# Distributed On-the-Fly Model Checking and Test Case Generation

Christophe Joubert and Radu Mateescu

INRIA Rhône-Alpes / VASY
655, av. de l'Europe, F-38330 Montbonnot St Martin, France
{Christophe.Joubert,Radu.Mateescu}@inria.fr

**Abstract.** The explicit-state analysis of concurrent systems must handle large state spaces, which correspond to realistic systems containing many parallel processes and complex data structures. In this paper, we combine the on-the-fly approach (incremental construction of the state space) and the distributed approach (state space exploration using several machines connected by a network) in order to increase the computing power of analysis tools. To achieve this, we propose MB-DSOLVE, a new algorithm for distributed on-the-fly resolution of multiple block, alternation-free boolean equation systems (BESS). First, we apply MB-DSOLVE to perform distributed on-the-fly model checking of alternation-free modal $\mu$-calculus, using the standard encoding of the problem as a BES resolution. The speedup and memory consumption obtained on large state spaces improve over previously published approaches based on game graphs. Next, we propose an encoding of the conformance test case generation problem as a BES resolution from which a diagnostic representing the complete test graph (CTG) is built. By applying MB-DSOLVE, we obtain a distributed on-the-fly test case generator whose capabilities scale up smoothly w.r.t. well-established existing sequential tools.

## 1 Introduction

The explicit-state verification of concurrent finite-state systems is confronted in practice with the *state explosion* problem (prohibitive size of the underlying state spaces), which occurs for realistic systems containing many parallel processes and complex data structures. Various approaches have been proposed for combating state explosion, among which: *on-the-fly* verification constructs the state space in a demand-driven way, thus allowing the detection of errors without a priori building the entire state space, and *distributed* verification uses the computing resources of several machines connected by a network, thus allowing to scale up the capabilities of verification tools by one or two orders of magnitude. Practical experience suggests that combining these two techniques leads potentially to better results than using them separately.

Given that verification tools are complex pieces of software, their design should promote modular architectures and intermediate representations, in order

to reuse existing achievements as much as possible. *Boolean Equation Systems* (BESs) [22] are a useful intermediate representation for various verification problems, such as model checking of modal $\mu$-calculus [1, 22], equivalence checking [2, 24], and partial order reduction [28]. Numerous sequential algorithms for on-the-fly BES resolution were proposed [1, 31, 22, 26], some of them being subject to generic implementations, such as the CÆSAR_SOLVE library [25], which serves as computing engine for the model checker EVALUATOR [26, 24], the equivalence checker BISIMULATOR [24, 3], and the reductor TAU_CONFLUENCE [28, 25], developed within the CADP toolbox [11]. Due to their modular architecture, distributed versions of these tools can be obtained in a straightforward manner by developing distributed BES resolution algorithms, such as DSOLVE [17], which handles BESs with a single equation block and underlies the distributed version of BISIMULATOR [16].

In this paper, we propose MB-DSOLVE, a new distributed on-the-fly resolution algorithm for multiple block, alternation-free BESs. The algorithm is based upon a distributed breadth-first exploration of the *boolean graph* [1] representing the dependencies between boolean variables of a BES. Our first application of MB-DSOLVE was the distributed on-the-fly model checking of alternation-free $\mu$-calculus formulas (as computing engine for EVALUATOR), using the standard translation of the problem into a BES resolution [19, 6, 1]. The only existing distributed on-the-fly algorithm for solving this problem was proposed in [4] and is based on *game graphs*, stemming from a game-based formulation of the problem [29]. The latest version of this algorithm, called PTCL1 and implemented in the model-checker UPPDMC [13], has an extension, called PTCL2, which is also able to handle $\mu$-calculus formulas of alternation depth 2 [21] and exhibits good performance on large state spaces, such as those of the VLTS benchmark suite[1]. Although the two algorithms MB-DSOLVE and PTCL1 are graph-based and therefore similar in spirit, MB-DSOLVE allows all machines involved in the distributed computation to handle simultaneously all equation blocks of a BES, thus potentially reaching a higher degree of concurrency than PTCL1, which at a given moment synchronizes and employs all machines to solve a precise part, called *component*, of the game graph. This intuition is confirmed experimentally on large states spaces from the VLTS benchmark.

Our second application of MB-DSOLVE was the distributed on-the-fly generation of conformance test cases from specifications and test purposes (both given as state spaces), following the approach advocated in the TGV tool [15]. To achieve this, we proposed an encoding of the test generation problem as a BES resolution from which a diagnostic representing the *Complete Test Graph* (CTG) is built, and we implemented it within CADP in a tool named EXTRACTOR. This led to sequential and distributed test case generation functionalities, obtained by applying the algorithms of CÆSAR_SOLVE optimized for disjunctive/conjunctive BESs [25] and MB-DSOLVE, respectively. The BES technology proved again its usefulness: the performance of the sequential version of EXTRACTOR exhibits comparable performances with the optimized algorithms of TGV, and the dis-

[1] `http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html`

tributed version scales smoothly to larger systems. As far as we know, this is the first attempt of building a distributed on-the-fly conformance test generator.

The paper is organized as follows. Section 2 recalls basic definitions of BESs and describes in detail the MB-DSOLVE resolution algorithm. Section 3 translates the problems of model checking alternation-free $\mu$-calculus formulas and of conformance test case generation into BES resolutions. Section 4 shows experimental data comparing the performance of the distributed tools with their sequential versions and with other similar distributed tools. Finally, Section 5 gives some concluding remarks and directions for future work.

## 2 Distributed Local Resolution of Alternation-Free BESs

We first define the framework underlying the manipulation of alternation-free BESs, and then we present the MB-DSOLVE algorithm for distributed on-the-fly resolution.

### 2.1 Alternation-Free BESs

A Boolean Equation System (BES) [1, 22], defined over $\mathcal{X}$, a set of boolean variables, is a tuple $B = (x, M_1, ..., M_n)$, where $x \in \mathcal{X}$ is a boolean variable and $M_i$ are equation blocks ($i \in [1, n]$). Each block $M_i = \{x_j \stackrel{\sigma_i}{=} op_j \boldsymbol{X}_j\}_{j \in [1, m_i]}$ is a set of minimal (resp. maximal) fixed point equations of sign $\sigma_i = \mu$ (resp. $\sigma_i = \nu$). The right-hand side of each equation $j$, from block $M_i$, is a pure disjunctive or conjunctive formula obtained by applying a boolean operator $op_j \in \{\vee, \wedge\}$ to a set of variables $\boldsymbol{X}_j \subseteq \mathcal{X}$. The boolean constants $\mathsf{false}$ and $\mathsf{true}$ abbreviate the empty disjunction $\vee\emptyset$ and the empty conjunction $\wedge\emptyset$. A variable $x_j$ depends upon a variable $x_l$ if $x_l \in \boldsymbol{X}_j$. A block $M_i$ depends upon a block $M_k$ if some variable of $M_i$ depends upon a variable defined in $M_k$. A block is *closed* if it does not depend upon any other blocks. A BES is *alternation-free* if there are no cyclic dependencies between its blocks; in this case, the blocks are sorted topologically such that a block $M_i$ only depends upon blocks $M_k$ with $k > i$. The *main* variable $x$ must be defined in block $M_1$.

