Budapest, Hungary)*. Kluwer Academic, Sept. 1999.
4. D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2005 (Edinburgh, Scotland, UK)*, volume 3440 of *Lecture Notes in Computer Science*, pages 581–585. Springer Verlag, Apr. 2005.
5. B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. FIACRE: An Intermediate Language for Model Verification in the TOPCASED Environment. In *Proceedings of the 4th European Congress on Embedded Real-Time Software ERTS'08 (Toulouse, France)*. SIA (the French Society of Automobile Engineers), AAAF (the French Society of Aeronautic and Aerospace), and SEE (the French Society for Electricity, Electronics, and Information & Communication Technologies), Jan. 2008.
6. S. Blom and S. Orzan. Distributed state space minimization. *Software Tools for Technology Transfer*, 7(3):280–291, 2005.
7. S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and Symbolic Reachability. In *Proceedings of the 22nd International conference on Computer Aided Verification CAV 2010 (Edinburgh, UK)*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359. Springer Verlag, July 2010.
8. A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodríguez, and J. Sifakis. Safety for Branching Time Semantics. In *Proceedings of 18th ICALP*. Springer Verlag, July 1991.

9. A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*. Springer Verlag, Aug. 1996.
10. F. Boyer, O. Gruber, and G. Salaün. Specifying and Verifying the SYNERGY Reconfiguration Protocol with LOTOS NT/CADP. In *Proceedings of the 17th International Symposium on Formal Methods FM'2011 (Limerick, Ireland)*, volume 6664 of *Lecture Notes in Computer Science*, pages 103–117. Springer Verlag, June 2011.
11. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *Proceedings of World Congress on Formal Methods in the Development of Computing Systems FM'99 (Toulouse, France)*. Springer Verlag, Sept. 1999.
12. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *J. ACM*, 31(3):560–599, July 1984.
13. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4). INRIA/VASY, 149 pages, Sept. 2011.
14. 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 *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 (Kaiserslautern, Germany)*, pages 435–450. IFIP, Chapman & Hall, Oct. 1996. Full version available as INRIA Research Report RR-2958.
15. G. Chehaibar, M. Zidouni, and R. Mateescu. Modeling Multiprocessor Cache Protocol Impact on MPI Performance. In *Proceedings of the 2009 IEEE International Workshop on Quantitative Evaluation of Large-Scale Systems and Technologies QuEST'09 (Bradford, UK)*. IEEE Computer Society Press, May 2009.
16. K. H. Cheung. *Compositional Analysis of Complex Distributed Systems*. PhD thesis, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.
17. S. C. Cheung and J. Kramer. Enhancing Compositional Reachability Analysis with Context Constraints. In *Proceedings of the 1st ACM SIGSOFT International Symposium on the Foundations of Software Engineering (Los Angeles, CA, USA)*, pages 115–125. ACM Press, Dec. 1993.
18. S. C. Cheung and J. Kramer. Compositional Reachability Analysis of Finite-State Distributed Systems with User-Specified Constraints. In *Proceedings of the 3rd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (Washington, DC, USA)*, pages 140–150. ACM Press, Oct. 1995.
19. S. C. Cheung and J. Kramer. Context Constraints for Compositional Reachability. *ACM Transactions on Software Engineering Methodology TOSEM*, 5(4):334–377, Oct. 1996.
20. R. Chossart. Évaluation d'outils de vérification pour les spécifications de systèmes d'information. Mémoire maître ès sciences, Université de Sherbrooke, Canada, Mar. 2010.
21. E. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic. In *10th Annual Symposium on Principles of Programming Languages*. ACM, 1983.
22. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, Apr. 1986.
23. R. Cleaveland, T. Li, and S. Sims. The Concurrency Workbench of the New Century (Version 1.2). User's manual, July 2000.
24. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench. In *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 24–37. Springer Verlag, June 1989.
25. M. A. Cornejo, H. Garavel, R. Mateescu, and N. de Palma. Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications. In *Proceedings of the 3rd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems DAIS'2001 (Krakow, Poland)*, pages 229–242. IFIP, Kluwer Academic Publishers, Sept. 2001. Full version available as INRIA Research Report RR-4222.
