A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems*

Radu Mateescu

INRIA Rhône-Alpes / VASY, 655, avenue de l'Europe
F-38330 Montbonnot Saint Martin, France
Radu.Mateescu@inria.fr

Abstract. Boolean Equation Systems (BESs) offer a useful representation for various verification problems on finite-state concurrent systems, such as equivalence/preorder checking and model checking. In particular, on-the-fly resolution methods enable a demand-driven construction of the BES (and hence, of the state space) during verification. In this paper, we present a generic library dedicated to on-the-fly resolution of alternation-free BESs. Four resolution algorithms are currently provided by the library: A1, A2 are general, the latter being optimized to produce small-depth diagnostics, and A3, A4 are specialized for handling acyclic and disjunctive/conjunctive BESs in a memory-efficient way. The library is developed within the CADP toolbox and serves as engine for on-the-fly equivalence/preorder checking modulo five widely-used relations, and for model checking of alternation-free μ-calculus.

1 Introduction

Boolean Equation Systems (BESs) [15] are a well-studied framework for the verification of concurrent finite-state systems, by allowing to formulate model checking and equivalence/preorder checking problems in terms of BES resolution. Numerous algorithms for solving BESs have been proposed (see [15, chap. 6] for a survey). They can be basically grouped in two classes: *global* algorithms, which require the BES to be constructed entirely before the resolution, and *local* (or *on-the-fly*) algorithms, which allow the BES to be generated dynamically during the resolution. Local algorithms are able to detect errors in complex systems even when the corresponding BESs are too large to be constructed explicitly. Another feature is the generation of *diagnostics* (portions of the BES explaining the truth value of a variable), which provide considerable help for debugging applications and for understanding temporal logic formulas [16].

However, as opposed to the situation in the field of symbolic verification, for which a significant number of BDD-based packages are available [24], we are not aware of any generic environment for BES resolution available for on-the-fly verification. In this paper we present CÆSAR_SOLVE, a generic library for

* This research was partially funded by the IST-2001-32360 Project "ArchWare" and by Bull S.A.

Bes resolution and diagnostic generation, created using the Open/Cæsar environment for on-the-fly verification [14]. Cæsar_Solve provides an application-independent representation of Bess as *boolean graphs* [1], much in the same way as Open/Cæsar provides a language-independent representation of Labeled Transition Systems (Ltss). Four algorithms are currently available in the library. Algorithms A1 and A2 are general (they do not assume anything about the right-hand sides of the equations), A2 being optimized to produce small-depth diagnostics. Algorithms A3 and A4 are specialized for memory-efficient resolution of acyclic Bess and disjunctive/conjunctive Bess, which occur frequently in practice. Cæsar_Solve serves as engine for two on-the-fly verification tools developed within the Cadp toolbox [10]: the equivalence/preorder checker Bisimulator, which implements five widely-used equivalence relations, and the model checker Evaluator for regular alternation-free μ-calculus [18].

The paper is organized as follows. Section 2 defines alternation-free Bess. Section 3 presents algorithms A1–A4 and compares them according to three criteria which aim at improving time complexity. Section 4 outlines the encodings of various equivalence relations and temporal logics in terms of alternation-free Bess, identifying the particular cases suitable for algorithms A3 and A4. Section 5 shows the architecture of the library and some performance measures. Section 6 summarizes the results and indicates directions for future work.

2 Alternation-Free Boolean Equation Systems

A Boolean Equation System (Bes) [1,15] 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 $M_i = \{X_j \overset{\sigma_i}{=} op_j \boldsymbol{X}_j\}_{j \in [1,m_i]}$ 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 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 F and T abbreviate the empty disjunction $\vee\emptyset$ and the empty conjunction $\wedge\emptyset$.

The *main* variable X must be defined in block M_1. 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 semantics $[\![op_i\{X_1, ..., X_k\}]\!]\delta$ of a formula $op_i\{X_1, ..., X_k\}$ w.r.t. $\mathbf{Bool} = \{\mathsf{F}, \mathsf{T}\}$ and a context $\delta : \mathcal{X} \to \mathbf{Bool}$, 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 δ is the σ_i-fixed point of a vectorial functional $\Phi_{i\delta} : \mathbf{Bool}^{m_i} \to \mathbf{Bool}^{m_i}$ 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 $\delta \oslash [b_1/X_1, ..., b_n/X_n]$ denotes a context identical to δ except for variables $X_1, ..., X_n$, which are assigned values $b_1, ..., b_n$, 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 δ_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 $k > i$).

A block is *acyclic* if the dependency graph induced by its equations is acyclic. A variable X_j is called *disjunctive* (resp. *conjunctive*) if $op_j = \vee$ (resp. $op_j = \wedge$). A block M_i is disjunctive (resp. conjunctive) if each of its variables either is disjunctive (resp. conjunctive), or it depends upon at most one variable defined in M_i, its other dependencies being constants or variables defined in other blocks.

The 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 on-the-fly BES resolution algorithms are available [6,1,15,7]. Here we follow an approach proposed in [1], which proceeds as follows. To each block M_i is associated a resolution routine R_i responsible for computing the values of M_i's variables. When a variable X_j of M_i is computed by a call $R_i(X_j)$, the values of other variables X_l defined in other blocks M_k may be needed; these values are computed by calls $R_k(X_l)$ of the routine associated to M_k. This process always terminates, because there are no cyclic dependencies between blocks (the call stack of resolution routines has a size bounded by the depth of the dependency graph between blocks). Since a variable X_j of M_i may be required several times during the resolution process, the computation results must be kept persistent between subsequent calls of R_i to obtain an efficient overall resolution.

Compared to other algorithms like LMC [7], which consists of a single routine handling the whole BES, the scheme above presents two advantages: (a) the algorithms used in the resolution routines of individual blocks are simpler, since they must handle a single type of fixed point equations; (b) the overall resolution process is easier to optimize, simply by designing more efficient algorithms for blocks with particular structure (e.g., acyclic, disjunctive or conjunctive).

