CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems

Radu Mateescu

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems

Radu Mateescu

N° 5948

Juillet 2006

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5948--FR+ENG

INRIA
RHÔNE-ALPES

CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems

Radu Mateescu*

Thème COM — Systèmes communicants
Projet VASY

Rapport de recherche n° 5948 — Juillet 2006 — 38 pages

Abstract: Boolean Equation Systems (BESs) provide a useful framework for modeling various verification problems on finite-state concurrent systems, such as equivalence checking and model checking. These problems can be solved on-the-fly (i.e., without constructing explicitly the state space of the system under analysis) by using a demand-driven construction and resolution of the corresponding BES. In this report, we present a generic software library dedicated to on-the-fly resolution of alternation-free BESs (i.e., without mutually recursive minimal and maximal fixed point equations). Four resolution algorithms are currently provided by the library: algorithms A1 and A2 are general, the latter being optimized to produce small-depth diagnostics, whereas algorithms A3 and A4 are specialized for handling acyclic and disjunctive/conjunctive BESs in a memory-efficient way. The library is developed within the CADP verification toolbox using the generic OPEN/CÆSAR environment and is currently used for three purposes: on-the-fly equivalence checking modulo five widely-used equivalence relations, on-the-fly model checking of regular alternation-free μ-calculus, and on-the-fly reduction of state spaces based on τ-confluence.

Key-words: bisimulation, boolean equation system, labeled transition system, model-checking, mu-calculus, partial order reduction, specification, temporal logic, verification

A short version of this report is also available as "CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems", *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2006.

* Radu.Mateescu@inria.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

CAESAR_SOLVE: une bibliothèque générique pour la résolution à la volée des systèmes d'équations booléennes sans alternance

Résumé : Les systèmes d'équations booléennes (SEBs) constituent un cadre utile pour modéliser différents problèmes de vérification sur des systèmes concurrents à nombre fini d'états, tels que la vérification par équivalences (*equivalence checking*) et par logiques temporelles (*model checking*). Ces problèmes peuvent être résolus à la volée (c.à.d. sans construire explicitement l'espace d'états du système à analyser) en effectuant une construction et une résolution incrémentale du SEB correspondant. Dans ce rapport, nous présentons une bibliothèque générique dédiée à la résolution à la volée des SEBs sans alternance (c.à.d. sans équations de plus petit et de plus grand point fixe mutuellement récursives). Quatre algorithmes de résolution sont couramment offerts par la bibliothèque : les algorithmes A1 et A2 sont généraux, A2 étant optimisé pour produire des diagnostics de profondeur réduite, alors que les algorithmes A3 et A4 sont spécialisés pour traiter les SEBs acycliques, respectivement disjonctifs/conjonctifs avec une consommation mémoire moindre. La bibliothèque a été développée au sein de la boîte à outils CADP au moyen de l'environnement générique OPEN/CÆSAR et elle est actuellement utilisée pour implémenter trois fonctionnalités de vérification á la volée : vérification par équivalence selon cinq relations largement utilisées, vérification par évaluation de formules du μ-calcul régulier sans alternance et réduction des espaces d'états basée sur la τ-confluence.

Mots-clés : bisimulation, logique temporelle, model checking, mu-calcul, réduction par ordre partiel, spécification, système d'équations booléennes, système de transitions étiquetées, vérification

1 Introduction

During the last decade, computer-assisted formal verification has become an essential step in the design process of complex safety-critical systems. Generally, formal verification consists in checking the conformance of a system w.r.t. its desired correctness requirements, both being described formally using appropriate languages with precise, mathematical semantics. A verification method increasingly used in industry due to its good cost-performance tradeoff is the so-called *explicit-state* verification, which enumerates and analyzes explicitly all reachable states (the *state space*) of the system under design. Compared to *symbolic* verification, which manipulates sets of, rather than individual states of the system by using specific encodings such as Binary Decision Diagrams (BDDs) [6], explicit-state verification exhibits better performances for systems with a high degree of asynchrony and nondeterminism [27]. Although limited to systems with finite state spaces, explicit-state verification provides a simple and efficient way of finding errors in complex systems, being particularly useful in the first phases of the design process, when errors are likely to occur more frequently.

Two different problems are traditionally distinguished in the framework of explicit-state verification: *equivalence checking* compares the system description with its external behaviour according to a certain equivalence relation, whereas *model checking* determines if the system description satisfies a list of temporal logic properties. In this article, we focus on a particular solution to these problems, called *on-the-fly* verification, which combats state explosion (prohibitive size of the state space for systems with many parallel processes and complex data structures) by constructing the state space incrementally during verification. More precisely, we aim at facilitating the construction of robust and efficient tools for on-the-fly verification, which — as practical experience has shown — is a difficult and costly task. One way towards this objective is to provide reusable software components (e.g., libraries) having a theoretical foundation that is sufficiently general to be applicable in several contexts.

Boolean Equation Systems (BESs) [33] are an appropriate framework for explicit-state verification, by allowing to represent equivalence checking and model checking problems in terms of BES resolution. Numerous algorithms for solving BESs have been proposed in the literature (see [33, chap. 6] for a survey). Among these, the on-the-fly (also called *local*) resolution algorithms provide a natural way to perform on-the-fly verification, since they compute the value of a boolean variable by constructing the BES (and hence, the state space of the system under analysis) in a demand-driven way. Another useful feature of BES resolution algorithms is the ability to generate *diagnostics* (portions of the BES explaining the truth value of a variable), which offer considerable help for debugging applications and for understanding properties expressed in temporal logic [34].

However, as opposed to the situation in the field of symbolic verification, for which a significant number of BDD-based packages are available (see [52] for a survey), very few attempts were made for constructing generic environments for BES resolution dedicated to on-the-fly verification. In this article we present CÆSAR_SOLVE, a generic library for BES resolution and diagnostic generation, created using the OPEN/CÆSAR environment for on-the-fly verification [21]. The CÆSAR_SOLVE library 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). This allows to clearly separate the aspects related to the representation of the verification problems in terms of BESs and those related to on-the-fly resolution of BESs, which can be addressed independently.

Four resolution algorithms are currently available in the CÆSAR_SOLVE library. Algorithms A1 and A2 are general (i.e., 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 the memory-efficient resolution of acyclic BESs and disjunctive/conjunctive BESs, which occur frequently in practice. At the present time, CÆSAR_SOLVE serves as engine for three on-the-fly verification tools developed within the CADP toolbox [22]: the equivalence/preorder checker BISIMULATOR, which implements five widely-used equivalence relations, the model checker EVALUATOR for regular alternation-free μ-calculus [36], and the tool REDUCTOR, which implements various reductions on LTSs, among which τ-confluence [25, 40].

The remainder of the article is organized as follows. Section 2 defines alternation-free BESs and states the principles underlying on-the-fly resolution and diagnostic generation. Section 3 presents in detail the resolution algorithms A1–A4 and compares them according to three requirements which aim at characterizing both worst-case and average-case time complexity. Section 4 shows the encodings of several equivalence relations, temporal logics, and of τ-confluence detection in terms of alternation-free BESs, also identifying the particular cases suitable for the optimized algorithms A3 and A4. Section 5 describes the architecture of the CÆSAR_SOLVE library and of the three verification tools BISIMULATOR, EVALUATOR, and REDUCTOR that are built upon it, together with experimental data comparing the performance of algorithms A1–A4. Finally, Section 6 summarizes the results and indicates directions for future work.

Related work

An extensive amount of literature has been dedicated to the study of BESs. The first algorithms with polynomial complexity for solving alternation-free BESs (i.e., without mutually recursive minimal and maximal fixed point variables) were proposed by Arnold and Crubillé [3], by Cleaveland and Steffen [12], and by Vergauwen and Lewi [49]. These algorithms were *global*, i.e., they required the BES to be completely constructed before starting the resolution, and computed the solution for *all* the variables of the BES; the algorithms given in [12, 49] have a linear time complexity in the size of the BES (number of variables and operators).

Since most of the time one is interested by solving a single variable of the BES (e.g., for equivalence checking and model checking), research has been undertaken for devising on-the-fly (or *local*) algorithms, which are able to compute the value of a single variable by examining only the BES portion influencing that variable. The first on-the-fly algorithms for solving alternation-free BESs were proposed by Larsen [30], by Andersen [1], and by Vergauwen and Lewi [50]; the algorithms given in [1, 50] have a linear time complexity in the BES size. Another on-the-fly resolution algorithm with linear time complexity for alternation-free BESs is the on-the-fly algorithm for checking non-emptiness of 1-letter alternating weak automata

proposed in [29]. This was the first automata-theoretic based algorithm yielding a model checking procedure with linear time complexity for branching temporal logics such as CTL and the alternation-free μ-calculus.

Other algorithms for solving BESs of alternation depth two (i.e., which allow pairs of minimal and maximal fixed point variables to depend upon each other) were proposed by Vergauwen and Lewi [50] (local) and by Andersen and Vergauwen [2] (global). The first local algorithm that handled BESs of arbitrary alternation depth (i.e., which contain an arbitrary degree of mutual recursion between minimal and maximal fixed point variables) was proposed by Liu, Ramakrishnan, and Smolka [31]. Another efficient local algorithm, called LMC, was proposed by Du, Smolka, and Cleaveland [15]: the algorithm speeds up the BES resolution by detecting the strongly connected components of the dependency graph between variables and was successfully used for analysing industrial-sized communication protocols.

Recently, Groote and Keinänen [24] identified the class of disjunctive/conjunctive BESs, which may be of arbitrary alternation depth, but in which any two mutually dependent variables are either disjunctive, or conjunctive (the boolean formulas in the right-hand sides of the corresponding equations contain either disjunctions, or conjunctions). For this class of BESs, which is shown to be useful in most practical situations, they devised resolution algorithms with quadratic time complexity (for BESs with alternation depth greater than two) and linear time complexity (for alternation-free BESs).

The theory underlying BESs and their relationship with modal logics and μ-calculi were thoroughly studied by Mader [33], who also proposed several efficient local and global resolution algorithms based upon Gauss elimination methods. An alternative formulation of equivalence checking and model checking in terms of *game graphs* was proposed by Stevens and Stirling [45]. Game graphs, which generalize the *boolean graphs* introduced by Andersen [1] for representing dependencies between BES variables, provide a more intuitive way of developing verification algorithms. However, the monolithic structure of most local resolution algorithms (such as those based on game graphs) makes them difficult to optimize for handling BESs with multiple equation blocks.

