CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes*

Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe

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

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 2010, which is the result of a considerable development effort spanning the last four 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 2010, including compilers for various formal specification languages, equivalence checkers, model checkers, performance evaluation tools, and parallel verification tools running on clusters and grids.

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

* 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).

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 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 Sect. 2). Enumerative techniques are based on a forward exploration of the transition relation between 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. Among the enumerative model checkers developed in the 80s, SPIN [33] and CADP are the two oldest that are still available on the latest 64-bit architectures and being used in an industrial setting. The principles underlying these two model checkers are a source of inspiration for other recent verification tools based on similar concepts.

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 [14], a compiler and explicit state space generator for the LOTOS language, and ALDÉBARAN [11, 13], an equivalence checker based on bisimulation minimization. Over the past 25 years, CADP has been continuously improved and extended [12, 19, 20]. This paper presents the latest release, CADP 2010 "*Zurich*", which currently contains 45 tools.

CADP offers now 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 scripting languages for describing elaborated verification scenarios. In addition, CADP 2010 brings deep changes with respect to previous releases, especially the support for many different specification languages.

This paper gives an overview of CADP 2010, highlighting new tools and recent enhancements. It is organized as follows. Sect. 2 presents the core semantic mod-

[1] http://vasy.inria.fr/cadp

els of CADP. Sect. 3 describes the three languages now supported by CADP and lists translations developed for other languages. Sect. 4, 5, 6, and 7 respectively present the CADP tools for model checking, equivalence checking, performance evaluation, and distributed verification. Finally, Sect. 8 summarizes the achievements and indicates directions for future work.

2 Architecture and Verification Technology

Compared to other explicit-state model checkers, CADP has the following principles and distinctive features (some of which were already present in inspiring tools rooted in concurrency theory, such as CWB [8] and CWB-NC [7]):

- 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 by 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 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) [31], 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, which has widely inspired recent competing model checkers developed in the 2000s. Early model checkers were "monolithic" in the sense that they tightly combined (1) the source language used to describe 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 2010 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-mode use, a scripting language named Svl [18] was designed, with user-friendly syntax and powerful verification strategies that make of Svl a unique feature of Cadp.

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. 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 2010 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*)[2], 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 different 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 [16], 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., verification

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

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 verification and state exploration 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 has been added, either by Vasy 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.

Boolean equation systems (Bess [39]). 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 [51] and parity games [50].

The Cæsar_Solve library [43] 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 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 [42].

New strategies have been added to Cadp 2010 for solving conjunctive Bess (arising from equivalence checking) and disjunctive Bes (arising from model checking), keeping in memory only the vertices (and not the edges) of the Boolean graphs. Currently, Cadp 2010 offers nine resolution strategies, which can solve Bess containing 10^7 variables in ten minutes. Recently, a new linear-time algorithm generalizing the detection of accepting cycles in Büchi automata was added [47], which serves for model checking fairness properties. For testing and benchmarking purposes, Cadp 2010 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.).

Parameterized Boolean equation systems. Cadp also uses internally the Pbes (Parameterized Bes) model [41], which extends the Bes model by adding typed data parameters and arbitrary Boolean expressions over these parameters. The Pbes model was originally invented as a means to represent the model checking of Mcl formulas (μ-calculus extended with typed data), implemented in the Evaluator 4.0 model checker now available in Cadp 2010 (see Sect. 4). Recently, this model received much attention from the model checking community, 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.

3 Specification languages

A major difference of Cadp 2010 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 [34] 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, Csp, and Circal. For this reason, Cadp provides two Lotos compilers, both sharing a common front-end.

Compiling the data part. The Cæsar.adt compiler [15, 28] 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 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. Amusingly, most of the compiler is written using Lotos abstract data types, so that Cæsar.adt is used to bootstrap itself.

The version of Cæsar.adt included in Cadp 2010 allows 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 [25, 24] 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 a C program, which is then compiled and executed.