3 On-the-Fly Resolution Algorithms

This section presents four different algorithms implementing the on-the-fly resolution of individual equation blocks in an alternation-free BES. The algorithms are defined only for μ-blocks, those for ν-blocks being completely dual. Algorithms A1 and A2 are general (they do not depend upon the structure of the right-hand sides of the equations), whereas algorithms A3 and A4 are optimized for acyclic blocks and for disjunctive or conjunctive blocks, respectively.

We develop the resolution algorithms in terms of *boolean graphs* [1], which provide a graphical, more intuitive representation of BESs. Given an equation block $M_i = \{X_j \overset{\mu}{=} op_j \boldsymbol{X}_j\}_{j \in [1, m_i]}$, the corresponding boolean graph is a tuple $G = (V, E, L)$, where: $V = \{X_j \mid j \in [1, m_i]\}$ is the set of *vertices* (boolean variables), $E = \{X_j \to X_k \mid j \in [1, m_i] \wedge X_k \in \boldsymbol{X}_j\}$ is the set of *edges* (dependencies between variables), and $L : V \to \{\vee, \wedge\}$, $L(X_j) = op_j$ is the *vertex labeling* (disjunctive or conjunctive). The set of successors of a vertex x is noted $E(x)$. Sink $\vee$-vertices (resp. $\wedge$-vertices) represent variables equal to F (resp. T). During a call of the resolution routine R_i associated to block M_i, all variables

X_l defined in other blocks M_k and occurring free in M_i can be seen as constants, because their values are computed on-the-fly by calls to R_k.

As expected, the boolean graphs associated to acyclic blocks are acyclic. The graphs associated to disjunctive (resp. conjunctive) blocks may contain $\wedge$-vertices (resp. $\vee$-vertices) having at most one successor (these vertices correspond either to constants, or to variables having at most one non-constant successor in the current block), the other vertices being disjunctive (resp. conjunctive).

The algorithms we present are all based upon the same principle: starting at the variable of interest, they perform an on-the-fly, forward exploration of the boolean graph and propagate backwards the values of the "stable" variables (i.e., whose final value has been determined); the propagation of a T (resp. a F) backwards to a $\vee$-variable (resp. $\wedge$-variable) makes it T (resp. F). The algorithms terminate either when the variable of interest becomes stable, or the entire boolean graph is explored. To compare the different algorithms, we precise below three requirements desirable for obtaining a good time complexity:

(R1) The resolution of a variable (vertex of the boolean graph) must be carried out in a time linear in the size of the graph, i.e., $O(|V| + |E|)$. This is necessary for obtaining a linear time overall resolution of a multiple-block, alternation-free Bes.

(R2) During the resolution of a variable, every new variable explored must be related to the variable of interest by (at least) a path of unstable variables in the boolean graph. This limits the graph exploration only to variables "useful" for the current resolution.

(R3) When a call of the resolution algorithm terminates, the portion of the boolean graph explored must be stable. This avoids that subsequent calls for solving the same variable lead to multiple explorations of the graph (which may destroy the overall linear time complexity).

3.1 Algorithm A1 (DFS, General)

Algorithm A1 is based upon a depth-first search (Dfs) of the boolean graph. It satisfies all three aforementioned requirements: (R1) its worst-case time and space complexity is $O(|V| + |E|)$, because every edge in the boolean graph is traversed at most twice: forwards, when its source variable is explored, and backwards, when the value of its target variable (if it became stable) is back-propagated; (R2) new variables, explored from the top of the Dfs stack, are related to the variable of interest, which is at the bottom of the Dfs stack, via the unstable variables present on the stack; (R3) the portion of boolean graph explored after each call of the algorithm contains only stable variables, i.e., depending only upon variables already explored.

The algorithm can be seen as an optimized version of the Avoiding 1's algorithm proposed in [1]: it is implemented iteratively rather than recursively, it has a better average complexity because values of variables are back-propagated as soon as they become stable, and it has a lower memory consumption because dependencies between variables are discarded during back-propagation. A1 was initially developed for model-checking regular alternation-free μ-calculus [18].

3.2 Algorithm A2 (BFS, General)

Algorithm A2 (see Figure 1) is based upon a breadth-first search (Bfs) of the boolean graph, starting from the variable of interest x. Visited vertices are stored in a set $A \subseteq V$ and visited but unexplored vertices are stored in a queue. To each vertex y are associated two informations: a counter $c(x)$, which keeps the number of y's successors that must become true in order to make y true ($c(y)$ is initialized to $|E(y)|$ if y is a $\wedge$-vertex and to 1 otherwise) and a set $d(y)$ containing the vertices that currently depend upon y. At each iteration of the main while-loop (lines 4–34), the vertex y in front of the queue is explored. If it is already stable (i.e., $c(y) = 0$), its value is back-propagated by the inner while-loop (lines 8–20) along the dependencies d; otherwise, all successors $E(y)$ are visited and (if they are stable or new) are inserted at the end of the queue.

The algorithm satisfies requirement (R1), since each call has a complexity $O(|V|+|E|)$. It does not satisfy (R2), because the back-propagation may stabilize vertices that "cut" all the paths relating x to vertices in the queue, and thus at some points the algorithm may explore vertices useless for deciding the truth value of x (however, the values of these vertices may be useful in later calls of A2). Finally, it satisfies (R3), since at the end of the main while-loop all visited vertices are stable (they depend only upon the vertices in A). These observations are confirmed experimentally, A2 being slightly slower than A1.

However, as regards the ability of generating positive diagnostics (examples) of small size, A2 performs better than A1. During the back-propagation carried out by the inner while-loop, to each $\vee$-vertex w that becomes stable is associated its successor $s(w)$ that made it stable (line 14). This information can be used to construct a diagnostic for x at the end of the algorithm, by performing another traversal of the subgraph induced by A and keeping the successors given by s (for $\vee$-vertices) or all successors (for $\wedge$-vertices) [16]. Being Bfs-based, A2 generally produces examples of smaller depth than A1, and even of minimal depth when the examples are sequences (e.g., in the case of disjunctive blocks). Of course, the same situation occurs in the dual case, when A2 is used for producing negative diagnostics (counterexamples) for ν-blocks.