26. N. Coste, H. Garavel, H. Hermanns, F. Lang, R. Mateescu, and W. Serwe. Ten Years of Performance Evaluation for Concurrent Systems Using CADP. In *Proceedings of the 4th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation ISoLA 2010 (Amirandes, Heraclion, Crete), Part II*, volume 6416 of *Lecture Notes in Computer Science*, pages 128–142. Springer Verlag, Oct. 2010.
27. N. Coste, H. Hermanns, E. Lantreibecq, and W. Serwe. Towards Performance Prediction of Compositional Models in Industrial GALS Designs. In *Proceedings of the 21th International Conference on Computer Aided Verification CAV'2009 (Grenoble, France)*, volume 5643 of *Lecture Notes in Computer Science*, pages 204–218. Springer Verlag, July 2009.
28. P. Crouzen and F. Lang. Smart Reduction. In *Proceedings of Fundamental Approaches to Software Engineering FASE'2011 (Saarbrücken, Germany)*, volume 6603 of *Lecture Notes in Computer Science*, pages 111–126. Springer Verlag, Mar. 2011.
29. D. D. Deavours and W. H. Sanders. An Efficient Well-Specified Check. In *Proceedings of the 8th International Workshop on Petri Nets and Performance Models PNPM'99 (Zaragoza, Spain)*, pages 124–133. IEEE Press, 1999.
30. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering ICSE'99 (Los Angeles, CA, USA)*, May 1999.
31. E. A. Emerson and C.-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Pro-*

ceedings of the 1st International Symposium on Logic in Computer Science LICS'86, pages 267–278, 1986.
32. J.-C. Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), May 1988.
33. J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, Aug. 1996.
34. J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodríguez, and J. Sifakis. A Toolbox for the Verification of LOTOS Programs. In *Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia)*, pages 246–259. ACM, May 1992.
35. J.-C. Fernandez and L. Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, pages 181–191. Springer Verlag, July 1991.
36. J.-C. Fernandez, J.-L. Richier, and J. Voiron. Verification of Protocol Specifications using the CESAR System. In *Proceedings of the 5th IFIP International Workshop on Protocol Specification, Testing and Verification (Moissac, France)*, pages 71–90. IFIP, North-Holland, June 1985.
37. M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *J. Comput. Syst. Sci.*, 18(2):194–211, Sept. 1979.
38. H. Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), Nov. 1989.
39. H. Garavel. Compilation of LOTOS Abstract Data Types. In *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, Dec. 1989.
40. H. Garavel. On the Introduction of Gate Typing in E-LOTOS. Rapport SPECTRE 94-3, VERIMAG, Grenoble, Feb. 1994. Annex D of ISO/IEC JTC1/SC21/WG1 N1314 Revised Draft on Enhancements to LOTOS and Annex C of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS.
41. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, Mar. 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.
42. H. Garavel. Défense et illustration des algèbres de processus. In *Actes de l'Ecole d'été Temps Réel ETR 2003 (Toulouse, France)*. Institut de Recherche en Informatique de Toulouse, Sept. 2003.
43. H. Garavel. Reflections on the Future of Concurrency Theory in General and Process Calculi in Particular. In *Proceedings of the LIX Colloquium on Emerging Trends in Concurrency Theory (Ecole Polytechnique de Paris, France), November 13–15, 2006*, volume 209 of *Electronic Notes in Theoretical Computer Science*, pages 149–164. Elsevier Science Publishers, Apr. 2008. Also available as INRIA Research Report RR-6368.
44. H. Garavel, C. Helmstetter, O. Ponsini, and W. Serwe. Verification of an Industrial SystemC/TLM Model using LOTOS and CADP. In *Proceedings of the 7th ACM-IEEE International Conference on Formal Methods and Models for Codesign MEMOCODE'2009 (Cambridge, MA, USA)*. IEEE Computer Society Press, June 2009.