The semantics $[\![op_i\{x_1, ..., x_k\}]\!]\delta$ of a formula $op_i\{x_1, ..., x_k\}$ w.r.t. $\mathbb{B} = \{\mathsf{false}, \mathsf{true}\}$ and a context $\delta : \mathcal{X} \rightarrow \mathbb{B}$, which must initialize all variables $x_1$, ..., $x_k$, is the boolean value $op_i(\delta(x_1), ..., \delta(x_k))$. The semantics $[\![M_i]\!]\delta$ of a block $M_i$ w.r.t. a context $\delta$ is the $\sigma_i$-fixed point of a vectorial functional $\Phi_{i\delta} : \mathbb{B}^{m_i} \rightarrow \mathbb{B}^{m_i}$ defined as $\Phi_{i\delta}(b_1, ..., b_{m_i}) = ([\![op_j \boldsymbol{X}_j]\!](\delta \oslash [b_1/x_1, ..., b_{m_i}/x_{m_i}]))_{j \in [1, m_i]}$, where $\delta \oslash [b_1/x_1, ..., b_{m_i}/x_{m_i}]$ denotes a context identical to $\delta$ except for variables $x_1, ..., x_{m_i}$, which are assigned values $b_1, ..., b_{m_i}$, respectively. The semantics of an alternation-free BES is the value of its main variable $x$ given by the solution of $M_1$, i.e., $\delta_1(x)$, where the contexts $\delta_i$ are calculated as follows: $\delta_n = [\![M_n]\!][]$ (the context is empty because $M_n$ is closed), $\delta_i = ([\![M_i]\!]\delta_{i+1}) \oslash \delta_{i+1}$ for $i \in [1, n-1]$ (a block $M_i$ is interpreted in the context of all blocks $M_k$ with $k > i$).

The *local* (or *on-the-fly*) resolution of an alternation-free BES $B = (x, M_1, ..., M_n)$ consists in computing the value of $x$ by exploring the right-hand

sides of the equations in a demand-driven way, without explicitly constructing the blocks. Several sequential on-the-fly BES resolution algorithms are available [6, 1, 22, 7]; here we adopt the approach proposed in [1], which formulates the resolution problem in terms of the *boolean graph* encoding the dependencies between variables of the BES (see Figure 2 for an example of BES with three blocks and its associated boolean graph; black and white vertices denote false and true variables, respectively). The resolution of variable $x$ amounts to perform a forward exploration of the dependencies going out of $x$, intertwined with a backward propagation of stable variables (whose value is determined) along dependencies; the resolution terminates either when $x$ becomes stable (after propagation of some stable successors) or when the portion of boolean graph reachable from $x$ is completely explored.

### 2.2 Distributed Local Resolution Algorithm

To achieve an on-the-fly distributed resolution of multiple block, alternation-free BESs, we have designed a new algorithm, called MB-DSOLVE (for *MultiBlock Distributed SOLVEr*), which consists mainly of the following components.

**A distributed-memory architecture model.** Performances of distributed algorithms highly depend on the architecture on which they are executed. Distributed architectures that are targeted in this study, do not hold a shared memory and are rather generic, since they are essentially composed of machines interconnected through standard network (e.g., Gigabit Ethernet) present in most companies and laboratories. Examples of such architectures, based upon message passing, are the networks of workstations (NOWS) and the clusters of PCS.

**A SPMD programming model.** Each computing machine, called *worker node*, executes an instance of our distributed algorithm MB-DSOLVE (defined on Figure 1), according to the *Single Program, Multiple Data* (SPMD) programming model. Data is provided through the local exploration of the boolean graph associated to the BES and by the reception of boolean variables (i.e., boolean graph vertices) sent from remote nodes. Equation blocks are processed by two intertwined graph traversals: (1) a local forward exploration, based on a DFS of the dependency graph between blocks (in EXPANSION at line 11), and on a BFS of the boolean graph of each block (in EXPAND at line 64)), starting from the main variable $x$; (2) a propagation of stable boolean variables (whose values have been computed) along backward dependencies (i.e., edges of the boolean graph). In addition to worker nodes, a special *supervisor* process, usually executed on the user local machine, is responsible for initializing the distributed computation by copying files and launching workers on remote nodes, for collecting statistics about the BES resolution, and for detecting (normal and urgent) termination. A description of the supervisor process associated to the DSOLVE resolution algorithm for single block BESs can be found in [17]. Its extension to multiple block BESs involves a multiplexing of the data structures for each equation block and of the distributed termination detection (DTD) algorithm in order to detect the partial termination of each block and the global termination of the resolution.

```
1  MB-DSOLVE(x,(V,E,L),N,P,h,i)→ 𝔹 :
2    if h(x) = i then
3      S_{b(x)i} := {x};  W_{b(x)i} := put(x, nil);
4      initialize(x)
5    endif;  term_{b(x)i} := false;
6    while ¬term_{b(x)i} do
7      if IRECEIVE(msg_i, sender_i) then
8        READ(msg_i, sender_i)
9      elsif (l_i := STABILIZATION?) ≤ N then
10       STABILIZATION(l_i)
11     elsif (k_i := EXPANSION?) ≥ 1 then
12       EXPANSION(k_i)
13     else
14       RECEIVE(msg_i, sender_i);
15       READ(msg_i, sender_i)
16     endif
17   endwhile;
18   returnv(x)

19 READ(m_i, s_i):
20   case m_i is
21     Exp(x_{ks_i}, y_{li}) →  if k ≠ l then
22           Q_{li} ∪:= {(x_{ks_i}, y_{li})}
23           else EXPAND(x_{ks_i}, y_{li}) endif
24     Evl(x_{ki}, y_{ls_i}) → if k ≠ l then
25           exp_req_{ki} −:= 1 endif;
26           if ¬stable(x_{ki}) then
27           STABILIZE(x_{ki}, y_{ls_i}) endif
28   endcase

29 STABILIZATION(l):
30   while B_{li} ≠ ∅ ∨ (term_{li} ∧ R_{li} ≠ ∅) do
31     if B_{li} ≠ ∅ then y_{li} := get(B_{li});
32       B_{li} \:= {y_{li}} else y_{li} := get(R_{li});
33     R_{li} \:= {y_{li}} endif;
34     forall w_{kj} ∈ d(y_{li}) ∧ (B_{li} ≠ ∅ ∨ k ≠ l)
35       ∧¬term_{b(x)i} ∧ ¬stable(w_{kj}) do
36       if h(w_{kj}) = i then
37         if k ≠ l then exp_req_{ki} −:= 1
38         endif; STABILIZE(w_{kj}, y_{li})
39       else SENDING(Evl(w_{kj}, y_{li}), h(w_{kj}))
40       endif
41     endfor;  d(y_{li}) := ∅
42   endwhile

43 STABILIZE(w_{ki}, y_{lj}):
44   if ((L(w_{ki}) = ∨) ∧ v(y_{lj}))∨
45   ((L(w_{ki}) = ∧) ∧ ¬v(y_{lj})) then
46     s(w_{ki}) := y_{lj};  c(w_{ki}) := 0;
47     stable(w_{ki}) := true
48   else c(w_{ki}) −:= 1 endif;
49   if stable(w_{ki}) then B_{ki} ∪:= {w_{ki}};
50     if w_{ki} ∈ R_{ki} then R_{ki} \:= {w_{ki}}
51     endif;  term_{b(x)i} := stable(x)
52   endif

53 EXPANSION(k):
54   if W_{ki} = nil then
55     forall (x_{lj}, y_{ki}) ∈ (Q_{ki})
56         ∧¬term_{b(x)i} do
57       if j ≠ i ∨ ¬stable(x_{lj})
58         then EXPAND(x_{lj}, y_{ki})
59       elsif l ≠ k then
60         exp_req_{li} −:= 1 endif
61     endfor
62   else
63     x_{ki} := head(W_{ki}); W_{ki} := tail(W_{ki});
64     forall y_{lj} ∈ E(x_{ki}) ∧ ¬term_{b(x)i}
65         ∧¬stable(x_{ki}) do
66       if k ≠ l then exp_req_{ki} +:= 1
67       endif;
68       if h(y_{lj}) = i then
69         if k ≠ l then
70           Q_{li} ∪:= {(x_{ki}, y_{lj})}
71         else EXPAND(x_{ki}, y_{lj}) endif
72       else
73         SENDING(Exp(x_{ki}, y_{lj}), h(y_{lj}))
74       endif
75     endfor
76   endif

77 EXPAND(x_{kj}, y_{li}):
78   if y_{li} ∉ li then
79     S_{li} ∪:= {y_{li}};  initialize(y_{li});
80     if c(y_{li}) ≠ 0 then
81       W_{li} := put(y_{li}, W_{li})
82     else stable(y_{li}) := true endif
83   endif;
84   if k ≠ l ∧ y_{li} ∉ R_{li} then
85     R_{li} ∪:= {y_{li}} endif;
86   if stable(y_{li}) then
87     if y_{li} ∈ R_{li} then R_{li} \:= {y_{li}}
88     endif;
89     if h(x_{kj}) = i then
90       if k ≠ l then exp_req_{li} −:= 1
91       endif; STABILIZE(x_{kj}, y_{li})
92     else B_{li} ∪:= {y_{li}};
93       d(y_{li}) ∪:= {x_{kj}} endif
94   else d(y_{li}) ∪:= {x_{kj}} endif
```