3.3 Algorithm A3 (DFS, Acyclic)

Algorithm A3 is based upon a Dfs of the boolean graph and is specialized for solving acyclic equation blocks. It is quite similar to algorithm A1, except that it does not need to store dependencies between variables, since back-propagation takes place only along the Dfs stack (the boolean graph being acyclic, variables become stable as soon as they are popped from the Dfs stack). Therefore, algorithm A3 has a worst-case memory consumption $O(|V|)$, improving over the general algorithms A1 and A2.

Being Dfs-based, algorithm A3 satisfies all requirements (R1)–(R3). A3 was initially developed for model-checking μ-calculus formulas on large traces obtained by intensive simulation of a system implementation [17].

```
function A2 ($x$, $(V, E, L)$) : Bool is
  $c(x)$ := if $L(x) = \wedge$ then $|E(x)|$ else 1 endif;
  $d(x) := \emptyset$; $A := \{x\}$; $queue := put(x, nil)$;
  while $queue \neq nil$ do
    $y := head(queue)$; $queue := tail(queue)$;
    if $c(y) = 0$ then
      $B := \{y\}$;
      while $B \neq \emptyset$ do
        let $u \in B$; $B := B \setminus \{u\}$;
        forall $w \in d(u)$ do
          if $c(w) > 0$ then
            $c(w) := c(w) - 1$;
            if $c(w) = 0$ then
              if $L(w) = \vee$ then $s(w) := u$ endif;
              $B := B \cup \{w\}$
            endif
          endif
        end;
        $d(u) := \emptyset$
      end
    else
      forall $z \in E(y)$ do
        if $z \in A$ then
          $d(z) := d(z) \cup \{y\}$;
          if $c(z) = 0$ then
            $queue := put(z, queue)$
          endif
        else
          $c(z)$ := if $L(z) = \wedge$ then $|E(z)|$ else 1 endif;
          $d(z) := \{y\}$; $A := A \cup \{z\}$; $queue := put(z, queue)$
        endif
      end
    endif
  end;
  return $c(x) = 0$
end
```

Fig. 1. Algorithm A2: BFS-based local resolution of a μ-block

3.4 Algorithm A4 (DFS, Disjunctive/Conjunctive)

Algorithm A4 (see Figure 2) is based upon a DFS of the boolean graph, performed recursively starting from the variable of interest x. A4 is specialized for solving disjunctive or conjunctive blocks; we show only its variant for disjunctive blocks, the other variant being dual. For simplicity, we assume that all $\wedge$-vertices of the disjunctive block have no successors (i.e., they are T): since each $\wedge$-vertex may have at most one non-constant successor in the block, it can be assimilated to a $\vee$-vertex if its other successors are evaluated first (possibly by calling the resolution routines of other blocks). In this case, solving a disjunctive block amounts to searching for a sink $\wedge$-vertex, since a T value will propagate back to x via $\vee$-vertices. This algorithm obviously meets requirements (R1) and (R2).

```
$A := \emptyset$; $n := 0$; $stack := nil$;
function A4 $(x, (V, E, L))$ : Bool is
  $A := A \cup \{x\}$; $n(x) := n$; $n := n + 1$;
  $stack := push(x, stack)$; $low(x) := n(x)$;
  if $|E(x)| = 0$ then
    $v(x) :=$ if $L(x) = \wedge$ then $\mathsf{T}$ else $\mathsf{F}$ endif; $stable(x) := \mathsf{T}$
  else
    $v(x) := \mathsf{F}$; $stable(x) := \mathsf{F}$
  endif;
  forall $y \in E(x)$ do
    if $y \in A$ then
      $val := v(y)$;
      if $\neg stable(y) \wedge n(y) < n(x)$ then
        $low(x) := min(low(x), n(y))$
      endif
    else
      $val :=$ A4 $(y, (V, E, L))$;
      $low(x) := min(low(x), low(y))$
    endif;
    if $val$ then
      $v(x) := \mathsf{T}$; $stable(x) := \mathsf{T}$; break
    endif
  end;
  if $v(x) \vee low(x) = n(x)$ then
    repeat
      $z := top(stack)$; $v(z) := v(x)$; $stable(z) := \mathsf{T}$;
      $stack := pop(stack)$
    until $z = x$
  endif;
  return $v(x)$
end
```

Fig. 2. Algorithm A4: DFS-based local resolution of a disjunctive μ-block

However, in order to guarantee requirement (R3), we must ensure that all visited vertices stored in $A \subseteq V$ are stable when x has been evaluated. This could be done by storing backward dependencies (as for algorithms A1 and A2), but for disjunctive blocks we can avoid this by computing the strongly connected components (SCCs) of the boolean graph. When x is evaluated to T, all vertices belonging to the SCC of x must become T (since they can reach x via a path of $\vee$-vertices) and the other ones must be stabilized to F.

Algorithm A4 combines the search for T vertices with a detection of Scc following Tarjan's classical algorithm. It proceeds as follows: for each successor y of vertex x (lines 10–23), it calculates its boolean value $v(y)$, its "lowlink" number $low(y)$, and a boolean $stable(y)$ which is set to F if y belongs to the current Scc and to T otherwise. Then, if $v(x) = \mathsf{T}$ or x is the root of a Scc, all vertices in the current Scc are stabilized to the value $v(x)$ (lines 24–29). In this way, algorithm A4 meets all requirements (R1)–(R3) and avoids to store transitions of the boolean graph, having a worst-case memory complexity $O(|V|)$.

4 Equivalence Checking and Model Checking

In this section we study two applications of Bes resolution in the field of finite-state verification: equivalence/preorder checking and model checking, both performed on-the-fly. Various encodings of these problems in terms of Bess have been proposed in the literature [5,1,15]. Here we aim at giving a uniform presentation of these results and also at identifying particular cases where the algorithms A3 and A4 given in Sections 3.3 and 3.4 can be applied.