In addition to various bug fixes, the version of CÆSAR included in CADP 2010 delivers increased performance, especially 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 [24], 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 [40], supported by the LTSA (*Labelled Transition System Analyser*) 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 allows abstract data types to be defined by the user, while FSP provides Booleans, integers, labels, and predefined numeric functions only. Also, LOTOS allows 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 2010 supports the FSP language, following the translation approach of [38], 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.OPEN [37] network of communicating processes with priorities. The FSP.OPEN tool provides a transparent interface between FSP and the OPEN/CÆSAR environment.

For the FSP user community, CADP 2010 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 2010 is the support of Lotos NT [5], a specification language derived from the Iso standard E-Lotos [35]. Lotos NT is an attempt [17] 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 for 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, and arrays; 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; 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.

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 2010 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 [48] 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 [23], 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 [26], 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 continuations during the translation.

3.4 Support for other languages

Numerous other languages have been connected to CADP 2010. Fig. 1 gives a global picture; dark grey boxes indicate the languages and software components included in CADP 2010; light grey boxes indicate the languages for which VASY has developed translators and connections to CADP 2010, these translators being distributed separately from CADP 2010; arcs are labeled with bibliographic references; arcs without labels correspond to work in progress.

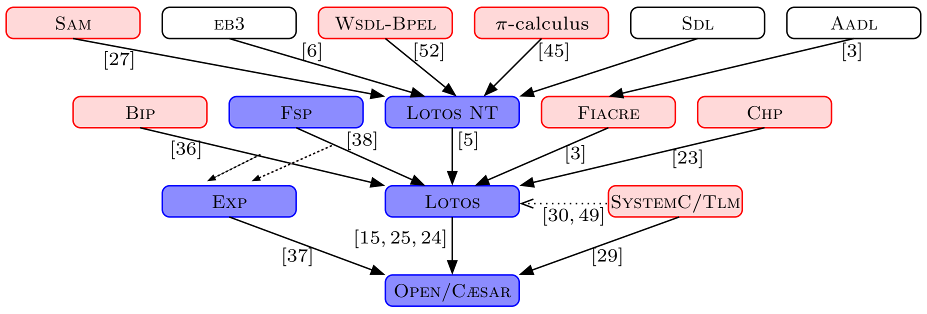

Fig. 1. Connection of the input languages of CADP 2010

4 Model Checking

CADP contains three model checkers operating on explicit and implicit LTSs. XTL (*eXecutable Temporal Language*) [44] 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 [46] evaluates formulas of Rafmc (the regular alternation-free μ-calculus) on an implicit Lts on the fly. Rafmc incorporates the Pdl 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 [43]. According to the shape of the formula, the most memory-efficient algorithm of the library is selected automatically. 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.

Evaluator 4.0 [47] 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, 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*. Although $L\mu_2$ has, in the worst case, a quadratic model checking complexity, the fairness operators of Mcl are evaluated in linear-time using an enhanced resolution algorithm of Cæsar_Solve [47].

5 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, branching bisimulation, as well as stochastic/probabilistic extensions of strong and branching bisimulations (which take into account the notion of *lumpability*) 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 [11, 13] and, since 1999, with the Bcg_Min 1.0 tool for minimization of explicit graphs using various partition-refinement algorithms. The functionalities of these two tools have been progressively subsumed by improved tools, namely Bcg_Min 2.0 and Bisimulator, available in Cadp 2010.

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 [4]. 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 a few hours, using less than 100 Gbytes RAM.

BISIMULATOR [2, 43] 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 [43] library of CADP. This encoding generalizes and, due to optimizations applied on the fly depending on the LTS structure, outperforms the pioneering on-the-fly equivalence checking algorithms [13]. For typical cases (e.g., when the service LTS is deterministic and/or τ-free, τ denoting the hidden/invisible action), 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, weak, branching, $\tau^*.a$ [13], safety, trace, and weak trace) and their associated preorders, and is 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 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.

6 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 [9]). Besides BCG_MIN, the EXP.OPEN tool [37] 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 [32], which eliminates stochastic nondeterminism in extended Markovian models on the fly using a variant of the algorithm presented in [10], 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.