Despite the substantial amount of theoretical work in the area of BES resolution, there are surprisingly few implementations and experimental comparisons available. One such implementation is the Fixpoint Analysis Machine (FAM) [42], which provides an efficient abstract resolution engine also capable of solving BESs. Within the METAFRAME [43] open tool coordination environment, FAM was used to perform various tasks, such as model checking of μ-calculus formulas and data flow analysis.

Our objective in developing the CÆSAR_SOLVE library was to provide a generic basis for developing BES resolution algorithms, experimenting and comparing them uniformly, and using them to build verification tools in a modular way.

2 Alternation-free boolean equation systems

A Boolean Equation System (BES) [1, 33] 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 assumed to be sorted topologically such that a block M_i only depends upon blocks M_k with $k > i$.

This definition of alternation-freedom (which is similar to the definition used in the equational version of the alternation-free μ-calculus used in [13]) can be shown to be equivalent to the original definition of alternation-free μ-calculus given in [16]. In alternation-free μ-calculus (see Section 4.2), a minimal fixed point variable X defined by a formula $\mu X.\varphi$ cannot occur free in a maximal fixed point subformula $\nu Y.\varphi'$ of φ, since this would cause a mutual dependency between variables X and Y, which have opposite fixed point signs. After translating the formula $\mu X.\varphi$ into a BES, the boolean variables corresponding to X (defined in a minimal fixed point block M_X) will depend upon those corresponding to Y (defined in a maximal fixed point block M_Y); therefore, forbidding the free occurrences of X inside $\nu Y.\varphi'$ amounts to forbid the dependencies from block M_Y to block M_X, i.e., the cycles between blocks.

The semantics of a formula $op_i\{X_1, ..., X_k\}$ w.r.t. $\mathbf{Bool} = \{\mathsf{F}, \mathsf{T}\}$ and a context $\delta : \mathcal{X} \rightarrow \mathbf{Bool}$, which must initialize all variables X_1, ..., X_k, is the boolean value defined as follows:

$$[\![op_i\{X_1, ..., X_k\}]\!]\delta = \delta(X_1)\ op_i\ ...\ op_i\ \delta(X_k)$$

The semantics of an equation block $M_i = \{X_j \stackrel{\sigma_i}{=} op_j \boldsymbol{X}_j\}_{j \in [1, m_i]}$ w.r.t. a context δ is the σ_i-fixed point of a vectorial functional $\Phi_{i\delta} : \mathbf{Bool}^{m_i} \rightarrow \mathbf{Bool}^{m_i}$:

$$[\![\{X_j \stackrel{\sigma_i}{=} op_j \boldsymbol{X}_j\}_{j \in [1, m_i]}]\!]\delta = \sigma_i \Phi_{i\delta}$$

where

$$\Phi_{i\delta}(b_1, ..., b_{m_i}) = ([\![op_j \boldsymbol{X}_j]\!](\delta \oslash [b_1/X_1, ..., b_{m_i}/X_{m_i}]))_{j \in [1, m_i]}$$

The notation $\delta \oslash [b_1/X_1, ..., b_n/X_n]$ stands for 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:

$$\begin{array}{rcl} \delta_n & = & [\![M_n]\!][] \\ \delta_i & = & ([\![M_i]\!]\delta_{i+1}) \oslash \delta_{i+1} \text{ for } i \in [1, n-1] \end{array}$$

Note that the context for interpreting M_n is empty (since M_n is closed) and a block M_i is interpreted in the context of all blocks M_k with $k > i$.

A block M_i is *acyclic* if there are no cyclic dependencies between the variables defined in M_i. 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, all 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 [1, 50, 33, 15]. 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 the variables defined in M_i. When a variable X_j defined in 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 defined in M_i may be required several times during the resolution process, the computation results must be kept persistent between subsequent calls of R_i in order to obtain an efficient overall resolution.

Compared to other on-the-fly resolution algorithms like LMC [15], which consists of a single monolithic routine handling the whole BES, we believe the scheme above presents two advantages:

- The algorithms used by the resolution routines associated to individual blocks are simpler, since they must handle a single type of fixed point equations (either minimal, or maximal).
- 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

In this section we present 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 our resolution algorithms in terms of *boolean graphs* [1], which provide a graphical, more intuitive representation of BESs. Given an equation block $M_i = \{X_j \stackrel{\mu}{=} op_j \boldsymbol{X}_j\}_{j \in [1,m_i]}$, 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), $E = \{x_j \rightarrow x_k \mid j \in [1, m_i] \wedge X_k \in \boldsymbol{X}_j\}$ is the set of *edges* (dependencies between variables), and $L : V \rightarrow \{\vee, \wedge\}$, $L(x_j) = op_j$ 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

$$\begin{cases} X_1 &= X_2 \vee X_3 \\ X_2 &= X_4 \wedge X_5 \\ X_3 &= X_6 \wedge X_7 \\ X_4 &= X_8 \vee X_9 \\ X_5 &= X_9 \vee X_{10} \vee X_6 \\ X_6 &= \mathsf{T} \\ X_7 &= \mathsf{T} \\ X_8 &= X_2 \wedge X_{10} \\ X_9 &= \mathsf{T} \\ X_{10} &= \mathsf{F} \end{cases}$$

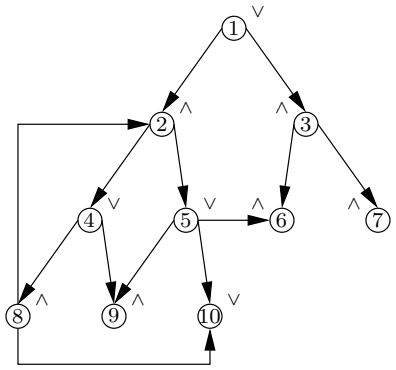


Figure 1: A BES and its corresponding boolean graph

and occurring free in M_i can be seen as constants, because their values are computed on-the-fly by calls to R_k. Figure 1 shows a single μ-block BES together with its corresponding boolean graph.

As expected, the boolean graphs associated to acyclic blocks are directed acyclic graphs (DAGS). 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).

In the sequel, we consider that boolean graphs $G = (V, E, L)$ are represented *implicitly* by their successor function $E : V \rightarrow 2^V$, which allows them to be constructed on-the-fly by a forward exploration.

The resolution algorithms we present are all based upon the same principle: starting at the vertex (variable) of interest, they perform an on-the-fly, forward exploration of the boolean graph and propagate backwards the values of the "stable" vertices (i.e., whose final value has been determined). When a vertex y becomes stable and its value T (resp. F) is propagated backwards to a $\vee$-vertex (resp. $\wedge$-vertex) x, it stabilizes it *actively* to the same value; in terms of the original BES, this amounts to substitute y by its value in the right-hand side formula of the equation defining x. The algorithms terminate either when the vertex of interest becomes stable after the backwards propagation of a value, or when the entire boolean graph is explored; in the latter case, the vertex of interest is stabilized *passively*.

During backwards propagation, some additional information is recorded for the purpose of diagnostic generation: for each vertex x which is stabilized actively by one of its successors y (e.g., when x is a $\vee$-vertex and y was stabilized to T), that successor is stored in a field $s(x)$. After the resolution terminates, a diagnostic (boolean subgraph rooted at the vertex of interest) can be obtained by performing a new exploration of the boolean graph starting at the vertex of interest and following, for each vertex x encountered, either its successor $s(x)$ (if x was stabilized actively), or all its successors (if x was stabilized passively) [34]. A diagnostic is called *example* (resp. *counterexample*) iff the vertex of interest was evaluated to T (resp. F). The *depth* of a diagnostic for a vertex x is the maximal distance from x to

all the other vertices occurring in the diagnostic, where the distance between two vertices is the length of the shortest path connecting them.

To have a basis of comparison for the different resolution algorithms, we propose below three requirements desirable to obtain a good time complexity.

R1: The resolution of a vertex of the boolean graph must be carried out in a time linear w.r.t. the size of the graph, i.e., $O(|V|+|E|)$. This is necessary for obtaining an overall linear time complexity for the resolution of an alternation-free BES containing multiple blocks.

R2: During the resolution of a vertex, every new vertex explored must be related to the vertex of interest by (at least) a path of unstable vertices in the boolean graph. This aims at reducing the average time complexity by limiting the graph exploration only to vertices "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 vertex lead to multiple explorations of the graph (which may destroy the overall linear time complexity).

3.1 Algorithm A1 (DFS, general)

Algorithm A1 (see Figure 2) is based upon a depth-first search (DFS) of the boolean graph, performed iteratively starting at the vertex of interest x. Visited vertices are stored in a set $A \subseteq V$ and visited but unexplored vertices are stored in a stack. To each vertex y are associated three informations: a counter $c(y)$, 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); a counter $p(y)$ indicating the current successor of y that must be explored (the successors in $E(y)$ are assumed to be indexed from 0 to $|E(y)|-1$); and a set $d(y)$ containing the vertices that currently depend upon y. At each iteration of the main while-loop (lines 12–57), the vertex y placed on top of the DFS stack is explored. If y is already stable (i.e., $c(y) = 0$), its value is back-propagated by the inner while-loop (lines 17–32) along the dependencies d; during the back-propagation, each time a $\vee$-vertex w is stabilized actively by one of its successors u, that successor is stored in $s(w)$ for the purpose of diagnostic generation (lines 23–26). If y is unstable, its next unexplored successor $(E(y))_{p(y)}$ is visited and, if it is stable or new, is pushed on top of the DFS stack (lines 37–53). Finally, if y is unstable and all its successors have been explored, it is popped from the DFS stack (lines 54–56). After termination of the DFS while-loop, the value computed for x is returned, which is T iff the associated counter $c(x)$ is 0 (line 58).