4.1 Encoding Equivalence Relations

Labeled Transition Systems (Ltss) are natural models for action-based languages describing concurrency, such as process algebras. An Lts is a quadruple $M = (Q, A, T, q_0)$, where: Q is the set of states, A is the set of actions ($A_\tau = A \cup \{\tau\}$ is the set of actions extended with the invisible action τ), $T \subseteq Q \times A_\tau \times Q$ is the transition relation, and $q_0 \in Q$ is the initial state. A transition $(q_1, a, q_2) \in T$ (also noted $q_1 \xrightarrow{a} q_2$) means that the system can evolve from state q_1 to state q_2 by performing action a. The notation is extended to transition sequences: if $l \subseteq {A_\tau}^*$ is a language defined over A_τ, $q_1 \xrightarrow{l} q_2$ means that from q_1 to q_2 there is a sequence of transitions whose actions concatenated form a word of l.

Let $M_i = (Q_i, A, T_i, {q_0}_i)$ be two Ltss ($i \in \{1, 2\}$). The table below shows the Bes encodings of the equivalence between M_1 and M_2 modulo five widely-used equivalence relations: strong bisimulation [22], branching bisimulation [23], observational equivalence [19], $\tau^*.a$ equivalence [12], and safety equivalence [3]. These encodings are derived from the characterizations given in [12]. Each relation is represented as a Bes with a single ν-block defining, for each couple of states $(p, q) \in Q_1 \times Q_2$, a variable $X_{p,q}$ which expresses that p and q are equivalent ($a \in A$ and $b \in A_\tau$). For each equivalence relation, the corresponding preorder relation is obtained simply by dropping either the second conjunct (for strong, $\tau^*.a$, and safety equivalence), or the third and fourth conjuncts (for branching and observational equivalence) in the right-hand sides of the equations defining $X_{p,q}$ (e.g., the strong preorder is defined by the Bes $\{X_{p,q} \overset{\nu}{=} \bigwedge_{p \xrightarrow{b} p'} \bigvee_{q \xrightarrow{b} q'} X_{p',q'}\}$). Other equivalences, such as delay bisimulation [21] and η-bisimulation [2], can be encoded using a similar scheme. Note that for all weak equivalences, the computation of the right-hand sides of equations requires to compute transitive closures of τ-transitions in one or both Ltss.

Relation	Encoding
Strong	$\left\{ X_{p,q} \overset{\nu}{=} \left(\bigwedge_{p \xrightarrow{b} p'} \bigvee_{q \xrightarrow{b} q'} X_{p',q'}\right) \wedge \left(\bigwedge_{q \xrightarrow{b} q'} \bigvee_{p \xrightarrow{b} p'} X_{p',q'}\right) \right\}$
Branching	$\left\{ \begin{array}{l} X_{p,q} \overset{\nu}{=} \bigwedge_{p \xrightarrow{b} p'} \left((b = \tau \wedge X_{p',q}) \vee \bigvee_{q \xrightarrow{\tau^*} q' \xrightarrow{b} q''} (X_{p,q'} \wedge X_{p',q''})\right) \wedge \\ \qquad \bigwedge_{q \xrightarrow{b} q'} \left((b = \tau \wedge X_{p,q'}) \vee \bigvee_{p \xrightarrow{\tau^*} p' \xrightarrow{b} p''} (X_{p',q} \wedge X_{p'',q'})\right) \end{array} \right\}$
Observational	$\left\{ \begin{array}{l} X_{p,q} \overset{\nu}{=} \left(\bigwedge_{p \xrightarrow{\tau} p'} \bigvee_{q \xrightarrow{\tau^*} q'} X_{p',q'}\right) \wedge \left(\bigwedge_{p \xrightarrow{a} p'} \bigvee_{q \xrightarrow{\tau^* a \tau^*} q'} X_{p',q'}\right) \wedge \\ \qquad \left(\bigwedge_{q \xrightarrow{\tau} q'} \bigvee_{p \xrightarrow{\tau^*} p'} X_{p',q'}\right) \wedge \left(\bigwedge_{q \xrightarrow{a} q'} \bigvee_{p \xrightarrow{\tau^* a \tau^*} p'} X_{p',q'}\right) \end{array} \right\}$
$\tau^* . a$	$\left\{ X_{p,q} \overset{\nu}{=} \left(\bigwedge_{p \xrightarrow{\tau^* a} p'} \bigvee_{q \xrightarrow{\tau^* a} q'} X_{p',q'}\right) \wedge \left(\bigwedge_{q \xrightarrow{\tau^* a} q'} \bigvee_{p \xrightarrow{\tau^* a} p'} X_{p',q'}\right) \right\}$
Safety	$\left\{ \begin{array}{l} X_{p,q} \overset{\nu}{=} Y_{p,q} \wedge Y_{q,p} \\ Y_{p,q} \overset{\nu}{=} \left(\bigwedge_{p \xrightarrow{\tau^* a} p'} \bigvee_{q \xrightarrow{\tau^* a} q'} Y_{p',q'}\right) \end{array} \right\}$

In order to apply the resolution algorithms given in Section 3, the BESs shown in the table above must be transformed by introducing extra variables such that the right-hand sides of equations become disjunctive or conjunctive formulas. For example, the BES for strong bisimulation is transformed as follows:

$$\left\{ \begin{array}{l} X_{p,q} \overset{\nu}{=} \bigwedge_{p \xrightarrow{b} p'} Y_{b,p',q} \wedge \bigwedge_{q \xrightarrow{b} q'} Z_{b,p,q'} \\ Y_{b,p',q} \overset{\nu}{=} \bigvee_{q \xrightarrow{b} q'} X_{p',q'} \\ Z_{b,p,q'} \overset{\nu}{=} \bigvee_{p \xrightarrow{b} p'} X_{p',q'} \end{array} \right\}$$

This kind of BESs can be solved by using the general algorithms A1 and A2 (note that the encodings given above allow to construct both LTSs on-the-fly during BES resolution). However, when one or both LTSs M_1 and M_2 have a particular structure, the BESs can be simplified in order to make applicable the specialized algorithms A3 or A4.