**Fig. 1.** Distributed local resolution of multiple block, alternation-free BES using its boolean graph

The function MB-DSOLVE, presented on Figure 1, describes the behavior of a worker node $i : [1..P]$ during a distributed resolution on $P$ nodes of the main variable $x \in V$ of the multiple block BES $B = (x, M_1, ..., M_N)$ composed of $N$ blocks and defined by the boolean graph $(V, E, L)$, where $V = \{x_j \mid j \in [1, m_i]\}$ is the set of *vertices* (boolean variables), $E : V \to 2^V,\ E = \{x_j \to x_k \mid x_k \in \boldsymbol{X}_j\}$, the set of *edges* (dependencies between variables), and $L : V \to \{\vee, \wedge\},\ L(x_j) = op_j$, the *vertex labeling* (disjunctive or conjunctive). The set of successors of a vertex $x$ is noted $E(x)$. Data distribution is ensured by a static hash function $h : V \to [1..P]$ globally known by all workers. Upon termination, the function returns the final boolean value computed for the main variable $x$.

Due to space limitations, only the main part of MB-DSOLVE is detailed on Figure 1. To each block $k$ is associated, locally to node $i$, a set $S_{ki} \subseteq V$ which stores visited vertices, a BFS queue $W_{ki}$ which stores vertices visited but not explored yet, and three variable sets $B_{ki}$, $R_{ki}$, and $Q_{ki}$ (all initially empty) used to store stable variables, unstable variables with an interblock predecessor dependency, and interblock transitions pending to be explored, respectively. $exp\_req_{ki}$, initialized to 0, gives the number of interblock transitions starting from variables in block $k$ locally to node $i$, which needs to be eventually traversed by propagating he values of stable target variables. To each vertex $y_{ki}$ are associated four informations: a counter $c(y_{ki})$, which keeps the number of $y_{ki}$'s successors that must be stabilized in order to make the value of $y_{ki}$ stable, its boolean value $v(y_{ki})$, a set $d(y_{ki})$ containing the vertices that currently depend upon $y_{ki}$, and $stable(y_{ki})$ indicating if $y_{ki}$ has a stable value (i.e., if $c(y_{ki}) = 0$ or if $y_{ki}$ belongs to a completely explored and stabilized portion of block $k$). These informations are set up by $initialize(y_{ki})$ as follows: depending on fixed point sign, $\mu$ or $\nu$, and variable type, $\wedge$ or $\vee$, $c(y_{ki})$ is initialized to $|E(y_{ki})|$ (i.e., if $(\sigma(b(x)) = \mu \wedge L(x) = \wedge) \vee\ (\sigma(b(x)) = \nu \wedge L(x) = \vee)$) or 1 otherwise; $v(y_{ki})$ is initialized w.r.t. the fixed point sign defining the variable $y_{ki}$ (i.e., if $\sigma(b(x)) = \mu$ then false); $d(y_{ki})$ is initially empty; and $stable(y_{ki})$ is initially false.

At each iteration of the main while-loop (lines 6–17), received messages are processed first (lines 7–8). Then, the block with minimal index $l_i \in [1, N]$ that has stable variables not propagated yet (i.e., $B_{li} \neq \emptyset$) or that is completely explored but contains interblock predecessor dependencies not yet traversed by backward propagation of stable values (i.e., $term_{li} \wedge R_{li}$), is returned by STABILIZATION and stabilized by STABILIZATION$(l_i)$ (lines 9–10). If such block does not exist, the block $k_i$ with maximal index that has a non-empty BFS queue (i.e., $W_{ki} \neq nil$) or that is completely explored and contains pending resolution requests on unvisited variables (i.e., $exp\_req_{ki} = 0 \wedge B_{ki} = R_{ki} = \emptyset \wedge term_{ki} \wedge Q_{ki} \neq \emptyset$), is returned by EXPANSION and explored with EXPANSION$(k_i)$ (lines 11–12). Finally, if there is no more work on any block, the worker $i$ remains blocked on reception, waiting, e.g., for termination detection messages sent by the supervisor (lines 14–15).

**A distributed generation of diagnostics (examples and counterexamples).** The result of the distributed BES resolution must be accompanied by a diagnostic (example or counterexample) which provides the minimal amount

of information needed for understanding the value computed for the main variable $x$. MB-DSOLVE computes diagnostic information in the form of a boolean subgraph rooted at $x$, following the approach proposed in [23]. The minimal information necessary for producing the diagnostic is stored as $s(w_{ki}$ (line 46), indicating the successor of variable $w_{ki}$ that stabilized it after a backward propagation (e.g., a true successor of an $\vee$-variable). This provides an implicit, distributed representation of the diagnostic, which can be explored on-the-fly once the resolution has finished.