Figure 3 illustrates the result of executing algorithm A1 in order to evaluate the variable X_1 of the BES previously shown in Figure 1. Variables whose final value is T (resp. F) are denoted by grey (resp. white) vertices. The white box delimits the portion of the boolean graph explored during the DFS traversal performed by A1 (during the traversal, the successors of a vertex are visited in the order given by scanning the right-hand side of the corresponding equation from left to right). The thick arrows indicate the successors $s(y)$

```
 1. function A1 (x, (V, E, L)) : Bool is
 2.   var A, B : 2^V; d : V → 2^V; stack : V*;
 3.       c, p : V → Nat; u, w, y, z : V;
 4.   if L(x) = ∧ then
 5.     c(x) := |E(x)|
 6.   else
 7.     c(x) := 1
 8.   endif;
 9.   p(x) := 0; d(x) := ∅;
10.   A := {x};
11.   stack := push(x, nil);
12.   while stack ≠ nil do
13.     y := top(stack);
14.     if c(y) = 0 then
15.       if d(y) ≠ ∅ then
16.         B := {y};
17.         while B ≠ ∅ do
18.           let u ∈ B;
19.           B := B \ {u};
20.           forall w ∈ d(u) do
21.             if c(w) > 0 then
22.               c(w) := c(w) − 1;
23.               if c(w) = 0 then
24.                 if L(w) = ∨ then
25.                   s(w) := u
26.                 endif;
27.                 B := B ∪ {w}
28.               endif
29.             endif
30.           end;
31.           d(u) := ∅
32.         end
33.       else
34.         stack := pop(stack)
35.       endif
36.     elsif p(y) < |E(y)| then
37.       z := (E(y))_{p(y)};
38.       p(y) := p(y) + 1;
39.       if z ∈ A then
40.         d(z) := d(z) ∪ {y};
41.         if c(z) = 0 then
42.           stack := push(z, stack)
43.         endif
44.       else
45.         if L(z) = ∧ then
46.           c(z) := |E(z)|
47.         else
48.           c(z) := 1
49.         endif;
50.         p(z) := 0; d(z) := {y};
51.         A := A ∪ {z};
52.         stack := push(z, stack)
53.       endif
54.     else
55.       stack := pop(stack)
56.     endif
57.   end;
58.   return c(x) = 0
59. end
```

Figure 2: Algorithm A1: DFS-based local resolution of a μ-block

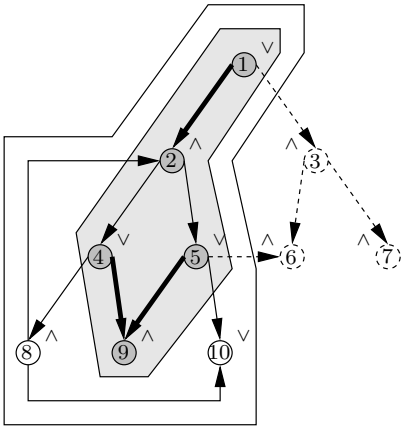


Figure 3: Result of executing A1

computed for those vertices y that were stabilized actively during back-propagation of T values. The grey box delimits the diagnostic (example for X_1) computed after termination of A1; all $\vee$-vertices (resp. $\wedge$-vertices) y contained in the example have only their successor $s(y)$ (resp. all their successors) contained in the example. The diagnostic produced is a DAG of depth 3.

We discuss below the behaviour of algorithm A1 w.r.t. the three efficiency requirements R1–R3. The algorithm satisfies requirement R1, because every edge in the boolean subgraph explored is traversed at most twice: forwards, when its target vertex is pushed on the DFS stack, and backwards, when the value of its target vertex (if it became stable) is back-propagated. Therefore, each call to A1 has a worst-case time complexity $O(|V| + |E|)$. The same bound applies for memory consumption, since in the worst-case every state and edge of the boolean graph will be stored (edges are stored as backward dependencies). A1 also satisfies requirement R2, because each new vertex visited from the top of the DFS stack is related to the vertex of interest (which is at the bottom of the DFS stack) via the unstable vertices present on the stack. Finally, it satisfies requirement R3, since the boolean subgraph explored by each call to the algorithm contains only stable vertices (i.e., which depend only upon vertices in A).

Algorithm A1 can be seen as an improved version of the local resolution algorithm proposed in [1]: it is simpler to understand, being implemented iteratively by using a while-loop and an explicit DFS stack instead of two mutually recursive functions; it has a better average complexity, since the values of vertices are back-propagated as soon as they become stable; and it exhibits a lower memory consumption, because dependencies between variables are discarded during back-propagation (line 31). The counter-based technique which allows to efficiently detect the stabilization of a vertex y when $c(y) = 0$ was initially proposed in [3, 13]. A1 was initially developed for model checking regular alternation-free μ-calculus [36].

3.2 Algorithm A2 (BFS, general)

Algorithm A2 (see Figure 4) is based upon a breadth-first search (Bfs) of the boolean graph, performed iteratively starting at the vertex 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 $c(y)$, 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 12–52), the vertex y placed in front of the Bfs queue is explored. If y is already stable (i.e., $c(y) = 0$), its value is back-propagated by the inner while-loop (lines 17–32) along the dependencies d; during the back-propagation, each time a $\vee$-vertex w is stabilized actively by one of its successors u, that successor is stored in $s(w)$ for the purpose of diagnostic generation (lines 23–26). If y is unstable, all successors $E(y)$ are visited and, if they are stable or new, are inserted at the end of the Bfs queue (lines 34–50). After termination of the Bfs while-loop, the value computed for x is returned, which is T iff the associated counter $c(x)$ is 0 (line 53).

Figure 5(b) illustrates the result of executing algorithm A2 in order to evaluate the variable X_1 of the Bes previously shown in Figure 1. The diagnostic produced is a Dag of depth 2, smaller than the diagnostic produced by A1, shown in Figure 3.

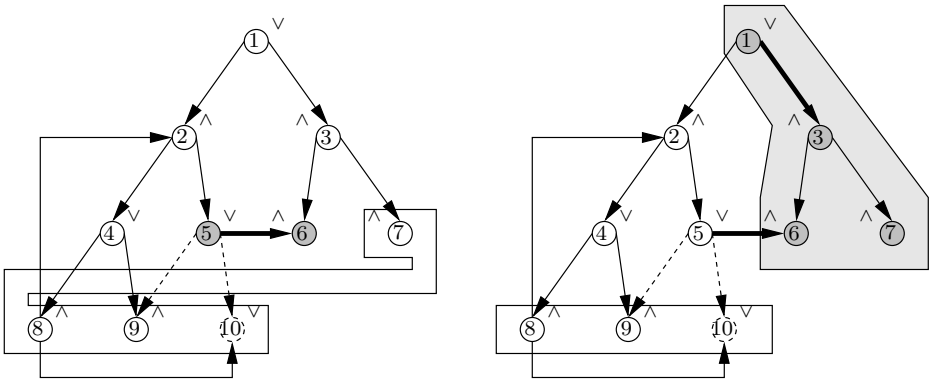
We discuss below the behaviour of algorithm A2 w.r.t. the three efficiency requirements R1–R3. Like A1, algorithm A2 satisfies requirement R1, because at each execution it performs a single Bfs traversal (possibly interleaved with an overall backward traversal of the boolean subgraph explored) having a worst-case time complexity $O(|V| + |E|)$. The same bound applies for memory consumption, since in the worst-case every state and edge of the boolean graph will be stored. Edges are kept as backward dependencies, which are discarded during back-propagation (line 31), but some of them (whose source vertices are stabilized passively) may remain present after termination of a call to A2.

However, unlike A1, algorithm A2 does not satisfy requirement R2, because the back-propagation may stabilize vertices that "cut" all the paths relating the vertex of interest x to (some) vertices currently present on the Bfs queue, and thus at some points the algorithm may explore vertices which are useless for deciding the truth value of x. This is illustrated in Figure 5(a): after vertex X_6 was back-propagated and has actively stabilized its predecessor X_5, the remaining dependencies $X_5 \rightarrow X_9$ and $X_5 \rightarrow X_{10}$ become useless, since they cannot influence anymore the value of X_5. At this point, vertex X_{10}, which is present on the Bfs queue, is unreachable from the vertex of interest X_1, and therefore its further exploration could safely be avoided (for the boolean graph shown in the figure, the exploration of X_{10} will not cost much since it is a $\vee$-sink vertex, but this may not be the case in general). Nevertheless, the extra effort spent to explore useless vertices when a call to A2 is executed may pay off if the values of these vertices are required during later calls of the algorithm. This observation is confirmed experimentally: in case of multiple calls, A2 is only about 15% slower than A1.

It is possible to devise a version of algorithm A2 compliant with requirement R2, but this would need an efficient solution for the Dynamic Graph Connectivity Problem (Dgcp), which consists in determining if there exists a path connecting two vertices of a graph in pres-

```
 1. function A2 (x, (V, E, L)) : Bool is
 2.   var A, B : 2^V; d : V → 2^V; queue : V*;
 3.       c : V → Nat; u, w, y, z : V;
 4.   if L(x) = ∧ then
 5.     c(x) := |E(x)|
 6.   else
 7.     c(x) := 1
 8.   endif;
 9.   d(x) := ∅;
10.   A := {x};
11.   queue := put(x, nil);
12.   while queue ≠ nil [∧ c(x) ≠ 0] do
13.     y := head(queue);
14.     queue := tail(queue);
15.     if c(y) = 0 then
16.       B := {y};
17.       while B ≠ ∅ do
18.         let u ∈ B;
19.         B := B \ {u};
20.         forall w ∈ d(u) do
21.           if c(w) > 0 then
22.             c(w) := c(w) − 1;
23.             if c(w) = 0 then
24.               if L(w) = ∨ then
25.                 s(w) := u
26.               endif;
27.               B := B ∪ {w}
28.             endif
29.           endif
30.         end;
31.         d(u) := ∅
32.       end
33.     else
34.       forall z ∈ E(y) do
35.         if z ∈ A then
36.           d(z) := d(z) ∪ {y};
37.           if c(z) = 0 then
38.             queue := put(z, queue)
39.           endif
40.         else
41.           if L(z) = ∧ then
42.             c(z) := |E(z)|
43.           else
44.             c(z) := 1
45.           endif;
46.           d(z) := {y};
47.           A := A ∪ {z};
48.           queue := put(z, queue)
49.         endif
50.       end
51.     endif
52.   end;
53.   return c(x) = 0
54. end
```

Figure 4: Algorithm A2: BFS-based local resolution of a μ-block



(a) intermediate result (b) final result

Figure 5: Result of executing A2

ence of edge additions (new edges explored during the Bfs traversal) and deletions (edges that become useless when their source vertex is stabilized). The most efficient algorithm for solving Dgcp that we are aware of has a time complexity $O(\log n / \log\log\log n)$ for handling connectivity queries in a dynamic graph with n vertices, and a complexity $O(\log n (\log\log n)^3)$ for handling edge additions/deletions [47]; enhancing A2 with this Dgcp algorithm would probably make the overall time complexity uncompatible with requirement R1. An alternative way to avoid the exploration of useless vertices would be to periodically perform additional "cleaning" traversals of the boolean subgraph already explored by A2 in order to identify the vertices in the Bfs queue which are reachable from the vertex of interest [51]. The difficulty with this solution relies on choosing the appropriate cleaning period, which has to be sufficiently small to limit the amount of useless vertices explored between two calls of the cleaning algorithm, and sufficiently large to prevent the overall complexity of becoming quadratic (the latter condition is impossible to achieve in general, since the number of vertices in the boolean graph is unknown in advance). Further experimentation is needed to assess which of these solutions (if any) would improve the speed of A2 effectively.