45. H. Garavel and H. Hermanns. On Combining Functional Verification and Performance Evaluation using CADP. In *Proceedings of the 11th International Symposium of Formal Methods Europe FME'2002 (Copenhagen, Denmark)*, volume 2391 of *Lecture Notes in Computer Science*, pages 410–429. Springer Verlag, July 2002. Full version available as INRIA Research Report 4492.
46. H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. In *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, Aug. 2001. Full version available as INRIA Research Report RR-4223.
47. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, Aug. 2002. Also available as INRIA Technical Report RT-0254 (December 2001).
48. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proceedings of the 19th International Conference on Computer Aided Verification CAV'2007 (Berlin, Germany)*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer Verlag, July 2007.
49. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2011 (Saarbrücken, Germany)*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer Verlag, Mar. 2011.
50. H. Garavel and R. Mateescu. SEQ.OPEN: A Tool for Efficient Trace-Based Verification. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software SPIN'2004 (Barcelona, Spain)*, volume 2989 of *Lecture Notes in Computer Science*, pages 150–155. Springer Verlag, Apr. 2004.
51. H. Garavel, R. Mateescu, D. Bergamini, A. Curic, N. Descoubes, C. Joubert, I. Smarandache-Sturm, and G. Stragier. DISTRIBUTOR and BCG_MERGE: Tools for Distributed Explicit State Space Generation. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2006 (Vienna, Austria)*, volume 3920 of *Lecture Notes in Computer Science*, pages 445–449. Springer Verlag, Mar.–Apr. 2006.

52. H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada)*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234, Berlin, May 2001. Springer Verlag. Revised version available as INRIA Research Report RR-4341 (December 2001).
53. H. Garavel, G. Salaün, and W. Serwe. On the Semantics of Communicating Hardware Processes and their Translation into LOTOS for the Verification of Asynchronous Circuits with CADP. *Science of Computer Programming*, 74(3):100–127, Jan. 2009.
54. H. Garavel and W. Serwe. State Space Reduction for Process Algebra Specifications. *Theoretical Computer Science*, 351(2):131–145, Feb. 2006.
55. H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.
56. H. Garavel and M. Sighireanu. Towards a Second Generation of Formal Description Techniques – Rationale for the Design of E-LOTOS. In *Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems FMICS'98 (Amsterdam, The Netherlands)*, pages 187–230, Amsterdam, May 1998. CWI. Invited lecture.
57. H. Garavel and M. Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'99 (Beijing, China)*, pages 185–202. IFIP, Kluwer Academic Publishers, Oct. 1999.
58. H. Garavel and D. Thivolle. Verification of GALS Systems by Combining Synchronous Languages and Process Calculi. In *Model Checking Software, Proceedings of the 16th International SPIN Workshop on Model Checking of Software SPIN'2009 (Grenoble, France)*, volume 5578 of *Lecture Notes in Computer Science*, pages 241–260. Springer Verlag, June 2009.
59. H. Garavel and P. Turlier. CÆSAR.ADT : un compilateur pour les types abstraits algébriques du langage LOTOS. In *Actes du Colloque Francophone pour l'Ingénierie des Protocoles CFIP'93 (Montréal, Canada)*, 1993.
60. H. Garavel, C. Viho, and M. Zendri. System Design of a CC-NUMA Multiprocessor Architecture using Formal Specification, Model-Checking, Co-Simulation, and Test Generation. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 3(3):314–331, July 2001. Also available as INRIA Research Report RR-4041.
61. D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College of Science, Technology and Medicine — University of London — Department of Computer Science, Jan. 1999.
62. 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 *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 275–285. Springer Verlag, June 1989.
63. S. Graf and B. Steffen. Compositional Minimization of Finite State Systems. In *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, volume 531 of *Lecture Notes in Computer Science*, pages 186–196. Springer Verlag, June 1990.
64. S. Graf, B. Steffen, and G. Lüttgen. Compositional Minimisation of Finite State Systems using Interface Specifications. *Formal Aspects of Computation*, 8(5):607–616, Sept. 1996.