Acyclic case. When M_1 or M_2 is acyclic, the BESs associated to strong bisimulation (and its preorder) become acyclic as well. This is easy to see for strong bisimulation: since the two-step sequences $X_{p,q} \rightarrow Y_{b,p',q} \rightarrow X_{p',q'}$ and $X_{p,q} \rightarrow Z_{b,p,q'} \rightarrow X_{p',q'}$ of the boolean graph correspond to transitions $p \xrightarrow{b} p'$ and $q \xrightarrow{b} q'$, a cycle $X_{p,q} \rightarrow \cdots X_{p,q}$ in the boolean graph would correspond to cycles $p \xrightarrow{b} \cdots p$ and $q \xrightarrow{b} \cdots q$ in both M_1 and M_2. For $\tau^* . a$ and safety equivalence (and their preorders), acyclic BESs are obtained when M_1 or M_2 contain no cycles going through visible transitions (but may contain τ-cycles): since two-step sequences in the boolean graph correspond to sequences of τ-transitions ended by a-transitions performed synchronously by the two LTSs, a cycle in the boolean graph would correspond to cycles containing an a-transition in both M_1 and M_2. For branching and observational equivalence (and their preorders), both LTSs M_1 and M_2 must be acyclic in order to get acyclic BESs, because τ-loops like $p \xrightarrow{\tau} p$ present in M_1 induce loops $X_{p,q} \rightarrow X_{p,q}$ in the boolean graph even if M_2 is acyclic.

If the above conditions are met, then the memory-efficient algorithm A3 can be used to perform equivalence/preorder checking. One practical application concerns the correctness of large execution traces produced by an implementation of a system w.r.t. the formal specification of the system [17]. Assuming the system specification given as an LTS M_1 and the set of traces given as an LTS M_2 (obtained by merging the initial states of all traces), the verification consists in checking the inclusion $M_1 \preceq M_2$ modulo the strong or safety preorder.

Conjunctive case. When M_1 or M_2 is deterministic, the BESs associated to the five equivalence relations considered and to their corresponding preorders can be reduced to conjunctive form. We illustrate this for strong bisimulation, the BESs of the other equivalences being simplified in a similar manner. If M_1 is deterministic, for every state $p \in Q_1$ and action $b \in A_\tau$, there is at most one transition $p \xrightarrow{b} p'_b$. Let $q \xrightarrow{b} q'$ be a transition in M_2. If there is no corresponding transition $p \xrightarrow{b} p'_b$ in M_1, the right-hand side of the equation defining $X_{p,q}$ trivially reduces to false (states p and q are not strongly bisimilar). Otherwise, the right-hand side of the equation becomes $\left(\bigvee_{q \xrightarrow{b} q'} X_{p'_b,q'}\right) \wedge \left(\bigwedge_{q \xrightarrow{b} q'} X_{p'_b,q'}\right)$, which reduces to $\bigwedge_{q \xrightarrow{b} q'} X_{p'_b,q'}$ since the first conjunct is absorbed by the second one. The same simplification applies when M_2 is deterministic, leading in both cases to a conjunctive BES.

For weak equivalences, further simplifications of the BESs can be obtained when one LTS is both deterministic and τ-free (i.e., without τ-transitions). For example, if M_1 is deterministic and τ-free, the BES for observational equivalence becomes $\left\{X_{p,q} \overset{\nu}{=} \bigwedge_{q \xrightarrow{\tau} q'} X_{p,q'} \wedge \bigwedge_{q \xrightarrow{a} q'} X_{p'_a,q'}\right\}$. These simplifications have been identified in [12]; we believe they can be obtained in a more direct way by using BES encodings.

When one of the above conditions is met, then the memory-efficient algorithm A4 can be used to perform equivalence/preorder checking. As pointed out in [12], when comparing the LTS M_1 of a protocol with the LTS M_2 of its service (external behaviour), it is often the case that M_2 is deterministic and/or τ-free.

4.2 Encoding Temporal Logics

Alternation-free BESs allow to encode the alternation-free μ-calculus [6,1,15]. The formulas of this logic, defined over an alphabet of propositional variables $X \in \mathcal{X}$, have the following syntax (given directly in positive form):

$$\varphi ::= \mathsf{F} \mid \mathsf{T} \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \langle a \rangle \, \varphi \mid [a] \, \varphi \mid X \mid \mu X.\varphi \mid \nu X.\varphi$$

The semantics of a formula φ on an LTS $M = (Q, A, T, q_0)$ denotes the set of states satisfying φ: boolean operators have the standard interpretation; possibility ($\langle a \rangle \, \varphi$) and necessity ($[a] \, \varphi$) operators denote the states from which some (resp. all) transitions labeled by a lead to states satisfying φ; minimal ($\mu X.\varphi$) and maximal ($\nu X.\varphi$) fixed point operators denote the least (resp. greatest) solution of the equation $X = \varphi$ interpreted over 2^Q. Fixed point operators act as binders for variables X in the same way as quantifiers in first-order logic. The alternation-free condition means that mutual recursion between minimal and maximal fixed point variables is forbidden.

Given an Lts M, the standard translation of an alternation-free formula φ into a Bes [6,1,15] proceeds as follows. First, extra propositional variables are introduced at appropriate places of φ to ensure that in every subformula $\sigma X.\varphi'$ (where $\sigma \in \{\mu, \nu\}$) of φ, φ' contains a single boolean or modal operator (this is needed in order to obtain only disjunctive or conjunctive formulas in the right-hand sides of the resulting Bes). Then, the Bes is constructed in a bottom-up manner, by creating an equation block for each closed fixed point subformula $\sigma X.\varphi'$ of φ. The alternation-free condition ensures that once the fixed point subformulas of $\sigma X.\varphi'$ have been translated into equation blocks, all remaining variables in $\sigma X.\varphi'$ are of sign σ. Each closed fixed point subformula $\sigma X.\varphi'$ is translated into an equation block $\{X_p \stackrel{\sigma}{=} (\varphi')_p\}_{p \in Q}$, where variables X_p express that state p satisfies X and the right-hand side boolean formulas $(\varphi')_p$ are obtained using the translation shown in the table below.