Finally, algorithm A2 satisfies requirement R3, since at the end of the main while-loop all visited vertices are stable (they depend only upon the vertices in A). In practice, there are certain situations in which the compliance with requirement R3 is not mandatory, namely when the algorithm A2 is invoked only once: this occurs in particular for the single-block Bess obtained by encoding equivalence checking problems (see Section 4.1). In such cases, the algorithm A2 may terminate as soon as the vertex of interest is stabilized, because the values of the currently unstable vertices (e.g., those present on the Bfs queue) will not be required anymore; this is implemented by adding a test in conjunction to the condition of the main while-loop (line 12). The result of executing A2 shown in Figure 5(b) illustrates in fact such an early termination: the Bfs queue still contains vertices $\{X_8, X_9, X_{10}\}$ when vertex X_1 is stabilized. Notice that this kind of early termination is automatically achieved by algorithm A1, which is based upon a Dfs traversal of the boolean graph: when the vertex of interest becomes stable, the Dfs stack is necessarily empty and the main while-loop of A1 terminates.

The essential advantage of algorithm A2 when compared to A1 relies on its ability to generate small-depth diagnostics, which is a direct consequence of using a BFS traversal instead of a DFS one. In practice, the diagnostics produced by A2 have a depth significantly smaller than those produced by A1 (up to two orders of magnitude), which makes them much more useful for the end-user. The diagnostic produced by A2 shown in Figure 5(b) has the minimal depth among the different diagnostics possible for variable X_1; however, A2 does not guarantee this in general. This is due to the fact that, to increase resolution speed, stable vertices are back-propagated as soon as they are encountered in front of the BFS queue, whereas some other vertices present on the queue would yield smaller-depth diagnostics if they were back-propagated first. If the diagnostics are trees (e.g., the example for X_1 shown in Figure 5(b)), the problem of finding a minimal-depth diagnostic can be solved in linear time. Although it seems possible to devise a version of A2 that produces minimal-depth tree diagnostics, we preferred to develop a separate algorithm which can be used in conjunction with any resolution algorithm (such as A1–A4) to compute minimal-depth tree diagnostics, once a diagnostic of depth k has been produced (see Section 3.5).

3.3 Algorithm A3 (DFS, acyclic)

Algorithm A3 (see Figure 6) is specialized for solving acyclic boolean graphs. Like A1, it is based upon a DFS traversal of the boolean graph, starting at the vertex of interest x; for the sake of conciseness, we describe A3 by using a recursive function instead of an iterative one[1]. Visited vertices are stored in a global set $A \subseteq V$, initially empty. To each vertex y are associated two informations: a counter $p(y)$ indicating the current successor of y that must be explored (the successors in $E(y)$ are assumed to be indexed from 0 to $|E(y)| - 1$); and a boolean $v(y)$ denoting the current value of y. The value of vertex x is initialized to F (resp. T) if x is a $\vee$-vertex (resp. $\wedge$-vertex) (lines 6–10). Then, for each successor y of x that has not been visited yet, its value *val* is computed by a recursive call to A3 (lines 15–17). If *val* is different from the current value $v(x)$ (e.g., if x is a $\vee$-vertex and $val = \mathsf{T}$), vertex x is stabilized to *val* because its value cannot be influenced anymore by its remaining unexplored successors; in this case, the successor $s(x)$ is set to y for the purpose of diagnostic generation, and the while-loop is exited (lines 18–21). After termination of the while-loop, the value $v(x)$ is returned as result (line 26), since at this moment the vertex x is stable. This property is due to the absence of cycles in the boolean graph, which ensures that once a vertex x has been explored by the DFS traversal, all vertices reachable from x have also been explored.

Figure 7 illustrates the result of executing algorithm A3 in order to evaluate the variable X_1 of an acyclic block. The white box delimits the portion of the boolean graph explored during the DFS traversal performed by A3, and the grey box delimits the diagnostic produced (example for X_1), which is a DAG of depth 3. The edges $X_4 \rightarrow X_8$ and $X_4 \rightarrow X_5$ were not traversed during the DFS, because X_4 was stabilized to F after its successor X_7 was evaluated, causing an early termination of the while-loop of A3.

[1]In practice, to avoid possible overflows of the system call stack, A3 is implemented iteratively by using a while-loop and an explicit DFS stack allocated on the heap.

```
 1.  var A : 2^V;
 2.  A := ∅;
 3.  function A3 (x, (V, E, L)) : Bool is
 4.    var v : V → Bool; p : V → Nat;
 5.        y : V; val : Bool;
 6.    if L(x) = ∨ then
 7.      v(x) := F
 8.    else
 9.      v(x) := T
10.    endif;
11.    p(x) := 0;
12.    A := A ∪ {x};
13.    while p(x) < |E(x)| do
14.      y := (E(x))_{p(x)};
15.      if y ∉ A then
16.        val := A3 (y, (V, E, L))
17.      endif;
18.      if val ≠ v(x) then
19.        v(x) := val;
20.        s(x) := y;
21.        p(x) := |E(x)|
22.      else
23.        p(x) := p(x) + 1
24.      endif
25.    end;
26.    return v(x)
27.  end
```

Figure 6: Algorithm A3: DFS-based local resolution of an acyclic μ-block

$$\begin{cases} X_1 &= X_2 \wedge X_3 \\ X_2 &= X_4 \vee X_5 \\ X_3 &= X_5 \vee X_6 \\ X_4 &= X_7 \wedge X_8 \wedge X_5 \\ X_5 &= X_8 \\ X_6 &= X_8 \wedge X_9 \\ X_7 &= \mathsf{F} \\ X_8 &= \mathsf{T} \\ X_9 &= \mathsf{T} \end{cases}$$

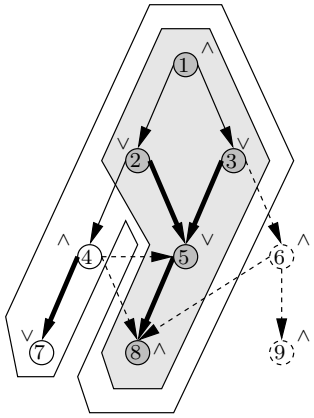


Figure 7: Result of executing A3

We discuss below the behaviour of algorithm A3 w.r.t. the three efficiency requirements R1–R3. The algorithm clearly satisfies requirement R1, because at each execution it performs a single DFS traversal having a worst-case time complexity $O(|V| + |E|)$. It also satisfies requirement R2, since every new vertex y explored by a recursive call to A3 is directly related to its predecessor x, which is unstable at that moment, and thus y is inductively related, via unstable vertices, to the vertex of interest, currently solved by the topmost call to A3. Finally, the algorithm satisfies requirement R3, because upon termination of a call to A3, all explored vertices have their final value computed (they have been stabilized actively).

The main advantage of A3 w.r.t. the general algorithms A1 and A2 (which can handle acyclic boolean graphs as well) relies on its reduced memory consumption, achieved by exploiting the absence of cycles in the boolean graph. Indeed, since back-propagation of values is performed directly between a vertex x and one of its successors y (in an iterative version of A3, back-propagation would take place only between vertices present on the DFS stack), there is no need to keep track of backward dependencies. Therefore, A3 avoids storing the edges of the boolean graph, having a worst-case memory complexity $O(|V|)$ instead of $O(|V| + |E|)$ as the general algorithms A1 and A2 would exhibit when executed on acyclic boolean graphs. Notice that the DFS traversal is crucial for ensuring this property: an algorithm based upon a BFS traversal, even when executed on an acyclic boolean graph, would not be able to perform back-propagation of values without storing backward dependencies.

Unlike A1 and A2, algorithm A3 has a symmetric structure: in its current form it can handle both μ-blocks and ν-blocks uniformly, and it performs simultaneous back-propagation of T and F values (this explains why the counters $c(y)$ used by A1 and A2 to keep track of the stable successors of y are not needed for A3). A3 was initially developed for model checking μ-calculus formulas on large event traces obtained by intensive simulation of a system [35].

3.4 Algorithm A4 (DFS, disjunctive/conjunctive)

Algorithm A4 (see Figure 8) is based upon a DFS traversal of the boolean graph, performed recursively starting at the vertex of interest x. A4 is specialized for solving boolean graphs corresponding to disjunctive or conjunctive blocks; we show here only its version for disjunctive blocks, the other version being symmetric. For simplicity, we assume that all $\wedge$-vertices of the graph have no successors, i.e., they denote T constants. Indeed, according to the definition of graphs corresponding to disjunctive blocks (see the beginning of Section 3), each $\wedge$-vertex having a non-constant successor in the graph can be assimilated to a $\vee$-vertex if its other successors are evaluated first (possibly by calling the resolution routines associated to other blocks).

Algorithm A4 combines the search for T ($\wedge$-sink) vertices with a detection of strongly connected components (SCCs) following Tarjan's classical algorithm [46]. Visited vertices are stored in a global set $A \subseteq V$ and vertices belonging to the currently explored SCCs are stored on a global stack, both initially empty. To each vertex y are associated five informations: a counter $p(y)$ indicating the current successor of y that must be explored (the successors in $E(y)$ are assumed to be indexed from 0 to $|E(y)| - 1$); a counter $n(y)$ containing the DFS number of y, which records that y was the $n(y)$-th vertex encountered during the DFS (this is computed by incrementing a global counter k, initially 0, each time a new vertex

is encountered); a counter $low(y)$ denoting the "lowlink" number of y [46], which records the smallest DFS number associated to the vertices reachable from y and belonging to the same SCC as y; a boolean $v(y)$ containing the value currently computed for y; and a boolean $stable(y)$ indicating whether y is stable, i.e., $v(y)$ has reached its final value.