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.

7 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 forward 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 [22, 21] split the generation of an Lts across several machines, each machine building only a fragment of the entire Lts. Interestingly, 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 2010 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.

8 Conclusion

Concurrency theory is now 40-year old; formal methods are 35-year old; model checking verification is nearly 30-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 2010 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, MacOS 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. As of December 2010, academic and commercial licences have been signed with more than 435 universities, public research institutes, and global corporations; 137 case-studies have been tackled using CADP; 58 research software applications have been developed using CADP; numerous academic courses are using CADP to teach concurrency; the CADP user forum gathers more than 150 registered members with 1000 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 distributed programming; yet this is the only way to exploit parallel computing resources, which are becoming pervasive.

References

1. H. R. Andersen. Model Checking and Boolean Graphs. *TCS*, 126(1):3–30, 1994.
2. D. Bergamini, N. Descoubes, C. Joubert, R. Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In *TACAS*, *LNCS* 3440. 2005.
3. B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat. FIACRE: An Intermediate Language for Model Verification in the TOPCASED Environment. In *ERTS*. 2008.

4. S. Blom, S. Orzan. Distributed state space minimization. *STTT*, 7:280–291, 2005.
5. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, F. Lang, W. Serwe, G. Smeding. Reference Manual of the Lotos NT to Lotos Translator (Version 5.1). Tech. Report INRIA/VASY, 117 pages, 2010.
6. R. Chossart. Évaluation d'outils de vérification pour les spécifications de systèmes d'information. Mémoire maître ès sciences, Univ. de Sherbrooke, Canada, 2010.
7. R. Cleaveland, T. Li, S. Sims. The Concurrency Workbench of the New Century (Version 1.2). User's manual, 2000.
8. R. Cleaveland, J. Parrow, B. Steffen. The Concurrency Workbench. In *the 1st Workshop on Automatic Verification Methods for Finite State Systems*, *LNCS* 407. 1989.
9. N. Coste, H. Garavel, H. Hermanns, F. Lang, R. Mateescu, W. Serwe. Ten Years of Performance Evaluation for Concurrent Systems Using Cadp. In *ISoLA*, *LNCS* 6416. 2010.
10. D. D. Deavours, W. H. Sanders. An Efficient Well-Specified Check. In *PNPM*. 1999.
11. J.-C. Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*. Thèse de Doctorat, Univ. J. Fourier (Grenoble), 1988.
12. J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, M. Sighireanu. Cadp (Cæsar/Aldébaran Development Package): A Protocol Validation and Verification Toolbox. In *CAV*, *LNCS* 1102. 1996.
13. J.-C. Fernandez, L. Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In *CAV*, *LNCS* 575. 1991.
14. H. Garavel. *Compilation et vérification de programmes* Lotos. Thèse de Doctorat, Univ. J. Fourier (Grenoble), 1989.
15. H. Garavel. Compilation of Lotos Abstract Data Types. In *FORTE*. 1989.
16. H. Garavel. Open/Cæsar: An Open Software Architecture for Verification, Simulation, and Testing. In *TACAS*, *LNCS* 1384, 1998.
17. H. Garavel. Reflections on the Future of Concurrency Theory in General and Process Calculi in Particular. In *LIX Colloquium on Emerging Trends in Concurrency Theory*, *ENTCS* 209. 2008.
18. H. Garavel, F. Lang. SVL: a Scripting Language for Compositional Verification. In *FORTE*. IFIP, 2001.
19. H. Garavel, F. Lang, R. Mateescu. An Overview of Cadp 2001. *EASST Newsletter*, 4:13–24, 2002.
20. H. Garavel, F. Lang, R. Mateescu, W. Serwe. Cadp 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *CAV*, *LNCS* 4590. 2007.
21. H. Garavel, R. Mateescu, D. Bergamini, A. Curic, N. Descoubes, C. Joubert, I. Smarandache, G. Stragier. Distributor and Bcg_Merge: Tools for Distributed Explicit State Space Generation. In *TACAS*, *LNCS* 3920. 2006.
22. H. Garavel, R. Mateescu, I. Smarandache. Parallel State Space Construction for Model-Checking. In *SPIN*, *LNCS* 2057. 2001.
23. H. Garavel, G. Salaün, W. Serwe. On the Semantics of Communicating Hardware Processes and their Translation into Lotos for the Verification of Asynchronous Circuits with Cadp. *SCP*, 74(3):100–127, 2009.
24. H. Garavel, W. Serwe. State Space Reduction for Process Algebra Specifications. *TCS*, 351(2):131–145, Feb. 2006.
25. H. Garavel, J. Sifakis. Compilation and Verification of Lotos Specifications. In *PSTV*. IFIP, 1990.
26. H. Garavel, M. Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In *FORTE/PSTV*. 1999.