**A distributed termination detection for each equation block.** The variable $term_{b(x)i}$ is set to true when distributed termination of the BES resolution is detected. Conditions of termination are: either the main variable $x$ has been stabilized ($c(x) = 0$) during backward propagation (line 51), or the boolean graph has been completely explored, i.e., all local working sets of variables are empty ($\forall i \in [1..P], k \in [1..N] \cdot W_{ki} = nil \wedge B_{ki} = R_{ki} = Q_{ki} = \emptyset \wedge exp\_req_{ki} = 0$) and no more messages are transiting through the network. Our proposed distributed algorithm contains a mechanism of partial termination detection ($term_{ki}$) given a block $k$ under resolution on node $i$. This approach enables to obtain a fine-grain parallelism of the BES resolution by allowing concurrent distributed graph traversals on different blocks at the same time. The inactivity detection of the nodes locally to the resolution of a particular block, enables to speedup the overall BES resolution by increasing the number of blocks explored in parallel, and then, the probability of finding more rapidly a partially solved block from which stable values can be propagated.

Our DTD is based on the four-counter method presented in [27] on a star-shaped topology with a central agent (the supervisor process) whose role is asymmetric to worker nodes [14]. Activity status of workers is regularly sent to the supervisor, which therefore has a global view of the computation and is able to initiate the DTD for an equation block with higher probability of success than traditional ring-based DTD algorithms.

**A linear complexity in time, memory and messages.** MB-DSOLVE is based on the theory of boolean graphs underlying the sequential resolution algorithms for alternation-free BESs [1, 31]. It consists roughly of two intertwined traversals (forward and backward) of the boolean graph, with a worst-case time complexity $O(|V| + |E|)$. The same bound applies for memory complexity, due to dependencies (between variables in a same block, and between blocks) that are stored during resolution for eventual propagations of stable variables. The communication cost of MB-DSOLVE can also be estimated, assuming that messages (excluding those for DTD) are sent for each *cross-dependency* (i.e., edge $(x, y) \in E \mid h(x) \neq h(y)$). Since the hash function $h$ shares variables equally among nodes without *a priori* knowledge about locality, it also shares dependencies equally. Thus, the number of cross-dependencies can be evaluated to $((P - 1)/P) \cdot |E|$, since statistically only $|E|/P$ edges will be local to a node. Hence, the message complexity is $O(|E|)$, the worst-case being obtained with two messages (expansion and stabilization) exchanged per cross-dependency, i.e., at most $2 \cdot ((P - 1)/P) \cdot |E|$ messages. Theoretically, our DTD algorithm has also

a message complexity linear in the number of edges ($O(|E|)$), but in practice it reveals to be very efficient, with only 0.01% of total exchanged messages used for DTD, thanks to the supervisor process, which has an up-to-date global view of the computation.

**Illustration of the distributed local resolution.** Let us consider the boolean graph on the left part of Figure 2 as a simple multiple block, alternation-free BES example. It suggests that the distribution function $h$ will map the given vertices onto three computation nodes (from $P_1$ to $P_3$) as shown on the right part of Figure 2. Starting with the main variable $x_{1,1}$, its successors $x_{2,1}$ and $x_{1,2}$ are computed locally to $P_1$ and sent to node $P_3$, respectively. Now, $x_{2,1}$ and $x_{1,2}$ can be expanded in parallel with the effect that $x_{3,1}$ is sent to node $P_2$, that $x_{1,3}$ has an interblock predecessor dependency with $x_{2,1}$, and that $x_{1,3}$ and $x_{2,2}$ are computed locally to node $P_3$. Since $x_{1,3}$ is an empty disjunction (i.e., a constant false), its value is stable and can be propagated through back-dependencies (i.e., predecessor vertices) in o! rder to stabilize $\wedge$-vertex $x_{2,1}$ but not $\vee$-vertex $x_{1,2}$. By further propagating stable values, such as the value false of $x_{2,1}$, we eventually stabilize the value of $x_{1,1}$ to false.

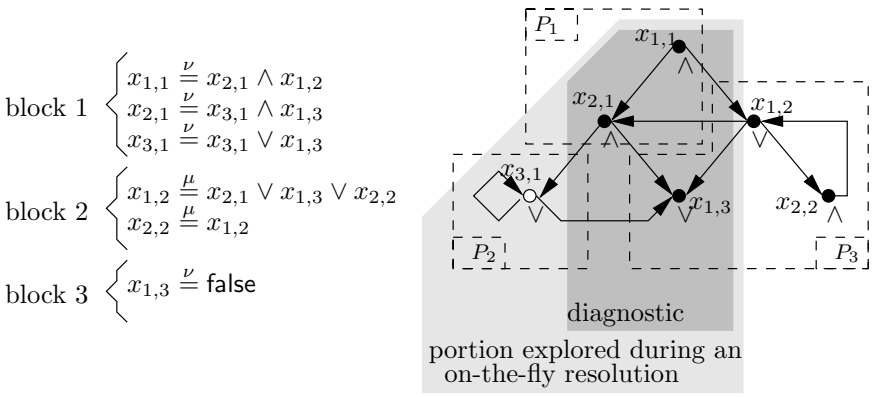



**Fig. 2.** A multiple block, alternation-free BES, its partitioned boolean graph, and the result of a distributed on-the-fly resolution for $x_{1,1}$ on three nodes

Since the boolean graph is constructed on-the-fly (and asynchronously between nodes), only a portion of it needs actually to be explored, as shown by vertices $x_{1,2}$ and $x_{2,2}$ that can be excluded from possible explored portion (in light grey area). This example also gives in the dark grey area a possible diagnostic built upon resolution of the boolean graph. During the backward-propagation of stable values, some information can be saved to enable later diagnostic generation whose basic mechanism is to choose which successor vertex, if it exists, has directly stabilized the current vertex ($s(w_{ki})$ at line 46). As previously indi-

cated, MB-DSOLVE has such a feature and gives an implicit successor function permitting a distributed on-the-fly construction of the diagnostic.

## 3 Applications

We illustrate in this section the application of the MB-DSOLVE algorithm on two analysis problems defined on *Labeled Transition Systems* (LTSs): model checking of alternation-free $\mu$-calculus formulas and generation of conformance test cases. An LTS is a tuple $(S, A, T, s_0)$ containing a set of states $S$, a set of actions $A$, a transition relation $T \subseteq S \times A \times S$ and an initial state $s_0 \in S$. A transition $(s, a, s') \in T$, noted also $s \xrightarrow{a} s'$, states that the system can move from state $s$ to state $s'$ by executing action $a$ ($s'$ is an $a$-successor of $s$). Both problems can be formulated as the resolution of a multiple block, alternation-free BES, the second one essentially relying upon diagnostic generation for BESs. By applying MB-DSOLVE as BES resolution engine, we obtain distributed versions of the on-the-fly model checker EVALUATOR 3.5 [24] and the on-the-fly test generator TGV [15] of the CADP toolbox [11].