φ	$(\varphi)_p$	φ	$(\varphi)_p$
F	F	T	T
$\varphi_1 \vee \varphi_2$	$(\varphi_1)_p \vee (\varphi_2)_p$	$\varphi_1 \wedge \varphi_2$	$(\varphi_1)_p \wedge (\varphi_2)_p$
$\langle a \rangle\, \varphi_1$	$\bigvee_{p \xrightarrow{a} q} (\varphi_1)_q$	$[a]\, \varphi_1$	$\bigwedge_{p \xrightarrow{a} q} (\varphi_1)_q$
X	X_p	$\sigma X.\varphi_1$	X_p

This kind of Bes can be solved by the general algorithms A1 and A2 given in Section 3 (note that the translation procedure above allows to construct the Lts on-the-fly during Bes resolution). However, when the Lts M and/or the formula φ have a particular structure, the Bes can be simplified in order to make applicable the specialized algorithms A3 or A4.

Acyclic case. When M is acyclic and φ is guarded (i.e., every recursive call of a propositional variable in φ falls in the scope of a modal operator), the formula can be simplified in order to have only minimal fixed point operators, leading to an acyclic, single-block Bes [17]. This procedure can be also applied when φ has higher alternation depth and/or is unguarded, in the latter case φ being first translated to guarded form (with a worst-case quadratic blow-up in size).

If the above conditions are met, then the memory-efficient algorithm A3 can be used to perform μ-calculus model checking. One practical application consists in verifying μ-calculus formulas on sets of large execution traces (represented as acyclic Ltss M by merging their initial states) produced by intensive random execution of a system implementation [17].

Disjunctive/conjunctive case. When φ is a formula of Ctl [4], Actl (Action-based Ctl) [20] or Pdl [13], the Bes resulting after translation is in disjunctive or conjunctive form. The table below shows the translations of Ctl and Pdl operators into alternation-free μ-calculus [8] (here the '$-$' symbol stands for 'any action' of the Lts). For conciseness, we omitted the translations of Pdl box modalities $[\beta]\, \varphi$, which can be obtained by duality. Actl can be translated in a way similar to Ctl, provided action predicates (constructed from action names and boolean operators) are used inside diamond and box modalities instead of simple action names [9].

Operator		Translation
Ctl	$\mathsf{EX}\varphi$	$\langle - \rangle\, \varphi$
	$\mathsf{AX}\varphi$	$\langle - \rangle\, \mathsf{T} \wedge [-]\, \varphi$
	$\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$	$\mu X.\varphi_2 \vee (\varphi_1 \wedge \langle - \rangle\, X)$
	$\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$	$\mu X.\varphi_2 \vee (\varphi_1 \wedge \langle - \rangle\, \mathsf{T} \wedge [-]\, X)$
Pdl	$\langle \alpha \rangle\, \varphi$	$\langle \alpha \rangle\, \varphi$
	$\langle \varphi_1 ? \rangle\, \varphi_2$	$\varphi_1 \wedge \varphi_2$
	$\langle \beta_1 ; \beta_2 \rangle\, \varphi$	$\langle \beta_1 \rangle\, \langle \beta_2 \rangle\, \varphi$
	$\langle \beta_1 \cup \beta_2 \rangle\, \varphi$	$\langle \beta_1 \rangle\, \varphi \vee \langle \beta_2 \rangle\, \varphi$
	$\langle \beta^* \rangle\, \varphi$	$\mu X.\varphi \vee \langle \beta \rangle\, X$

The translation of Ctl formulas into Bess can be performed bottom-up, by creating a $\vee$-block (resp. a $\wedge$-block) for each subformula dominated by an operator $\mathsf{E}[\_\mathsf{U}\_]$ (resp. $\mathsf{A}[\_\mathsf{U}\_]$). For instance, the formula $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ is translated, via the μ-calculus formula $\mu X.\varphi_2 \vee (\varphi_1 \wedge \langle - \rangle\, X)$, first into the formula $\mu X.\varphi_2 \vee \mu Y.(\varphi_1 \wedge \mu Z.\, \langle - \rangle\, X)$ by adding extra variables Y and Z, and then into the equation block $\{X_p \overset{\mu}{=} (\varphi_2)_p \vee Y_p, Y_p \overset{\mu}{=} (\varphi_1)_p \wedge Z_p, Z_p \overset{\mu}{=} \bigvee_{p \to q} X_q\}_{p \in Q}$. This block is disjunctive, because its only $\wedge$-variables are Y_p and their left successors $(\varphi_1)_p$ correspond to Ctl subformulas encoded by some other block of the Bes. The formula $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ is translated, in a similar manner, into the equation block $\{X_p \overset{\mu}{=} (\varphi_2)_p \vee Y_p, Y_p \overset{\mu}{=} (\varphi_1)_p \wedge Z_p \wedge \bigwedge_{p \to q} X_q, Z_p \overset{\mu}{=} \bigvee_{p \to q} \mathsf{T}\}_{p \in Q}$. This block is conjunctive, because its $\vee$-variables X_p have their left successors $(\varphi_2)_p$ defined in some other block of the Bes, and its $\vee$-variables Z_p have all their successors constant.

Actl formulas can also be translated into disjunctive or conjunctive equation blocks, modulo their translations in μ-calculus [9]. In the same way, the translation of Pdl formulas into Bess creates a $\vee$-block (resp. a $\wedge$-block) for each subformula $\langle \beta \rangle\, \varphi$ (resp. $[\beta]\, \varphi$): normal boolean operators can be factorized such that at most one of their successors belongs to the current block, and the conjunctions (resp. disjunctions) produced by translating the test-modalities $\langle \varphi_1 ? \rangle\, \varphi_2$ (resp. $[\varphi_1 ?]\, \varphi_2$) have their left operands defined in other blocks of the Bes, resulting from the translation of the φ_1 subformulas.