The algorithm begins by initializing the various informations associated to vertex x (lines 9–25); in particular, x is stabilized if it has no successors, otherwise it is marked as unstable and its value is set to F. Then, each successor y of x is examined: if y has already been visited, its current value val is stored and if y belongs to the same SCC as x, the lowlink number of x is updated accordingly[2] (lines 29–32); if y is a new vertex, its value val is computed by a recursive call to A4 and the lowlink number of x is updated (lines 33–35). If val is T, vertex x is stabilized to T (since the block is disjunctive), its successor $s(x)$ is set to y for the purpose of diagnostic generation, and the while-loop is exited (lines 37–41). After termination of the while-loop, if x was stabilized to T or is the root of a SCC, all vertices belonging to that SCC (which are placed above x on the stack) are stabilized to the value of x (lines 46–53), since each of them can reach x via a path of $\vee$-vertices. Finally, the current value $v(x)$ is returned as result (line 54).

Figure 9 illustrates the result of executing algorithm A4 in order to evaluate the variable X_1 of a disjunctive block. The white box delimits the portion of the boolean graph explored during the DFS traversal performed by A4, and the grey box delimits the diagnostic produced (example for X_1), which is a path of length 3. The shaded box delimits a SCC containing vertices $\{X_2, X_4, X_7\}$, which were all stabilized to F when the root vertex X_2 was completely explored.

We discuss below the behaviour of algorithm A4 w.r.t. the three efficiency requirements R1–R3. The algorithm clearly satisfies requirement R1, because at each execution it performs a single DFS traversal (superposed with a linear time processing of vertices belonging to SCCs) having a worst-case time complexity $O(|V| + |E|)$. It also satisfies requirement R2, since every new vertex y explored by a recursive call to A4 is directly related to its (currently unstable) predecessor x, and thus y is inductively related, via unstable vertices, to the vertex of interest, given as argument to the topmost call of A4. Finally, the algorithm satisfies requirement R3, because upon termination of a call to A4, all explored vertices have their final value computed.

Like A3, algorithm A4 presents an advantage w.r.t. the general algorithms A1 and A2 (which can handle disjunctive or conjunctive blocks as well), namely its reduced memory consumption, achieved by exploiting the particular structure of the boolean graphs. Indeed, in a boolean graph corresponding to a disjunctive block, vertices are stabilized either by back-propagation of a T value from one of their successors, or by stabilization of the SCC to which they belong; in both cases, there is no need to keep track of backward dependencies. Therefore, A4 avoids storing the edges of the boolean graph, having a worst-case memory complexity $O(|V|)$ instead of $O(|V|+|E|)$ as the general algorithms A1 and A2 would exhibit when executed on boolean graphs corresponding to disjunctive or conjunctive blocks. Notice that, as for algorithm A3, the choice of a DFS traversal is crucial for obtaining this result: an

[2]The membership test of a vertex y to the currently explored SCC (line 30) is slightly different from the test used in [46]: instead of checking whether y is present on the stack, we use the boolean $stable(y)$, which is set to T (resp. F) for all vertices belonging to a completely explored (resp. to the currently explored) SCC.

```
 1. var A : 2^V; k : Nat; stack : V*;
 2. A := ∅;
 3. k := 0;
 4. stack := nil;
 5. function A4 (x, (V, E, L)) : Bool is
 6.    var n, p, low : V → Nat;
 7.        v, stable : V → Bool;
 8.        y, z : V; val : Bool;
 9.    if |E(x)| = 0 then
10.       if L(x) = ∧ then
11.          v(x) := T
12.       else
13.          v(x) := F
14.       endif;
15.       stable(x) := T
16.    else
17.       v(x) := F;
18.       stable(x) := F
19.    endif;
20.    p(x) := 0;
21.    n(x) := k;
22.    k := k + 1;
23.    low(x) := n(x);
24.    A := A ∪ {x};
25.    stack := push(x, stack);
26.    while p(x) < |E(x)| do
27.       y := (E(x))_{p(x)};
28.       if y ∈ A then
29.          val := v(y);
30.          if ¬stable(y) ∧ n(y) < n(x) then
31.             low(x) := min(low(x), n(y))
32.          endif
33.       else
34.          val := A4 (y, (V, E, L));
35.          low(x) := min(low(x), low(y))
36.       endif;
37.       if val then
38.          v(x) := T;
39.          stable(x) := T;
40.          s(x) := y;
41.          p(x) := |E(x)|
42.       else
43.          p(x) := p(x) + 1
44.       endif
45.    end;
46.    if v(x) ∨ low(x) = n(x) then
47.       repeat
48.          z := top(stack);
49.          v(z) := v(x);
50.          stable(z) := T;
51.          stack := pop(stack)
52.       until z = x
53.    endif;
54.    return v(x)
55. end
```

Figure 8: Algorithm A4: DFS-based local resolution of a disjunctive μ-block

$$\begin{cases} X_1 &=& X_2 \vee X_3 \\ X_2 &=& X_4 \\ X_3 &=& X_5 \vee X_6 \\ X_4 &=& X_7 \vee X_4 \\ X_5 &=& X_4 \vee X_7 \vee X_8 \\ X_6 &=& X_9 \\ X_7 &=& X_2 \\ X_8 &=& \mathsf{T} \\ X_9 &=& X_8 \end{cases}$$

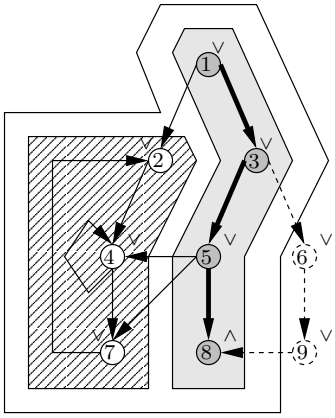


Figure 9: Result of executing A4

algorithm based upon a Bfs traversal, even specialized for disjunctive or conjunctive blocks, would be unable to satisfy all requirements R1–R3 without storing backward dependencies.

3.5 Generation of minimal-depth tree diagnostics

If the diagnostics generated for a boolean vertex contain cycles (e.g., counterexamples for μ-blocks or examples for ν-blocks [34]), the problem of finding a minimal-depth diagnostic is Np-complete [9]. If the diagnostics are sequences (e.g., examples in disjunctive μ-blocks or counterexamples in conjunctive ν-blocks) or trees, the problem can be solved efficiently. In this section, we present a linear time algorithm for computing a minimal-depth tree diagnostic for a boolean vertex x, at the condition that a diagnostic for x has already been computed. The interest of such an algorithm is twofold: it eliminates the disadvantage of Dfs-based algorithms (such as A1, A3, and A4), which may generate in practice diagnostics with prohibitive depth, and is even useful for Bfs-based algorithms (such as A2), which may not be guaranteed to generate minimal-depth diagnostics; and, since it is separated from the resolution process, it can be used in conjunction with any local Bes resolution algorithm equipped with diagnostic generation features.

The Diag algorithm that we propose (see Figure 10) requires as input, in addition to the variable x and the boolean graph (V, E, L) represented implicitly, also the depth k of a diagnostic[3] previously computed for x by using, e.g., one of the algorithms A1–A4. We describe only the version of Diag for computing examples in μ-blocks, the other version being dual. The Diag algorithm proceeds in two phases:

- Firstly, a forward Bfs traversal of the boolean graph is performed starting at x, up to a distance k from x (lines 5–37). Visited vertices are stored in a set $A \subseteq V$, and visited but unexplored vertices are stored in two queues q_1 and q_2, which keep the vertices placed at a distance i and $i+1$ from x, respectively. At each iteration i of the while-loop (lines 10–37), the Bfs queue q_1 is scanned and each of its vertices y is explored by

[3]The diagnostic depth can be easily computed by performing a Bfs traversal of the diagnostic starting at x.

the inner while-loop (lines 12–34). If y is a $\wedge$-sink vertex (constant T), it is inserted in a queue sq_1 and its counter $c(y)$, denoting the number of successors that must become T in order to stabilize y to T, is initialized to 0 (lines 15–18). Otherwise, the counter $c(y)$ is initialized to $|E(y)|$ or to 1, depending whether y is a $\wedge$- or a $\vee$-vertex (lines 19–24). Then, each successor z of y is visited; if z is new, it is inserted into the Bfs queue q_2, since it is placed at a distance $i+1$ from x (lines 27–30) and the backward dependency from z to y is stored in a set $d(z)$ (line 32). After termination of the inner while-loop, q_1 is replaced by q_2 in order to set up the next iteration (lines 35–36). The forward Bfs while-loop terminates when all vertices placed at distance k from x have been explored. At the end of this loop, every T vertex placed at a distance at most k from x has been captured in the queue sq_1.

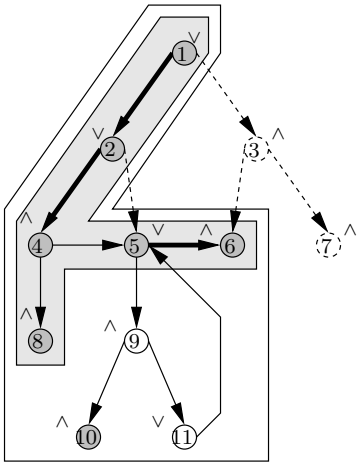
- Secondly, a backward Bfs traversal of the boolean graph is performed starting at the vertices accumulated during the first phase in the queue sq_1 (lines 38–56). At each iteration of the while-loop, the queue sq_1 is scanned and each of its vertices y is back-propagated to its predecessors by the inner while-loop (lines 40–54). For each predecessor z contained in the backward dependency set $d(y)$, its counter $c(z)$ is decremented; if it becomes 0, then z is stabilized to T, its successor $s(z)$ is initialized to y for the purpose of diagnostic generation, and z is inserted into a queue sq_2 (lines 45–50). After termination of the inner while-loop, sq_1 is replaced by sq_2 in order to set up the next iteration (line 55). The backward Bfs while-loop terminates when vertex x is stabilized to T (i.e., its counter has become 0). This loop preserves the following invariant: at the beginning of the j-th iteration of the loop, the queue sq_1 contains all vertices having a minimal-depth tree diagnostic of depth j whose terminal T vertices are placed at a distance at most k from x (these are the vertices contained in sq_1 computed at the end of the first phase). This invariant ensures the termination of the while-loop (since by hypothesis x must have a diagnostic of depth k) and the fact that it computes a minimal-depth diagnostic for x.

Figure 11 shows a boolean graph and three different diagnostics produced for vertex X_1 by applying algorithms A1, A2, and Diag. A1 and A2 produce tree diagnostics of depth 4 and 3, respectively; therefore, we can apply Diag for a depth $k = 3$. After the first phase (forward Bfs traversal on a distance k from X_1), the queue sq_1 contains vertices $\{X_6, X_7, X_8\}$. The second phase (backward Bfs traversal) converges in two iterations, indicated on Figure 11(c), producing a tree diagnostic of depth 2 for X_1.