### 3.1 Model checking for alternation-free mu-calculus

Modal $\mu$-calculus [18] is a powerful fixed point based logic for specifying temporal properties on LTSs. Its formulas are defined by the following grammar (where $X$ is a propositional variable):

$$\phi ::= \mathsf{false} \mid \mathsf{true} \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \langle a \rangle\, \phi \mid [a]\, \phi \mid X \mid \mu X.\phi \mid \nu X.\phi$$

Given an LTS $(S, A, T, s_0)$, a formula $\phi$ denotes a set of states, defined as follows: boolean formulas have their usual set interpretation; modalities $\langle a \rangle\, \phi$ (resp. $[a]\, \phi$) denote the states having some (resp. all) $a$-successors satisfying $\phi$; fixed point formulas $\mu X.\phi$ (resp. $\nu X.\phi$) denote the minimal (resp. maximal) solution of the equation $X = \phi$, interpreted over $2^S$. The local model checking problem amounts to establish whether the initial state $s_0$ of an LTS satisfies a formula $\phi$, i.e., belongs to the set of states denoted by $\phi$.

The alternation-free fragment of the modal $\mu$-calculus, noted $L^1_\mu$ [8], is obtained by forbidding mutual recursion between minimal and maximal fixed point variables. $L^1_\mu$ benefits from model checking algorithms whose complexity is linear in the size of the LTS (number of states and transitions) and the formula (number of operators), while still allowing to express useful properties, since it subsumes CTL [5] and PDL [9]. The model checking of $L^1_\mu$ formulas on LTSs can be encoded as the resolution of an alternation-free BES [6, 20, 1]. We illustrate the encoding by considering the following formula, which states that the emission *snd* of a message is eventually followed by its reception *rcv* ('$-$' stands for 'any action' and '$\overline{a}$' stands for 'any action different from $a$'):

$$\nu X.([\mathit{snd}]\, \mu Y.(\langle - \rangle\, \mathsf{true} \wedge [\overline{\mathit{rcv}}]\, Y) \wedge [-]\, X)$$

The formula is translated first into a specification in HML with recursion [19], which contains two blocks of modal equations:

$$\{X =_\nu [snd]\, Y \wedge [-]\, X\}, \{Y =_\mu \langle - \rangle\, \mathsf{true} \wedge [\overline{rcv}]\, Y\}$$

Then, each modal equation block is converted into a boolean equation block by 'projecting' it on each state of the LTS:

$$\{X_s =_\nu \bigwedge_{s \xrightarrow{snd} s'} Y_{s'} \wedge \bigwedge_{s \rightarrow s'} X_{s'}\}_{s \in S}, \{Y_s =_\mu \bigvee_{s \rightarrow s'} \mathsf{true} \wedge \bigwedge_{s \not\xrightarrow{rcv} s'} Y_{s'}\}_{s \in S}$$

A boolean variable $X_s$ (resp. $Y_s$) is true iff state $s$ satisfies the propositional variable $X$ (resp. $Y$). Thus, the local model checking of the initial formula amounts to compute the value of variable $X_{s_0}$ by applying a local BES resolution algorithm. This method underlies the on-the-fly model checker EVALUATOR 3.5 [26, 24] of CADP [11], which handles formulas of $L^1_\mu$ extended with PDL-like modalities containing regular expressions over transition sequences.

### 3.2 Conformance test case generation

Conformance testing aims at establishing that the implementation under test (IUT) of a system is correct w.r.t. a specification. We consider here the conformance test approach advocated in the pioneering work underlying the TGV tool [15]. We give only a brief overview of the theory used by TGV and focus on the algorithmic aspects of test selection, with the objective of reformulating them in terms of BES resolution and diagnostic generation.

The IUT and the specification are modelled as Input-output LTSs (IOLTSs), which distinguish between inputs and outputs: e.g., the actions of the IOLTS of the specification $M = (S^M, A^M, T^M, s^M_0)$ are partitioned into $A^M = A^M_I \cup A^M_O \cup \{\tau\}$, where $A^M_I$ (resp. $A^M_O$) are input (resp. output) actions and $\tau$ is the internal (unobservable) action. Intuitively, input actions of the IUT are controllable by the environment, whereas output actions are only observable. In practice, tests observe the execution traces consisting of observable actions of the IUT, but can also detect *quiescence*, which can be of three kinds: deadlock (states without successors), outputlock (states without outgoing output actions), and livelock (cycles of internal actions). For an IOLTS $M$, quiescence is modelled by a *suspension automaton* $\Delta(M)$, an IOLTS which marks quiescent states by adding self-looping transitions labeled by a new output action $\delta$. The traces of $\Delta(M)$ are called suspension traces of $M$. The conformance relation **ioco** [30] between the IUT and the specification $M$ states that after executing each suspension trace of $M$, the (suspension automaton of the) IUT exhibits only those outputs and quiescences that are allowed by $M$.

Test generation requires a determinization of $M$, since two sequences with the same traces of observable actions cannot be distinguished. Since quiescence must be preserved, determinization must take place after the construction of the suspension automaton $\Delta(M)$.

A *test case* is an IOLTS $TC = (S^{TC}, A^{TC}, T^{TC}, s^{TC}_0)$ equipped with three sets of trap states $\mathbf{Pass} \cup \mathbf{Fail} \cup \mathbf{Inconc} \subseteq S^{TC}$ denoting verdicts. The actions

of $TC$ are partitioned $A^{TC} = A_I^{TC} \cup A_O^{TC}$, where $A_O^{TC} \subseteq A_I^M$ ($TC$ emits only inputs of $M$) and $A_I^{TC} \subseteq A_O^{IUT} \cup \{\delta\}$ ($TC$ captures outputs and quiescences of the IUT). A test case must satisfy several structural properties, detailed in [15].

The test generation technique of TGV is based upon *test purposes*, which allow to guide the test case selection. A test purpose is a deterministic and complete IOLTS $TP = (S^{TP}, A^{TP}, T^{TP}, s_0^{TP})$, with the same actions as the specification $A^{TP} = A^M$, and equipped with two sets of trap states $Accept^{TP}$ and $Reject^{TP}$, which are used to select targeted behaviours and to cut the exploration of $M$, respectively. Here we focus on the computation of the *complete test graph* (CTG), which contains all test cases corresponding to a specification and a test purpose, and therefore can serve as a criterion for comparison and performance measures.

The CTG is produced by TGV as the result of three operations, all performed on-the-fly: (a) computation of the synchronous product $SP = M \times TP$ between the IOLTSs of the specification and the test purpose, in order to mark accepting and refusal states; (b) suspension and determinization of $SP$, leading to $SP^{vis} = det(\Delta(SP))$, which keeps only visible behaviours and quiescence; (c) selection of the test cases denoting the behaviours of $SP^{vis}$ accepted by $TP$, which form the CTG. The main operation (c) roughly consists in computing $L2A$, the subset of the states of $SP^{vis}$ from which an accepting state is reachable (*lead to accept*), checking whether the initial state of $SP^{vis}$ belongs to $L2A$ (which indicates the existence of a test case), and defining, based upon $L2A$, the sets **Pass**, **Fail**, and **Inconc** corresponding to the verdicts. This is the operation we chose to encode as a BES resolution with diagnostic.