Thus, the memory-efficient algorithm A4 can be used for model checking Ctl, Actl, and Pdl formulas. This covers most of the practical needs, since many interesting properties can be expressed using the operators of these logics.

5 Implementation and Experiments

We implemented the Bes resolution algorithms A1–A4 described in Section 3 in a generic software library, called Cæsar_Solve, which is built upon the primitives of the Open/Cæsar environment for on-the-fly exploration of Ltss [14]. Cæsar_Solve is used by the Bisimulator equivalence/preorder checker and the EVALUATOR model checker. We briefly describe the architecture of these tools and give some experimental results concerning the A1–A4 algorithms.

5.1 Architecture of the Solver Library

The CÆSAR_SOLVE library (see Figure 3) provides an Application Programming Interface (API) allowing to solve on-the-fly a variable of a BES. It takes as input the boolean graph associated to the BES together with the variable of interest, and produces as output the value of the variable, possibly accompanied by a diagnostic (portion of the boolean graph). Depending on its particular form, each block of the BES can be solved using one of the algorithms A1–A4, which were developed using the OPEN/CÆSAR primitives (hash tables, stacks, etc.).

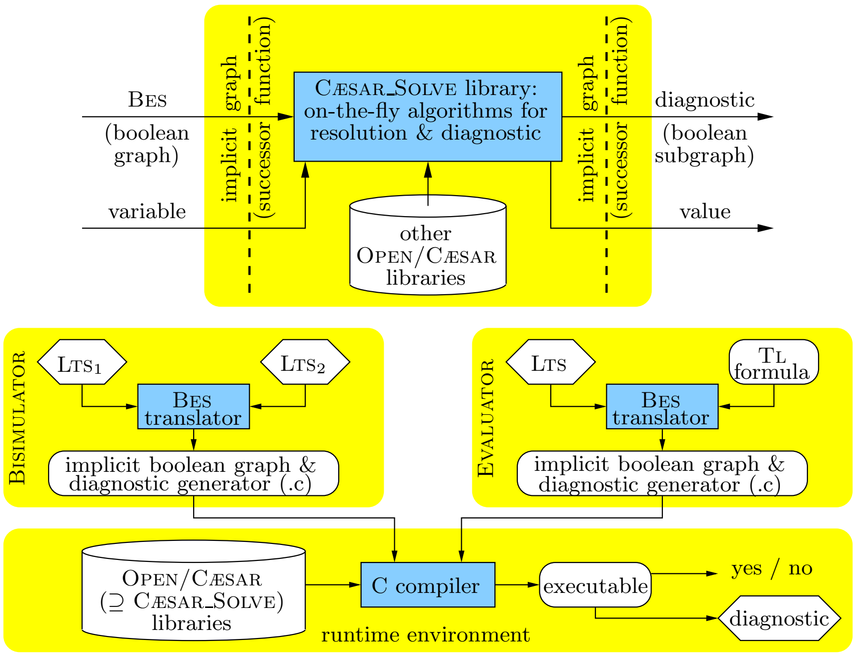
BES
(boolean graph)
variable
implicit graph
(successor function)
CÆSAR_SOLVE library: on-the-fly algorithms for resolution & diagnostic
other OPEN/CÆSAR libraries
diagnostic
(boolean subgraph)
value
BISIMULATOR
LTS1
LTS2
BES translator
implicit boolean graph & diagnostic generator (.c)
EVALUATOR
LTS
TL formula
BES translator
implicit boolean graph & diagnostic generator (.c)
OPEN/CÆSAR (⊇ CÆSAR_SOLVE) libraries
C compiler
executable
yes / no
diagnostic
runtime environment

Fig. 3. The CÆSAR_SOLVE library and the tools BISIMULATOR and EVALUATOR

Both the input boolean graph and the diagnostic are represented implicitly by their successor functions, which allow to iterate over the outgoing edges (dependencies) of a given vertex (variable) and hence to perform on-the-fly traversals of the boolean graphs. This scheme is similar to the implicit representation of LTSs defined by the OPEN/CÆSAR environment [14]. To use the library, a user must provide the successor function of the Bes (obtained by encoding some specific problem) and, if necessary, must interpret the resulting diagnostic by traversing the corresponding boolean subgraph using its successor function.

Two on-the-fly verification tools (see Figure 3) are currently using the Cæsar_Solve library: Bisimulator, an equivalence/preorder checker between two Ltss modulo the five relations mentioned in Section 4.1, and Evaluator, a model checker for regular alternation-free μ-calculus [18] over Ltss. Each tool translates its corresponding verification problem into a Bes resolution, identifying the particular cases suitable for algorithms A3–A4, and translates back the diagnostics produced by the library in terms of its input Lts(s).

5.2 Performance Measures

We performed several experiments to compare the performances of the resolution algorithms A1–A4. The applications selected were (several variants of) three communication protocols[1]: an alternating bit protocol (Abp), a bounded retransmission protocol (Brp), and a distributed leader election protocol (Dle).

The results are shown in the table below. The 1st series of experiments compares A1 with A2 as regards diagnostic depth; the 2nd and 3rd series compare A1 with A3, resp. A1 with A4 as regards memory consumption (measured in Kbytes). For each experiment, the table gives the measures obtained using A1 and A2–A4, and the corresponding difference ratios. Comparisons and inclusions between Ltss are performed using Bisimulator, and evaluations of temporal logic properties on Ltss are performed using Evaluator. All temporal properties are expressed using combinations of Actl and Pdl operators, which lead to disjunctive/conjunctive Bess, therefore enabling the use of algorithm A4.

The 1st experiments compare each protocol Lts modulo strong bisimulation with an erroneous Lts, and verify an invalid property on the protocol Lts. The 2nd experiments check that an execution sequence of 100000 transitions is included in each protocol Lts, and check a valid property on the sequence (both problems yield acyclic boolean graphs, hence enabling the use of algorithm A3). The 3rd experiments compare each protocol Lts modulo $\tau^* . a$ equivalence with its service Lts, which is deterministic (hence enabling the use of algorithm A4), and verify a valid property on the protocol Lts. We observe important reductions of diagnostic depth (up to 99%) whenever algorithm A2 can be used instead of A1, and reductions of memory consumption (up to 63%) whenever algorithms A3–A4 can be used instead of A1.