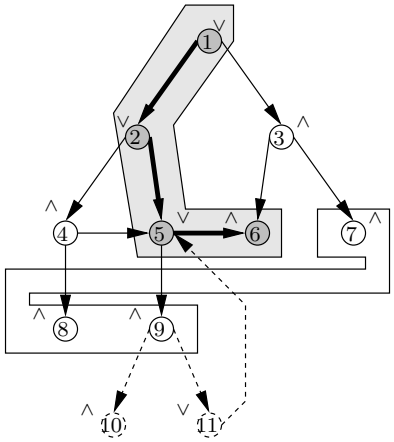
The Diag algorithm has a time complexity $O(|V| + |E|)$, since it consists of two Bfs traversals of the boolean subgraph containing the vertices at distance k from the vertex of interest x. In practice, Diag works faster when used in conjunction with the Bfs-based algorithm A2, since the diagnostics produced by A2 have a depth k usually much smaller than those produced by the Dfs-based algorithms A1, A3, and A4.

```
 1. procedure DIAG (x, (V,E,L), k) is
 2.   var A : 2^V; c : V → Nat;
 3.       d : V → 2^V; i : Nat;
 4.       q1, q2, sq1, sq2 : V*; y, z : V;
 5.   A := {x};
 6.   d(x) := ∅;
 7.   q1 := put(x, nil);
 8.   sq1 := nil;
 9.   i := 0;
10.   while i ≤ k do
11.     q2 := nil;
12.     while q1 ≠ nil do
13.       y := head(q1);
14.       q1 := tail(q1);
15.       if |E(y)| = 0 then
16.         if L(y) = ∧ then
17.           sq1 := put(y, sq1)
18.         endif
19.       else
20.         if L(y) = ∧ then
21.           c(y) := |E(y)|
22.         else
23.           c(y) := 1
24.         endif
25.       endif;
26.       forall z ∈ E(y) do
27.         if z ∉ A then
28.           A := A ∪ {z};
29.           d(z) := ∅;
30.           q2 := put(z, q2)
31.         endif;
32.         d(z) := d(z) ∪ {y}
33.       end
34.     end;
35.     q1 := q2;
36.     i := i + 1
37.   end;
38.   while c(x) ≠ 0 do
39.     sq2 := nil;
40.     while sq1 ≠ nil do
41.       y := head(sq1);
42.       sq1 := tail(sq1);
43.       forall z ∈ d(y) do
44.         if c(z) > 0 then
45.           c(z) := c(z) − 1;
46.           if c(z) = 0 then
47.             if L(z) = ∨ then
48.               s(z) := y
49.             endif;
50.             sq2 := put(z, sq2)
51.           endif
52.         endif
53.       end
54.     end;
55.     sq1 := sq2
56.   end
57. end
```

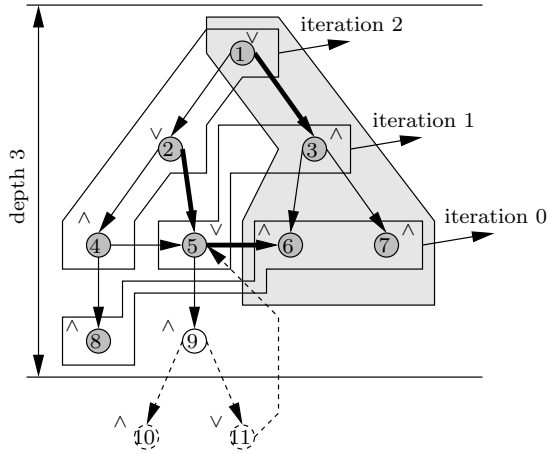
Figure 10: Algorithm DIAG: computation of a minimal-depth tree example in a μ-block



(a) diagnostic computed by A1



(b) diagnostic computed by A2



(c) diagnostic computed by Diag

Figure 11: Diagnostics computed by A1, A2, and Diag

4 Applications

In this section we study three applications of BES resolution in the field of finite-state verification: equivalence/preorder checking, model checking, and τ-confluence reduction, all performed on-the-fly. Various encodings of these problems in terms of BESs have been proposed in the literature [11, 1, 2, 33, 40]. Here we aim at giving a uniform presentation of these results and also at identifying particular cases where the memory-efficient algorithms A3 and A4 given in Sections 3.3 and 3.4 can be applied.

4.1 Equivalence checking

Labeled Transition Systems (LTSs) are natural semantic 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. A similar notation is used for 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 concatenated actions form a word of l. All states are assumed to be reachable from the initial state q_0 by sequences of transitions in T.

The approach of computing equivalence relations by performing a transformation to μ-calculus formulas or systems of modal fixed point equations was proposed in [28], and was the basis of the verification algorithms implemented in the Concurrency Workbench [10]. Here we present the encodings of equivalence relations directly in terms of BESs, which are derived from the characterizations given in [19, 2].

Let $M_i = (Q_i, A, T_i, q_{0i})$ be two LTSs ($i \in \{1, 2\}$). Table 1 shows the BES encodings of the equivalence between M_1 and M_2 modulo five widely-used equivalence relations: strong equivalence [41], branching equivalence [48], observational equivalence [37], $\tau^*.a$ equivalence [19], and safety equivalence [5]. 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. For each equivalence relation, the corresponding preorder relation is obtained simply by keeping the first half of the right-hand sides in the equations defining $X_{p,q}$, i.e., the following sets of variables: $Y_{b,p',q}$ (strong), $Y_{b,p,p',q}$ (branching), $Y_{p',q}$ and $Z_{a,p',q}$ (observational), $Y_{a,p',q}$ ($\tau^*.a$), $Y_{p,q}$ (safety). Other equivalences, such as delay bisimulation [39] and η-bisimulation [4], can be encoded using a similar scheme. Note that for all weak equivalences, the evaluation of the right-hand sides of equations requires to compute transitive closures of τ-transitions in one or both LTSs.

The BESs shown in Table 1 can be solved by using the general algorithms A1 and A2 (notice that the encodings given in the table are based upon computing successors of states, therefore allowing 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.

Table 1: BES encodings of five widely-used equivalence relations

Strong equivalence

$$\left\{\begin{array}{rcl} X_{p,q} & \stackrel{\nu}{=} & \bigwedge(\{Y_{b,p',q} \mid p \xrightarrow{b} p'\} \cup \{Z_{b,p,q'} \mid q \xrightarrow{b} q'\}) \\ Y_{b,p',q} & \stackrel{\nu}{=} & \bigvee\{X_{p',q'} \mid q \xrightarrow{b} q'\} \\ Z_{b,p,q'} & \stackrel{\nu}{=} & \bigvee\{X_{p',q'} \mid p \xrightarrow{b} p'\} \end{array}\right\} \quad p, p' \in Q_1, q, q' \in Q_2, \; b \in A_1 \cup A_2 \cup \{\tau\}$$

Branching equivalence

$$\left\{\begin{array}{rcl} X_{p,q} & \stackrel{\nu}{=} & \bigwedge(\{Y_{b,p,p',q} \mid p \xrightarrow{b} p'\} \cup \{Z_{b,p,q,q'} \mid q \xrightarrow{b} q'\}) \\ Y_{b,p,p',q} & \stackrel{\nu}{=} & \bigvee(\{X_{p',q} \mid b = \tau\} \cup \{U_{p,p',q',q''} \mid q \xrightarrow{\tau^*} q' \xrightarrow{b} q''\}) \\ Z_{b,p,q,q'} & \stackrel{\nu}{=} & \bigvee(\{X_{p,q'} \mid b = \tau\} \cup \{W_{p',p'',q,q'} \mid p \xrightarrow{\tau^*} p' \xrightarrow{b} p''\}) \\ U_{p,p',q',q''} & \stackrel{\nu}{=} & \bigwedge\{X_{p,q'}, X_{p',q''}\} \\ W_{p',p'',q,q'} & \stackrel{\nu}{=} & \bigwedge\{X_{p',q}, X_{p'',q'}\} \end{array}\right\} \quad \begin{array}{l} p, p', p'' \in Q_1, \\ q, q', q'' \in Q_2, \\ b \in A_1 \cup A_2 \cup \{\tau\} \end{array}$$

Observational equivalence

$$\left\{\begin{array}{rcl} X_{p,q} & \stackrel{\nu}{=} & \bigwedge(\{Y_{p',q} \mid p \xrightarrow{\tau} p'\} \cup \{Z_{a,p',q} \mid p \xrightarrow{a} p'\} \cup \\ & & \quad \{U_{p,q'} \mid q \xrightarrow{\tau} q'\} \cup \{W_{a,p,q'} \mid q \xrightarrow{a} q'\}) \\ Y_{p',q} & \stackrel{\nu}{=} & \bigvee\{X_{p',q'} \mid q \xrightarrow{\tau^*} q'\} \\ Z_{a,p',q} & \stackrel{\nu}{=} & \bigvee\{X_{p',q'} \mid q \xrightarrow{\tau^*.a.\tau^*} q'\} \\ U_{p,q'} & \stackrel{\nu}{=} & \bigvee\{X_{p',q'} \mid p \xrightarrow{\tau^*} p'\} \\ W_{a,p,q'} & \stackrel{\nu}{=} & \bigvee\{X_{p',q'} \mid p \xrightarrow{\tau^*.a.\tau^*} p'\} \end{array}\right\} \quad p, p' \in Q_1, q, q' \in Q_2, \; a \in A_1 \cup A_2$$

$\tau^*.a$ equivalence

$$\left\{\begin{array}{rcl} X_{p,q} & \stackrel{\nu}{=} & \bigwedge(\{Y_{a,p',q} \mid p \xrightarrow{\tau^*.a} p'\} \cup \{Z_{a,p,q'} \mid q \xrightarrow{\tau^*.a} q'\}) \\ Y_{a,p',q} & \stackrel{\nu}{=} & \bigvee\{X_{p',q'} \mid q \xrightarrow{\tau^*.a} q'\} \\ Z_{a,p,q'} & \stackrel{\nu}{=} & \bigvee\{X_{p',q'} \mid p \xrightarrow{\tau^*.a} p'\} \end{array}\right\} \quad p, p' \in Q_1, q, q' \in Q_2, \; a \in A_1 \cup A_2'$$

Safety equivalence

$$\left\{\begin{array}{rcl} X_{p,q} & \stackrel{\nu}{=} & \bigwedge\{Y_{p,q}, Y_{q,p}\} \\ Y_{p,q} & \stackrel{\nu}{=} & \bigwedge\{Z_{a,p',q} \mid p \xrightarrow{\tau^*.a} p'\} \\ Z_{a,p',q} & \stackrel{\nu}{=} & \bigvee\{Y_{p',q'} \mid q \xrightarrow{\tau^*.a} q'\} \end{array}\right\} \quad p, p' \in Q_1, q, q' \in Q_2, \; a \in A_1 \cup A_2$$

4.1.1 Acyclic case

When M_1 or M_2 is acyclic, the BESs associated to some of the five equivalence relations considered and to their corresponding preorders become acyclic as well. This is easy to see for strong equivalence (and its preorder): 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 observational equivalence (and its preorder), 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 consists in establishing the correctness of large event traces produced by intensive simulation of a system w.r.t. the formal specification of the system [23]. Assuming that the system specification is given as an LTS M_1 and the set of traces is given as an LTS M_2 (obtained by merging the initial states of all traces), the verification amounts to checking the inclusion $M_1 \preceq M_2$ modulo the strong or safety preorder.