65. J. Groote and F. Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In *Proceedings of the 17th ICALP (Warwick)*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer Verlag, 1990.
66. J. F. Groote and A. Ponse. The Syntax and Semantics of μCRL. In *Algebra of Communicating Processes'94*, Workshops in Computing Series, pages 26–62. Springer Verlag, 1995.
67. J. F. Groote and T. A. C. Willemse. Parameterised Boolean Equation Systems. *Theoretical Computer Science*, 343:332–369, 2005.
68. C. Helmstetter. TLM.OPEN: a SystemC/TLM Front-End for the CADP Verification Toolbox. Workshop on Simulation Based Development of Certified Embedded Systems SBDCES'09 (Awaji Island, Hyogo, Japan), Oct. 2009.
69. C. Helmstetter and O. Ponsini. A Comparison of Two SystemC/TLM Semantics for Formal Verification. In *Proceedings of the 6th ACM-IEEE International Conference on Formal Methods and Models for Codesign MEMOCODE'2008 (Anaheim, CA, USA)*, pages 59–68. IEEE Computer Society Press, June 2008.
70. H. Hermanns. *Interactive Markov Chains and the Quest for Quantified Quality*, volume 2428 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.
71. H. Hermanns and C. Joubert. A Set of Performance and Dependability Analysis Components for CADP. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003 (Warsaw, Poland)*, volume 2619 of *Lecture Notes in Computer Science*, pages 425–430. Springer Verlag, Apr. 2003.
72. H. Hermanns and M. Siegle. Bisimulation Algorithms for Stochastic Process Algebras and their BDD-based Implementation. In *Proceedings of the 5th International AMAST Workshop ARTS'99 (Bamberg, Germany)*, volume 1601 of *Lecture Notes in Computer Science*, pages 244–265. Springer Verlag, May 1999.
73. C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.
74. G. J. Holzmann. *Design and Validation of Computer Protocols*. Software Series. Prentice Hall, 1991.
75. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
76. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Process-

ing Systems — Open Systems Interconnection, Genève, Sept. 1989.
77. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, Sept. 2001.
78. ITU-T. Specification and Description Language (SDL). ITU-T Recommendation Z.100, International Telecommunication Union, Genève, 1992.
79. P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.
80. J. G. Kemeny and J. L. Snell. *Finite Markov Chains*. Springer, 1976.
81. A. M. Khan. Connection of Compositional Verification Tools for Embedded Systems. Mémoire master 2 recherche, Université Joseph Fourier, Grenoble, June 2006.
82. J.-P. Krimm and L. Mounier. Compositional State Space Generation from LOTOS Programs. In *Proceedings of TACAS'97 Tools and Algorithms for the Construction and Analysis of Systems (University of Twente, Enschede, The Netherlands)*, volume 1217 of *Lecture Notes in Computer Science*, Berlin, Apr. 1997. Springer Verlag.
83. F. Lang. Compositional Verification using SVL Scripts. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002 (Grenoble, France)*, volume 2280 of *Lecture Notes in Computer Science*, pages 465–469. Springer Verlag, Apr. 2002.
84. F. Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. In *Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, volume 3771 of *Lecture Notes in Computer Science*, pages 70–88. Springer Verlag, Nov. 2005. Full version available as INRIA Research Report RR-5673.
85. F. Lang, G. Salaün, R. Hérilier, J. Kramer, and J. Magee. Translating FSP into LOTOS and Networks of Automata. *Formal Aspects of Computing*, 22(6):681–711, Nov. 2010.
86. E. Lantreibecq and W. Serwe. Model Checking and Cosimulation of a Dynamic Task Dispatcher Circuit Using CADP. In *Proceedings of the 16th International Workshop on Formal Methods for Industrial Critical Systems FMICS 2011 (Trento, Italy)*, volume 6959 of *Lecture Notes in Computer Science*, pages 180–195. Springer Verlag, Aug. 2011.
87. X. Liu and S. A. Smolka. Simple Linear-Time Algorithms for Minimal Fixed Points. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming ICALP'98 (Aalborg, Denmark)*, volume 1443 of *Lecture Notes in Computer Science*, pages 53–66. Springer Verlag, July 1998.