[1] All these examples can be found in the Cadp distribution, available at the Url `http://www.inrialpes.fr/vasy/cadp`.

A2 versus A1			Diagnostic depth					
App.	Size		Bisimulator			Evaluator		
	States	Trans.	A1	A2	%	A1	A2	%
Abp	935000	3001594	235	19	91.9	50	12	76.0
Brp	355091	471119	1455	31	97.8	744	18	97.5
Dle	143309	220176	2565	25	99.0	147	14	90.4

A3 versus A1			Memory consumption					
App.	Size		Bisimulator			Evaluator		
	States	Trans.	A1	A3	%	A1	A3	%
Abp	935000	3001594	37472	32152	14.1	10592	8224	22.3
Brp	355091	471119	17656	13664	22.6	10240	7432	27.4
Dle	28710	73501	15480	11504	25.6	8480	6248	26.3

A4 versus A1			Memory consumption					
App.	Size		Bisimulator			Evaluator		
	States	Trans.	A1	A4	%	A1	A4	%
Abp	935000	3001594	178744	152672	14.5	163800	60248	63.2
Brp	355091	471119	35592	23608	33.6	26752	17432	34.8
Dle	18281	44368	107592	94584	12.0	3904	3224	17.4

6 Conclusion and Future Work

We presented a generic library, called Cæsar_Solve, for on-the-fly resolution with diagnostic of alternation-free Bess. The library was developed using the Open/Cæsar environment [14] of the Cadp toolbox [10]. It implements an application-independent representation of Bess, precisely defined by an Api. The library currently offers four resolution algorithms A1–A4, A2 being optimized to produce small-depth diagnostics and A3, A4 being memory-efficient for acyclic and disjunctive/conjunctive Bess. Cæsar_Solve is used at the heart of the equivalence/preorder checker Bisimulator and the model checker Evaluator [18]. The experiments carried out using these tools assess the performance of the resolution algorithms and the usefulness of the diagnostic features.

We plan to continue our work along three directions. Firstly, in order to increase its flexibility, the Cæsar_Solve library can be enriched with other Bes resolution algorithms, such as Lmc [7] or the Gauss elimination-based algorithm proposed in [15]. Due to the well-defined Api of the library and the availability of the Open/Cæsar primitives, the prototyping of new algorithms is quite straightforward; from this point of view, Cæsar_Solve can be seen as an open platform for developing and experimenting Bes resolution algorithms. Another interesting way of research is the development of parallel versions of the algorithms A1–A4, in order to exploit the computing resources of massively parallel machines such as Pc clusters. Finally, other applications of the library can be envisaged, such as on-the-fly generation of test cases (obtained as diagnostics) from the Lts of a specification and the Lts of a test purpose, following the approach put forward in [11].

References

1. H. R. Andersen. Model checking and boolean graphs. *TCS*, 126(1):3–30, 1994.
2. J. C. M. Baeten and R. J. van Glabbeek. Another Look at Abstraction in Process Algebra. In *ICALP'87*, LNCS 267, pp. 84–94.
3. A. Bouajjani, J-C. Fernandez, S. Graf, C. Rodríguez, and J. Sifakis. Safety for Branching Time Semantics. In *ICALP'91*, LNCS 510.
4. 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, April 1986.
5. R. Cleaveland and B. Steffen. Computing behavioural relations, logically. In *ICALP'91*, LNCS 510, pp. 127–138.
6. R. Cleaveland and B. Steffen. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. In *CAV'91*, LNCS 575, pp. 48–58.
7. X. Du, S. A. Smolka, and R. Cleaveland. Local Model Checking and Protocol Analysis. *Springer STTT Journal*, 2(3):219–241, 1999.
8. E. A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *LICS'86*, pp. 267–278.
9. A. Fantechi, S. Gnesi, and G. Ristori. From ACTL to Mu-Calculus. In *ERCIM'92 Ws. on Theory and Practice in Verification (Pisa, Italy)*, IEI-CNR, pp. 3–10, 1992.
10. J-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In *CAV'96*, LNCS 1102, pp. 437–440.
11. J-C. Fernandez, C. Jard, Th. Jéron, L. Nedelka, and C. Viho. Using On-the-Fly Verification Techniques for the Generation of Test Suites. In *CAV'96*, LNCS 1102.
12. J-C. Fernandez and L. Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In *CAV'91*, LNCS 575.
13. M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *J. of Comp. and System Sciences*, (18):194–211, 1979.
14. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *TACAS'98*, LNCS 1384, pp. 68–84.
15. A. Mader. *Verification of Modal Properties Using Boolean Equation Systems.* VERSAL 8, Bertz Verlag, Berlin, 1997.
16. R. Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In *TACAS'00*, LNCS 1785, pp. 251–265.
17. R. Mateescu. Local Model-Checking of Modal Mu-Calculus on Acyclic Labeled Transition Systems. In *TACAS'02*, LNCS 2280, pp. 281–295.
18. R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Comp. Programming*, 2002. To appear.
19. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
20. R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems.* In *Semantics of Concurrency*, LNCS 469, pp. 407–419.
21. R. De Nicola, U. Montanari, and F. Vaandrager. Back and Forth Bisimulations. CS R9021, CWI, Amsterdam, May 1990.
22. D. Park. Concurrency and Automata on Infinite Sequences. In *Th. Comp. Sci.*, LNCS 104, pp. 167–183.
23. R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics. In *Proc. IFIP 11th World Computer Congress*, 1989.
24. B. Yang, R.E. Bryant, D. R. O'Hallaron, A. Biere, O. Condert, G. Janssen, R.K. Ranjan, and F. Somenzi. A Performance Study of BDD-Based Model-Checking. In *FMCAD'98*, LNCS 1522, pp. 255–289.