4.1.2 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 equivalence, the BESs of the other equivalences being simplified similarly. 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 F (states p and q are not strongly equivalent). Otherwise, the right-hand side of the equation becomes $\bigwedge(\{Y_{b,p'_b,q}\} \cup \{X_{p'_b,q'} \mid q \xrightarrow{b} q'\})$, which reduces to $\bigwedge\{X_{p'_b,q'} \mid q \xrightarrow{b} q'\}$ since the first conjunct $Y_{b,p'_b,q}$, equal to $\bigvee\{X_{p',q'} \mid q \xrightarrow{b} q'\}$, is absorbed by the second one $\{X_{p'_b,q'} \mid q \xrightarrow{b} q'\}$. 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 deterministic and τ-free (i.e., does not contain τ-transitions). For example, if M_1 is deterministic and τ-free, the equation defining $X_{p,q}$ for observational equivalence becomes $X_{p,q} \stackrel{\nu}{=} \bigwedge(\{X_{p,q'} \mid q \xrightarrow{\tau} q'\} \cup \{X_{p'_a,q'} \mid q \xrightarrow{a} q'\})$. Similar simplifications were identified in [19]; we believe they can be obtained more elegantly and systematically 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 [19], 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.

Table 2: Translation from state to boolean formulas

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

4.2 Model checking

Alternation-free BESs allow to encode the alternation-free μ-calculus [12, 1, 33]. 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):

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

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 [12, 1, 33] 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 $\{X_p \stackrel{\sigma}{=} op(\varphi')(\varphi')_p\}_{p \in Q}$, where variables X_p express that state p satisfies X, and the operator $op(\varphi')$ and the right-hand side boolean formulas $(\varphi')_p$ are produced by the translation given in Table 2.

This kind of BES can be solved by the general algorithms A1 and A2 given in Section 3 (notice that the translation given in Table 2 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.

4.2.1 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 [35]. 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 using the succinct translation given in [29].

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 event traces (represented as acyclic LTSs M by merging their initial states) produced by intensive simulation of a system [35].

4.2.2 Disjunctive/conjunctive case

When φ is a formula of CTL [8], ACTL (Action-based CTL) [38] or PDL [20], the BES resulting after translation is in disjunctive or conjunctive form. Table 3 shows the translations of CTL and PDL operators into alternation-free μ-calculus [16] (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 (built from action names and boolean operators) are used inside diamond and box modalities instead of simple action names [17].

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 following equation block:

$$\left\{ \begin{array}{lcl} X_p & \stackrel{\mu}{=} & \bigvee\{(\varphi_2)_p, Y_p\} \\ Y_p & \stackrel{\mu}{=} & \bigwedge\{(\varphi_1)_p, Z_p\} \\ Z_p & \stackrel{\mu}{=} & \bigvee\{X_q \mid p \rightarrow q\} \end{array} \right\}_{p \in Q}$$

Table 3: Translation of CTL and PDL into alternation-free μ-calculus

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	$\langle \alpha \rangle\, \varphi$	$\langle \alpha \rangle\, \varphi$
	$\langle \varphi_1 ? \rangle\, \varphi_2$	$\varphi_1 \wedge \varphi_2$
	$\langle \beta_1 ; \beta_2 \rangle\, \varphi$	$\langle \beta_1 \rangle\, \langle \beta_2 \rangle\, \varphi$
	$\langle \beta_1 \cup \beta_2 \rangle\, \varphi$	$\langle \beta_1 \rangle\, \varphi \vee \langle \beta_2 \rangle\, \varphi$
	$\langle \beta^* \rangle\, \varphi$	$\mu X.\varphi \vee \langle \beta \rangle\, X$

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 below:

$$\left\{ \begin{array}{lcl} X_p & \stackrel{\mu}{=} & (\varphi_2)_p \vee Y_p \\ Y_p & \stackrel{\mu}{=} & (\varphi_1)_p \wedge Z_p \wedge \bigwedge_{p \rightarrow q} X_q \\ Z_p & \stackrel{\mu}{=} & \bigvee_{p \rightarrow q} \mathsf{T} \end{array} \right\}_{p \in Q}$$

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 [17]. 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. $[\beta] \varphi$): 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.

4.3 Tau-confluence reduction

Given an LTS $M = (Q, A, T, q_0)$, the notion of τ-confluent transition formalizes the intuition that certain τ-transitions going out of a state do not change the future behaviour of the system starting at that state. Therefore, computing a set of τ-transitions with this property allows to reduce M to another LTS which is generally smaller, but branching equivalent to M. This reduction based upon τ-confluence, proposed in [25], is a form of partial-order reduction defined in the context of LTSs. A transition $q_1 \stackrel{\tau}{\rightarrow} q_2$ is τ-confluent (see Figure 12) if for every other transition $q_1 \stackrel{b}{\rightarrow} q_3$, one of the following conditions holds:

(a) there exists a transition $q_2 \stackrel{b}{\rightarrow} q_3$;

(b) there exists a state q_4 and two transitions $q_2 \stackrel{b}{\rightarrow} q_4$ and $q_3 \stackrel{\tau}{\rightarrow} q_4$ such that the latter is also τ-confluent;

(c) $b = \tau$ and there exists a transition $q_3 \stackrel{\tau}{\rightarrow} q_2$ which is also τ-confluent.

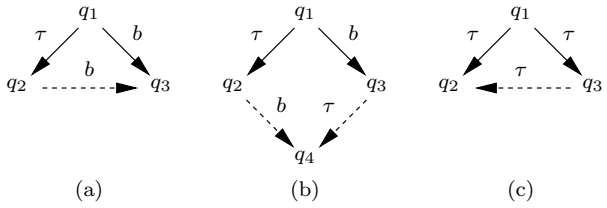


Figure 12: τ-confluent transitions in an LTS

Table 4: BES encoding of τ-confluence

$$\left\{\begin{array}{lcl} X_{q_1,q_2} & \stackrel{\nu}{=} & \bigwedge\{Y_{b,q_2,q_3} \mid q_1 \stackrel{\tau}{\rightarrow} q_2 \wedge q_1 \stackrel{b}{\rightarrow} q_3 \wedge \\ & & \quad q_2 \stackrel{b}{\not\rightarrow} q_3\} \\ Y_{b,q_2,q_3} & \stackrel{\nu}{=} & \bigvee(\{X_{q_3,q_4} \mid q_2 \stackrel{b}{\rightarrow} q_4 \wedge q_3 \stackrel{\tau}{\rightarrow} q_4\} \cup \\ & & \quad \{X_{q_3,q_2} \mid b = \tau \wedge q_3 \stackrel{\tau}{\rightarrow} q_2\}) \end{array}\right\} \begin{array}{l} q_1, q_2 \in Q, \\ q_3, q_4 \in Q, \\ b \in A \cup \{\tau\} \end{array}$$

Intuitively, a transition $q_1 \stackrel{\tau}{\rightarrow} q_2$ is τ-confluent if every other transition going out of q_1 can be simulated after executing the τ-confluent transition. For every τ-confluent transition, its source and target states are branching equivalent [25].

Checking the τ-confluence of a transition can be straightforwardly encoded by using a BES with a single ν-block [40], as shown in Table 4. For each transition $q_1 \stackrel{\tau}{\rightarrow} q_2$, the BES defines a variable X_{q_1,q_2} indicating whether the transition is τ-confluent or not (the maximal fixed point is used in order to characterize the maximal τ-confluent set of transitions contained in the LTS).

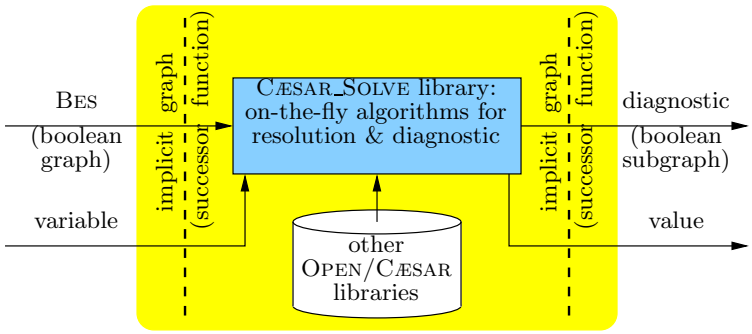
This kind of BES can be solved by using the general algorithms A1 and A2 (notice that the encoding given in Table 4 allows to construct the LTS on-the-fly during BES resolution). Once a confluent τ-transition $q_1 \stackrel{\tau}{\rightarrow} q_2$ has been detected, all the other transitions going out of q_1 can be safely removed without losing branching equivalence; this on-the-fly reduction procedure is called τ-prioritization [25]. Our preliminary experimental results show that τ-prioritization reduces the number of LTS states and transitions up to one order of magnitude.