Assuming that the accepting states of $SP^{vis}$ are marked by self-looping transitions labeled by an action $acc$ (as it is done in practice), the reachability of an accepting state is denoted by the following $\mu$-calculus formula:

$$\phi_{acc} = \mu Y.(\langle acc \rangle \, \mathsf{true} \vee \langle - \rangle \, Y)$$

The set $L2A$ contains all states satisfying $\phi_{acc}$. It can be computed in a backwards manner by using a fixed point iteration to evaluate $\phi_{acc}$ on $SP^{vis}$. Since this requires the prior computation of $SP^{vis}$, we seek another solution suitable for on-the-fly exploration, by considering the formula below:

$$\phi_{l2a} = \nu X.(\phi_{acc} \wedge [-] \, (\phi_{acc} \Rightarrow X))$$

Formula $\phi_{l2a}$ has the same interpretation as $\phi_{acc}$, meaning that its satisfaction by the initial state of $SP^{vis}$ denotes the existence of a test case. Moreover, the on-the-fly evaluation of $\phi_{l2a}$ on a state $s$ satisfying $\phi_{acc}$ requires the inspection of every successor $s'$ of $s$ and, if it satisfies $\phi_{acc}$, the recursive evaluation of $\phi_{l2a}$ on $s'$, etc., until all states in $L2A$ have been explored.

The CTG could be obtained as the diagnostic produced by an on-the-fly model checker (such as EVALUATOR) for the formula $\phi_{l2a}$. However, to annotate the CTG with verdict information and to avoid redundancies caused by the two occurrences of $\phi_{acc}$ present in $\phi_{l2a}$, a finer-grained encoding of the problem in terms of a BES resolution with diagnostic is preferred. The corresponding BES

denotes the model checking problem of $\phi_{l2a}$ on $SP^{vis}$, by applying the translation given in Section 3.1 ($s, s'$ are states of $SP^{vis}$):

$$\{X_s =_\nu Y_s \wedge \bigwedge_{s \to s'} (Z_{s'} \vee X_{s'})\}, \{Y_s =_\mu \bigvee_{s \xrightarrow{acc} s'} \mathsf{true} \vee \bigvee_{s \to s'} Y_{s'}\},$$

$$\{Z_s =_\nu \bigwedge_{s \xrightarrow{acc} s'} \mathsf{false} \wedge \bigwedge_{s \to s'} Z_{s'}\}$$

If $X_{s_0^{vis}}$ is true, then a positive diagnostic (example) can be exhibited in the form of a boolean subgraph [23] containing, for each conjunctive variable (such as $X_s$ and $Z_s$), all its successor variables, and for each disjunctive variable (such as $Y_s$) only one successor which evaluates to true. This diagnostic can be obtained by another forward traversal of the boolean graph portion already explored for evaluating $X_{s_0^{vis}}$. We turn it into a CTG in the following manner: we associate a state of the CTG to each variable $X_s$; we produce an accepting transition going out of $X_s$ only if the first subformula in the right-hand side of the equation defining $Y_s$ is true (i.e., $s$ has an *acc*-successor); we produce a transition $X_s \xrightarrow{a} X_{s'}$ for each state $s'$ such that $Z_{s'}$ is false. Note that the diagnostic for variables $Z_s$ does not need to be explored. Additional verdict information in the form of refuse and inconclusive transitions is produced in a similar way during disgnostic generation, since the information needed for verdicts in the CTG is local w.r.t. states of *L2A* [15].

In the discussion above, formula $\phi_{l2a}$ was evaluated on the IOLTS $SP^{vis}$ obtained after suspension and determinization of *SP*; however, these two operations can also be performed *after* test case selection. In other words, the BES based generation procedure sketched above can be applied directly on the synchronous product *SP* between the specification and the test purpose, producing a 'raw' CTG, which is subsequently suspended and determinized to yield the final CTG. This procedure underlies the EXTRACTOR tool we developed within CADP for producing raw CTGs, which are then processed by the DETERMINATOR tool [12], resulting in CTGs strongly bisimilar to those produced by TGV. Although this ordering of operations is not the most efficient one for sequential on-the-fly test case generation (since the IOLTS of the specification can contain large amounts of $\tau$-transitions), it appears to be suitable for the distributed setting, since it leads to large BESs solved efficiently by our distributed algorithm MB-DSOLVE.

## 4 Implementation and Experiments

The model checker EVALUATOR 3.5 [26, 24] and the test case generator EXTRACTOR (see Figure 3) have been developed within CADP [11] by using the generic OPEN/CÆSAR environment [10] for on-the-fly exploration of LTSs.

EVALUATOR (*resp.* EXTRACTOR) consists of two parts: a front-end, responsible for encoding the verification of the $L^1_\mu$ formula (*resp.* the test selection guided by the test purpose $\mathrm{LTS}_2$) on $\mathrm{LTS}_1$ as a BES resolution, and for producing a counterexample (*resp.* a CTG) by interpreting the diagnostic provided by the BES resolution; and a back-end, responsible of BES resolution, playing the role of verification engine.

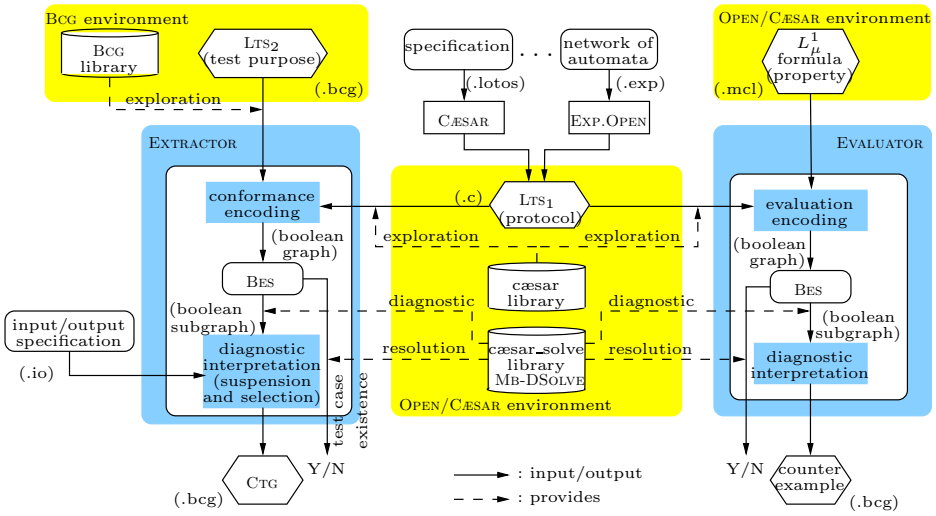



**Fig. 3.** The distributed on-the-fly tools EVALUATOR and EXTRACTOR

Sequential and distributed versions of EVALUATOR and EXTRACTOR are obtained by using as back-end either the sequential algorithms of the CÆSAR_SOLVE library [24, 25], or the MB-DSOLVE algorithm, respectively.