88. Y. Liu, J. Sun, and J. S. Dong. Developing Model Checkers Using PAT. In *Proceedings of the 8th International Symposium on Automated Technology for Verification and Analysis ATVA 2010 (Singapore)*, volume 6252 of *Lecture Notes in Computer Science*, pages 371–377. Springer Verlag, Sept. 2010.
89. A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. VERSAL 8, Bertz Verlag, Berlin, 1997.
90. J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2006 edition, Apr. 2006.
91. J. Malhotra, S. A. Smolka, A. Giacalone, and R. Shapiro. A Tool for Hierarchical Design and Simulation of Concurrent Systems. In *Proceedings of the BCS-FACS Workshop on Specification and Verification of Concurrent Systems (Stirling, Scotland)*, pages 140–152, Swindon, UK, July 1988. British Computer Society.
92. R. Mateescu. *Vérification des propriétés temporelles des programmes parallèles*. Thèse de Doctorat, Institut National Polytechnique de Grenoble, Apr. 1998.
93. R. Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In *Proceedings of 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2000 (Berlin, Germany)*, volume 1785 of *Lecture Notes in Computer Science*, pages 251–265. Springer Verlag, Mar. 2000. Full version available as INRIA Research Report RR-3861.
94. R. Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003 (Warsaw, Poland)*, volume 2619 of *Lecture Notes in Computer Science*, pages 81–96. Springer Verlag, Apr. 2003. Full version available as INRIA Research Report RR-4711.
95. R. Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 8(1):37–56, Feb. 2006. Full version available as INRIA Research Report RR-5948, July 2006.
96. R. Mateescu and H. Garavel. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98 (Aalborg, Denmark)*, pages 33–42. BRICS, July 1998.
97. R. Mateescu and E. Oudot. Improved On-the-Fly Equivalence Checking using Boolean Equation Systems. In *Proceedings of the 15th International SPIN Workshop on Model Checking of Software SPIN'2008 (Los Angeles, USA)*, volume 5156 of *Lecture Notes in Computer Science*, pages 196–213. Springer Verlag, Aug. 2008. Full version available as INRIA Research Report RR-6777.
98. R. Mateescu and G. Salaün. Translating Pi-Calculus into LOTOS NT. In *Proceedings of the 8th International Conference on Integrated Formal Methods IFM'2010 (Nancy, France)*, volume 6396 of *Lecture Notes in Computer Science*, pages 229–244. Springer Verlag, Oct. 2010.
99. R. Mateescu and W. Serwe. Model Checking and Performance Evaluation with CADP Illustrated on Shared-Memory Mutual Exclusion Protocols. *Science of Computer Programming*, 2012.
`http://dx.doi.org/10.1016/j.scico.2012.01.003`
100. R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free

Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, Mar. 2003.

101. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proceedings of the 15th International Symposium on Formal Methods FM'08 (Turku, Finland)*, volume 5014 of *Lecture Notes in Computer Science*, pages 148–164. Springer Verlag, May 2008.
102. J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.
103. G. J. Milne. CIRCAL and the Representation of Communication, Concurrency, and Time. *ACM Trans. Prog. Lang. Syst.*, 7(2):270–298, Apr. 1985.
104. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
105. R. D. Nicola and F. W. Vaandrager. *Action versus State Based Logics for Transition Systems*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, 1990.
106. C. Pecheur. Specification and Verification of the CO4 Distributed Knowledge System Using LOTOS. In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering ASE-97 (Incline Village, Nevada, USA)*, Nov. 1997.
107. C. Pecheur. Advanced Modelling and Verification Techniques Applied to a Cluster File System. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering ASE-99 (Cocoa Beach, Florida, USA)*. IEEE Computer Society, Oct. 1999.
108. O. Ponsini, C. Fédèle, and E. Kounalis. Rewriting of imperative programs into logical equations. *Science of Computer Programming*, 56(3):363–401, May – June 2005.