27. H. Garavel, D. Thivolle. Verification of GALS Systems by Combining Synchronous Languages and Process Calculi. In *SPIN*, *LNCS* 5578. 2009.
28. H. Garavel, P. Turlier. CÆSAR.ADT : un compilateur pour les types abstraits algébriques du langage LOTOS. In *Actes du CFIP*, 1993.
29. C. Helmstetter. TLM.OPEN: a SYSTEMC/TLM Front-End for the CADP Verification Toolbox. http://hal.archives-ouvertes.fr/hal-00429070/
30. C. Helmstetter, O. Ponsini. A Comparison of Two SystemC/TLM Semantics for Formal Verification. In *MEMOCODE*. 2008.
31. H. Hermanns. *Interactive Markov Chains and the Quest for Quantified Quality*, *LNCS* 2428. Springer, 2002.
32. H. Hermanns, C. Joubert. A Set of Performance and Dependability Analysis Components for CADP. In *TACAS*, *LNCS* 2619. 2003.
33. G. J. Holzmann. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, 2003.
34. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization, Geneva, 1989.
35. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization, Geneva, 2001.
36. A. M. Khan. Connection of Compositional Verification Tools for Embedded Systems. Mémoire master 2 recherche, Univ. J. Fourier, Grenoble, 2006.
37. F. Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. In *IFM*, *LNCS* 3771. 2005.
38. F. Lang, G. Salaün, R. Hérilier, J. Kramer, J. Magee. Translating FSP into LOTOS and Networks of Automata. *FACJ*, 22(6):681–711, Nov. 2010.
39. A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. Bertz, Berlin, 1997.
40. J. Magee, J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2006.
41. R. Mateescu. *Vérification des propriétés temporelles des programmes parallèles*. Thèse de Doctorat, Institut National Polytechnique de Grenoble, Apr. 1998.
42. R. Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In *TACAS*, *LNCS* 1785. 2000.
43. R. Mateescu. CÆSAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *STTT*, 8(1):37–56, Feb. 2006.
44. R. Mateescu, H. Garavel. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In *STTT*. BRICS, 1998.
45. R. Mateescu, G. Salaün. Translating Pi-Calculus into LOTOS NT. In *IFM*, *LNCS* 6396. 2010.
46. R. Mateescu, M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *SCP*, 46(3):255–281, 2003.
47. R. Mateescu, D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *FM*, *LNCS* 5014. 2008.
48. O. Ponsini, C. Fédèle, E. Kounalis. Rewriting of imperative programs into logical equations. *SCP*, 56(3):363–401, 2005.
49. O. Ponsini, W. Serwe. A Schedulerless Semantics of TLM Models Written in SYSTEMC via Translation into LOTOS. In *FM*, *LNCS* 5014. 2008.
50. S. Schewe. Solving Parity Games in Big Steps. In *FSTTCS*, *LNCS* 4855. 2007.
51. P. Stevens, C. Stirling. Practical Model-Checking using Games. In *TACAS*, *LNCS* 1384. 1998.
52. D. Thivolle. *Langages modernes pour la vérification des systèmes asynchrones*. PhD thesis, Univ. J. Fourier Grenoble and Polytechnic. Univ. of Bucharest, 2011.