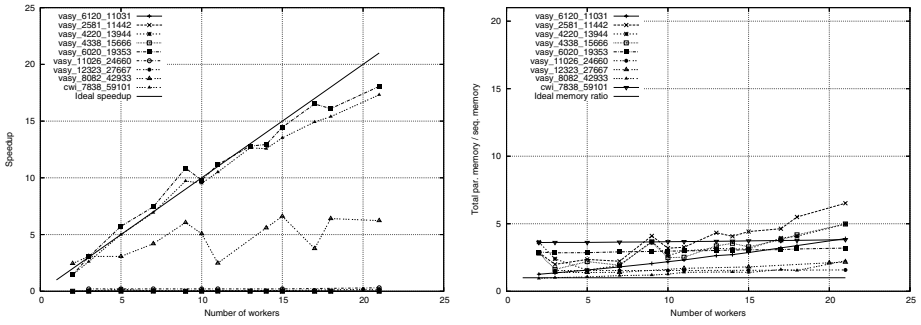
MB-DSOLVE (15 000 lines of C code) is a conservative extension of the distributed resolution algorithm DSOLVE [17] for single block BESs and was implemented by using the OPEN/CÆSAR environment. The size of the worker and supervisor processes is roughly the double in MB-DSOLVE w.r.t. DSOLVE. For communication, MB-DSOLVE is also based on the CÆSAR_NETWORK library of CADP, which offers a set of 40 primitives finely-tuned for verification problems, such as non-blocking asynchronous emission/reception of messages through TCP/IP sockets and explicit memory management by means of bounded communication buffers.

We present in this section experimental measures comparing the distributed versions of EVALUATOR and EXTRACTOR with their sequential counterparts and (as regards EVALUATOR) with the UPPDMC distributed model checker.

### 4.1 Performance of distributed model checking

We begun our experiments by checking two simple properties expressed in modal $\mu$-calculus, namely absence of deadlock ($\nu X.\,(\langle - \rangle\,\mathsf{true} \wedge [-]\,X)$) and presence of livelock ($\mu X.\,(\nu Y.\,(\langle \tau \rangle\,Y) \vee \langle - \rangle\,X)$), on a cluster of 21 XEON 2.4 GHZ PCS, with 1.5 GB of RAM, running LINUX, and interconnected by a 1 Gigabit ETHERNET network. These properties were checked on the largest LTSS of the VLTS benchmark. Figure 4 shows the speedup and memory ratio between distributed EVALUATOR and its sequential optimized version based on the reso-

lution algorithms for disjunctive/conjunctive BESs present in CÆSAR_SOLVE [24, 25]. The sequential version is very fast in finding counterexamples, but when it is necessary to explore the underlying BES entirely (e.g., for *cwi_7838_59101*), the distributed version becomes interesting, allowing close to linear speedups and a good scalability as the number of workers increases. The memory overhead of the distributed version is not really affected by an increasing number of workers and remains low, with an averaged memory consumption around 5 times bigger than the sequential one. Moreover, we observed almost no idle time, the distributed computation using systematically around 99% of the CPUs capacity on each worker node. This is partly a consequence of the well-balanced data partitioning induced by the static hash function, and indicates a good overlapping between communications and computations.




(a) (b)

**Fig. 4.** Speedup (a) and memory overhead (b) of distributed w.r.t. sequential EVALUATOR when checking absence of deadlock

We have also compared time and memory performances of distributed EVALUATOR against the UPPDMC model checker based on game graphs. Results are given in Table 1, where each VLTS example considered (e.g., *vasy_2581_11442*, an LTS with $2\,581 \cdot 10^3$ states and $11\,442 \cdot 10^3$ transitions) is checked for deadlock and livelock. Distributed EVALUATOR is very fast in detecting counterexamples, which explains most of the improvements in time and memory compared to UPPDMC. When the whole BES (*resp.* game graph) has to be explored (e.g., for *vasy_6020_19353*), the execution time is closer to that of UPPDMC, but always remains between 2 and 8 times lower. In this case, the memory consumption of distributed EVALUATOR is slightly greater w.r.t. UPPDMC; this is due to the simple data structures used for storing backward dependencies (linked lists) and could be reduced by using more compact data structures (e.g., packet lists).

| EXAMPLE | absence of deadlock | | | | | presence of livelock | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | truth | U (s) | U (MB) | E (s) | E (MB) | truth | U (s) | U (MB) | E (s) | E (MB) |
| *vasy_2581_11442* | false | 44 | 461 | 2 | 272 | false | 47 | n.c. | 7 | 844 |
| *vasy_4338_15666* | false | 64 | 745 | 2 | 313 | false | 64 | n.c. | 11 | 1 203 |
| *vasy_6020_19353* | true | 59 | 1 085 | 24 | 1 239 | true | 125 | n.c. | 8 | 1 442 |
| *vasy_6120_11031* | false | 95 | 947 | 1 | 170 | false | 108 | n.c. | 13 | 1 092 |
| *cwi_7838_59101* | true | 149 | 1 531 | 46 | 2 298 | true | 314 | n.c. | 16 | 2 793 |
| *vasy_8082_42933* | false | 162 | 1 374 | 2 | 268 | false | 134 | n.c. | 24 | 2 401 |

**Table 1.** Execution time (in seconds) and memory consumption (in MB) of two distributed on-the-fly model checkers: UPPDMC (U) with 25 nodes and EVALUATOR (E) with 21 nodes

We further experimented the scalability of distributed EVALUATOR by considering LTSs of increasing size and more complex properties taken from the CADP demos[2]. For instance, we used the following formula, stating that after a *put* action, either a livelock, or a *get* action will be eventually reached:

$$\nu X.\,([put]\,\mu Y.\,((\nu Z.(\langle\tau\rangle\,Z) \vee (\langle -\rangle\,\mathsf{true} \wedge [\overline{get}]\,\mathsf{false})) \vee (\langle -\rangle\,\mathsf{true} \wedge [-]\,Y)) \wedge [-]\,X)$$

Using distributed EVALUATOR on 10 machines, we successfully checked in 90 seconds that an LTS with 6 067 712 states and 19 505 146 transitions (modelling the behaviour of a communication protocol that exchanges 256 different messages) satisfies this property, whereas the sequential version (based onDFS algorithm) of EVALUATOR achieved it in a prohibitive time, e.g., more than 15 minutes. When using the sequential version (based on BFS) of EVALUATOR, the advantage of distributed EVALUATOR over sequential solution is emphasized since data is equally splited over a network of machines, whereas sequential BFS evaluations need to store locally a lot of information, and are thus more prone to memory shortage than DFS traversals.

### 4.2 Performance of distributed conformance test case generation

Symetrically to the model checking experiments presented in Section 3.1, we adopted a generic approach for experimenting conformance test case generation. We used a generic test purpose (e.g., "after 10 visible actions, an accepting state is reachable") and several LTSs taken from the VLTS benchmark as inputs for our distributed test case generator tool EXTRACTOR, and studied its behaviour on generating corresponding raw CTGs, subsequently suspended and determinized using DETERMINATOR. Distributed EXTRACTOR exhibits a good behavior (with significant speedup) for the resolution of the underlying BES, and generates CTGs strongly equivalent to those produced by TGV. Further experiments are ongoing in order to get a precise comparison with TGV in terms of speedup and memory consumption.

We also experimented the generic test purpose on an LTS with 237 500 states and 760 794 transitions (modelling the behaviour of a communication proto-

[2] `http://www.inrialpes.fr/vasy/cadp/demos.html`

col that exchanges 50 different messages) using each of the three CTG generators, namely TGV, sequential and distributed versions of EXTRACTOR. The CTG produced by TGV contains 10 402 states, 33 052 transitions, whereas the CTG produced by both sequential and distributed EXTRACTOR and processed by DETERMINATOR has 20 703 states and 66 002 transitions.