109. O. Ponsini and W. Serwe. A Schedulerless Semantics of TLM Models Written in SystemC via Translation into LOTOS. In *Proceedings of the 15th International Symposium on Formal Methods FM'08 (Turku, Finland)*, volume 5014 of *Lecture Notes in Computer Science*, pages 278–293. Springer Verlag, May 2008.
110. J.-P. Queille. *Le système CESAR : description, spécification et analyse des applications réparties*. Université Scientifique et Médicale de Grenoble, Grenoble, June 1982.
111. A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.
112. A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez. *Transaction Level Modeling in SystemC*. Open SystemC Initiative, 2005.
113. K. K. Sabnani, A. M. Lapone, and M. U. Uyar. An Algorithmic Procedure for Checking Safety Properties of Protocols. *IEEE Transactions on Communications*, 37(9):940–948, Sept. 1989.
114. G. Salaün, X. Etchevers, N. D. Palma, F. Boyer, and T. Coupaye. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In *Proceedings of the 27th Symposium On Applied Computing SAC'12 (Riva del Garda, Italy)*. ACM Press, 2012. To appear.
115. S. Schewe. Solving Parity Games in Big Steps. In *Proceedings of the 27th International Conference on Software Technology and Theoretical Computer Science FSTTCS'07 (New Delhi, India)*, volume 4855 of *Lecture Notes in Computer Science*, pages 449–460, Berlin, Dec. 2007. Springer Verlag.
116. P. Stevens and C. Stirling. Practical Model-Checking using Games. In *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 85–101, Berlin, Mar. 1998. Springer Verlag.
117. R. Streett. Propositional Dynamic Logic of Looping and Converse. *Information and Control*, (54):121–141, 1982.
118. K. C. Tai and V. Koppol. Hierarchy-Based Incremental Reachability Analysis of Communication Protocols. In *Proceedings of the IEEE International Conference on Network Protocols (San Francisco, CA)*, pages 318–325, Piscataway, NJ, Oct. 1993. IEEE Press.
119. K. C. Tai and V. Koppol. An Incremental Approach to Reachability Analysis of Distributed Programs. In *Proceedings of the 7th International Workshop on Software Specification and Design (Los Angeles, CA)*, pages 141–150, Piscataway, NJ, Dec. 1993. IEEE Press.
120. D. Thivolle. *Langages modernes pour la vérification des systèmes asynchrones*. Thèse de Doctorat, Université Joseph Fourier (Grenoble, France) and Universitatea Politehnica din Bucuresti (Bucharest, Romania), Apr. 2011.
121. F. Tronel, F. Lang, and H. Garavel. Compositional Verification Using CADP of the ScalAgent Deployment Protocol for Software Components. In *Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems FMOODS'2003 (Paris, France)*, volume 2884 of *Lecture Notes in Computer Science*, pages 244–260. Springer Verlag, Nov. 2003. Full version available as INRIA Research Report RR-5012.
122. A. Valmari. Compositional State Space Generation. In *Proceedings of Advances in Petri Nets*, volume 674 of *Lecture Notes in Computer Science*, pages 427–457. Springer Verlag, 1993.
123. R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.
124. C. West. A General Technique for Communication Protocol Validation. *IBM Journal of Research and Development*, pages 393–404, July 1978.
125. P. Wolper. A Translation from Full Branching Time Temporal Logic to One Letter Propositional Dynamic Logic with Looping. Unpublished manuscript, 1982.
126. W. J. Yeh. *Controlling State Explosion in Reachability Analysis*. PhD thesis, Software Engineering Research Center (SERC) Laboratory, Purdue University, Dec. 1993. Technical Report SERC-TR-147-P.
127. W. J. Yeh and M. Young. Compositional Reachability Analysis Using Process Algebra. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis, and Verification (SIGSOFT'91, Victoria, British Columbia, Canada)*, pages 49–59, New York, NY, Oct. 1991. ACM Press.

128. S. Yovine. Kronos: A verification tool for real-time systems. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 1(1/2):123–133, Oct. 1997.