As regards scalability of distributed EXTRACTOR, we could resolve on 16 machines the BES corresponding to the generic test purpose and an LTS with 6 067 712 states and 19 505 146 transitions (previously used for model checking) in less than 372 seconds, whereas sequential EXTRACTOR took more than 30 minutes, and we stopped the execution of TGV after an hour of computation. Another concern is the good memory consumption exhibited by EXTRACTOR tool for both sequential and distributed versions. An example is the VLTS benchmark *cwi_2165_8723* on which TGV had a extremely rapid memory shortage (less than a minute of computation) whereas EXTRACTOR was still computing after 25 minutes.

## 5 Conclusion and Future Work

Building efficient and generic verification components is crucial for facilitating the development of robust explicit-state analysis tools. Our MB-DSOLVE algorithm for distributed on-the-fly resolution of multiple block, alternation-free BESs, goes towards this objective. MB-DSOLVE was designed to be compliant with the interface of the BES resolution library CÆSAR_SOLVE [24, 25], thus being directly available as verification back-end for all analysis tools based on CÆSAR_SOLVE. Here we illustrated its application for alternation-free $\mu$-calculus model checking and conformance test generation, as distributed computing engine for the tools EVALUATOR [26, 24] and EXTRACTOR, developed within CADP [11] using the generic OPEN/CÆSAR environment [10] for LTS exploration.

The modular architecture of these tools does not penalize their performance. Our experiments using large state spaces from the VLTS benchmarks have shown that distributed EVALUATOR compares favourably in terms of speed and memory with UPPDMC, the other existing implementation of distributed on-the-fly model checking, based on game graphs [13]. Moreover, distributed EVALUATOR exhibits a good speedup and scalability w.r.t. its sequential version, relying on the optimized algorithms of CÆSAR_SOLVE for disjunctive/conjunctive BESs. Distributed EXTRACTOR, to our knowledge the first tool of its kind to perform distributed on-the-fly conformance test generation, allows to scale up the capabilities of well-established dedicated tools, such as TGV [15].

We plan to continue our work along two directions. First, we will study other distributed resolution strategies, aiming at reducing memory consumption for disjunctive/conjunctive BESs, which occur frequently in practice [24]: one such strategy could combine a distributed breadth-first and a local depth-first exploration of the boolean graph. Next, we will consider the encoding of other problems, such as discrete controller synthesis, in terms of BES resolution with diagnostic, following, e.g., the approach proposed in [32].

## References


1. H. R. Andersen. Model checking and boolean graphs. *Th. Comp. Sci.*, 126(1):3–30, April 1994.
2. H. R. Andersen and B. Vergauwen. Efficient Checking of Behavioural Relations and Modal Assertion s using Fixed-Point Inversion. In *Proc. of CAV'95*, LNCS vol. 939, pp. 142–154.
3. D. Bergamini, N. Descoubes, Ch. Joubert, and R. Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In *Proc. of Tacas'05*, LNCS vol. 3440, pp. 581–585.
4. B. Bollig, M. Leucker, and M. Weber. Local Parallel Model Checking for the Alternation Free Mu-Calculus. Technical Report AIB-04-2001, Aachen University of Technology, 2001.
5. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Trans. on Prog. Lang. and Systems* 8(2):244–263, 1986.
6. R. Cleaveland and B. Steffen. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. In *Proc. of CAV'91*, LNCS vol. 575, pp. 48–58.
7. X. Du, S. A. Smolka, and R. Cleaveland. Local Model Checking and Protocol Analysis. *Springer Int. J. on Software Tools for Technology Transfer (STTT)*, 2(3):219–241, 1999.
8. E. A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proc. of the 1st LICS*, pp. 267–278, 1986.
9. M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *J. of Computer and System Sciences*, 18(2):194–211, 1979.
10. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proc. of TACAS'98*, LNCS vol. 1384, pp. 68–84.
11. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *European Assoc. for Software Science and Technology (EASST) Newsletter*, 4:13–24, 2002.
12. H. Hermanns and Ch. Joubert. A Set of Performance and Dependability Analysis Components for CADP. In *Proc. of TACAS'03*, LNCS vol. 2619, pp. 425–430.
13. F. Holmén, M. Leucker, and M. Lindström. UppDMC – A Distributed Model Checker for Fragments of the $\mu$-calculus. In *Proc. of PDMC'04*, ENTCS vol. 128, pp. 91–105.
14. S. T. Huang and P. W. Kao. Detecting Termination of Distributed Computations by External Agents. *J. of Inf. Sci. and Engineering*, 7(2):187–201, 1991.
15. C. Jard and T Jéron. TGV: Theory, Principles and Algorithms. *Springer Int. J. on Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005.
16. Ch. Joubert and R. Mateescu. Distributed On-the-Fly Equivalence Checking. In *Proc. of PDMC'04*, ENTCS vol. 128, pp. 47–62.
17. Ch. Joubert and R. Mateescu. Distributed Local Resolution of Boolean Equation Systems. In *Proc. of PDP'05*. IEEE Computer Society Press.
18. D. Kozen. Results on the Propositional $\mu$-calculus. *Th. Comp. Sci.*, 27:333–354, 1983.
19. K. G. Larsen. Proof Systems for Hennessy-Milner logic with Recursion. In *Proc. of CAAP'88*, LNCS vol. 299, pp. 215–230.
20. K. G. Larsen. Efficient Local Correctness Checking. In *Proc. of CAV'92*, LNCS vol. 663, pp. 30–43.

21. M. Leucker, R. Somla, and M. Weber. Parallel Model Checking for LTL, CTL$^*$ and $L^2_\mu$. In *Proc. of PDMC'03*, ENTCS vol. 89, pp. 4–16.
22. A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. VERSAL 8, Bertz Verlag, Berlin, 1997.
23. R. Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In *Proc. of TACAS'00*, LNCS vol. 1785, pp. 251–265.
24. R. Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In *Proc. of TACAS'03*, LNCS vol. 2619, pp. 81–96.
25. R. Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *Springer Int. J. on Software Tools for Technology Transfer (STTT)*, 2006.
26. R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Sci. of Comp. Programming*, 46(3):255–281, 2003.
27. F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2:161–175, 1987.
28. G. Pace, F. Lang, and R. Mateescu. Calculating $\tau$-Confluence Compositionally. In *Proc. of CAV'03*, LNCS vol. 2725, pp. 446–459.
29. P. Stevens and C. Stirling. Practical Model-Checking using Games. In *Proc. of TACAS'98*, LNCS vol. 1384, pp. 85–101.
30. J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.
31. B. Vergauwen and J. Lewi. Efficient Local Correctness Checking for Single and Alternating Boolean Equation Systems. In *Proc. of ICALP'94*, LNCS vol. 820, pp. 304–315.
32. R. Ziller and K. Schneider. Combining Supervisor Synthesis and Model Checking. *Trans. on Embedded Computing Sys.*, 4(2):331–362, 2005.