5 Implementation and experiments

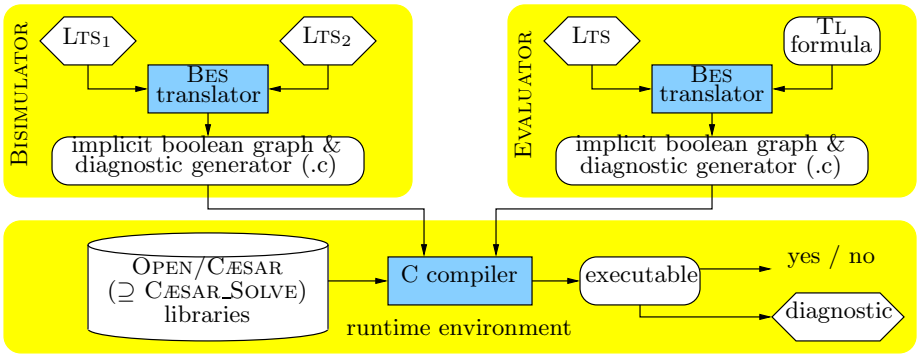
We implemented the BES resolution algorithms A1–A4 described in Section 3 in a generic software library, called CÆSAR_SOLVE [7], which is built upon the primitives of the OPEN/CÆSAR environment for on-the-fly exploration of LTSs [21]. CÆSAR_SOLVE is used by the BISIMULATOR equivalence/preorder checker, the EVALUATOR model checker, and the REDUCTOR tool. We briefly describe the architecture of these tools and give some experimental results concerning the A1–A4 resolution algorithms.

5.1 The CAESAR_SOLVE library

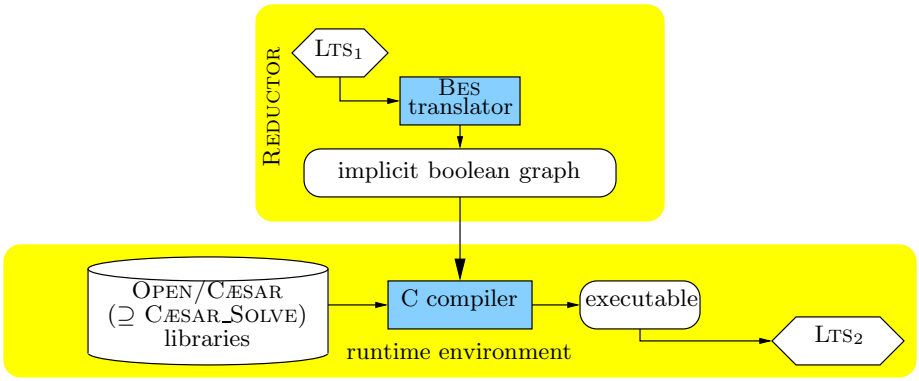
The CÆSAR_SOLVE library (see Figure 13(a)) provides an Application Programming Interface (API) allowing to solve on-the-fly a variable of a BES. It takes as input the boolean graph associated to the BES together with the variable of interest, and produces as output the value of the variable, possibly 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.).



(a) CÆSAR_SOLVE library



(b) EVALUATOR and BISIMULATOR



(c) REDUCTOR

Figure 13: The CÆSAR_SOLVE library and its associated tools BISIMULATOR, EVALUATOR, and REDUCTOR

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 [21]. 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.

Three on-the-fly verification tools developed within CADP are currently using the CÆSAR_SOLVE library (see Figure 13(b) and 13(c)): BISIMULATOR, an equivalence/preorder checker comparing two LTSs modulo the five relations mentioned in Section 4.1; EVALUATOR, a model checker for regular alternation-free μ-calculus [36] over LTSs; and REDUCTOR, a tool performing various reductions on LTSs, among which τ-confluence. Each tool translates its corresponding verification problem into a BES resolution following the encodings given in Section 4; in addition, BISIMULATOR and EVALUATOR identify the particular cases suitable for algorithms A3–A4, and translate back the diagnostics produced by the library in terms of the 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 versions of) three communication protocols[4]: an alternating bit protocol (ABP), a bounded retransmission protocol (BRP), and a distributed leader election protocol (DLE).

The results are shown in Table 5. The 1st series of experiments compares A1 with A2 as regards diagnostic depth (measured in number of transitions). The 2nd and 3rd series compare A1 with A3, respectively 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 series of experiments compare each protocol LTS modulo strong equivalence with an erroneous LTS, and verify an invalid property on the protocol LTS (both problems yield counterexamples). The 2nd series of 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 series of 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.

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

Table 5: Performance of resolution algorithms for equivalence checking and model checking

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

We also performed a series of experiments for investigating the effectiveness of the τ-confluence reduction. Besides the three communication protocols used previously, we also considered six other examples of distributed systems available in the demo examples of Cadp: an atomic multicast protocol (Rel), a clustered file system (Cfs), a distributed summing protocol (Sum), the Erathostene's sieve (Esv), an open distributed processing trader (Odp), and an asynchronous circuit (Des). Table 6 shows the size of the corresponding Ltss before and after reduction by τ-confluence. For the first four communication protocols and the Cfs, we observe reductions of up to one order of magnitude. For the last four examples, which are more prone to partial order reduction since they involve many loosely-coupled processes, the reduced Ltss were up to three orders of magnitude smaller.

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 [21] of the Cadp toolbox [22]. It implements an application-independent representation of Bess, precisely defined by an Api [7]. The library currently offers four resolution algorithms A1–A4, A2 being optimized to produce small-depth diagnostics and A3, A4 being

Table 6: Reduction by τ-confluence

App.	Original size		Reduced size			
	States	Trans.	States	%	Trans.	%
ABP	935000	3001594	20802	2.2	60901	2.0
BRP	2227357	3262644	400721	17.9	1436008	44.0
DLE	94231	252988	41795	44.3	137591	54.3
REL	113590	385798	38285	33.7	109971	28.5
CFS	140136	679452	4058	2.8	31128	4.5
SUM	156957	767211	178	0.11	481	0.06
ESV	23627	84707	11	0.04	10	0.01
ODP	85640	595864	433	0.50	2276	0.38
DES	1418	3539	6	0.42	7	0.19

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. 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 [15], the Gauss elimination-based algorithm proposed in [33], and the algorithms for disjunctive/conjunctive BESs with arbitrary alternation depth recently devised in [24]. 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 equivalence checking by encoding other relations between standard LTSs, such as delay bisimulation [39] and η-bisimulation [4], or equivalence relations between probabilistic and stochastic LTSs [26]; on-the-fly Horn clause resolution by using the translations from Horn clauses to BESs proposed in [14, 32]; and 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 [18].

Acknowledgement

We are grateful to Frédéric Lang for developing the REDUCTOR 5.0 tool, already integrated in CADP, which performs several on-the-fly reductions on LTSs and also includes the τ-confluence reduction implemented by the author.

References

[1] H. R. Andersen. Model checking and boolean graphs. Theoretical Computer Science, 126(1):3–30, April 1994

[2] H. R. Andersen and B. Vergauwen. Efficient checking of behavioural relations and modal assertions using fixed-point inversion. In P. Wolper (ed.), Proc. of the 7th International Conference on Computer Aided Verification CAV'95 (Liege, Belgium), Lecture Notes in Computer Science 939, Berlin, Heidelberg, New York: Springer-Verlag, July 1995, pp. 142–154

[3] A. Arnold and P. Crubillé. A linear algorithm to solve fixed-point equations on transition systems. Information Processing Letters, 29(1):57–66, September 1988

[4] J. C. M. Baeten and R. J. van Glabbeek. Another look at abstraction in process algebra. In Th. Ottman (ed.), Proc. of ICALP'87, Lecture Notes in Computer Science 267, Berlin, Heidelberg, New York: Springer-Verlag, 1987, pp. 84–94

[5] A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodríguez, and J. Sifakis. Safety for branching time semantics. In Proc. of the 18th International Colloquium on Automata, Languages, and Programming ICALP'91 (Madrid, Spain), Lecture Notes in Computer Science 510, Berlin, Heidelberg, New York: Springer-Verlag, July 1991

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, C-35(8), 1986

[7] R. Mateescu. Man page of the Cæsar_Solve_1 library, INRIA Rhône-Alpes / VASY, January 2005

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems, 8(2):244–263, April 1986

[9] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In Proc. of the 32nd ACM/IEEE Design Automation Conference DAC'95 (San Francisco, CA, USA), 1995, pp. 427–432

[10] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: a semantics-based verification tool for finite state systems. ACM Transactions on Programming Languages and Systems, 15(1):36–72, January 1993

[11] R. Cleaveland and B. Steffen. Computing behavioural relations, logically. In J. L. Albert, B. Monien, and M. Rodríguez-Artalejo (eds.), Proc. of the 18th International Colloquium on Automata, Languages, and Programming ICALP'91 (Madrid, Spain), Lecture Notes in Computer Science 510, Berlin, Heidelberg, New York: Springer-Verlag, July 1991, pp. 127–138

[12] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. In K. G. Larsen and A. Skou (eds.), Proc. of the 3rd International Conference on Computer Aided Verification CAV'91 (Aalborg, Denmark), Lecture Notes in Computer Science 575, Berlin, Heidelberg, New York: Springer-Verlag, July 1991, pp. 48–58

[13] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. Formal Methods in System Design, 2(2):121–147, April 1993

[14] W.F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. Journal of Logic Programming, 3:267–284, 1984

[15] X. Du, S. A. Smolka, and R. Cleaveland. Local model checking and protocol analysis. Springer International Journal on Software Tools for Technology Transfer (STTT), 2(3):219–241, 1999

[16] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In Proc. of the 1st LICS, 1986, pp. 267–278

[17] A. Fantechi, S. Gnesi, and G. Ristori. From ACTL to mu-calculus. In Proc. of the ERCIM Workshop on Theory and Practice in Verification (Pisa, Italy), IEI-CNR, December 1992, pp. 3–10

[18] J.-C. Fernandez, C. Jard, T. Jéron, L. Nedelka, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T. A. Henzinger (eds.), Proc. of the 8th International Conference on Computer-Aided Verification CAV'96 (Rutgers University, New Brunswick, NJ, USA), Lecture Notes in Computer Science 1102, Berlin, Heidelberg, New York: Springer-Verlag, August 1996, pp. 348–359

[19] J.-C. Fernandez and L. Mounier. "On the fly" verification of behavioural equivalences and preorders. In K. G. Larsen and A. Skou (eds.), Proc. of the 3rd International Workshop on Computer-Aided Verification CAV'91 (Aalborg, Denmark), Lecture Notes in Computer Science 575, Berlin, Heidelberg, New York: Springer-Verlag, July 1991

[20] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. Journal of Computer and System Sciences, (18):194–211, 1979

[21] H. Garavel. OPEN/CÆSAR: an open software architecture for verification, simulation, and testing. In B. Steffen (ed.), Proc. of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal), Lecture Notes in Computer Science 1384, Berlin, Heidelberg, New York: Springer-Verlag, March 1998, pages 68–84. Full version available as INRIA Research Report RR-3352

[22] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. European Association for Software Science and Technology (EASST) Newsletter, 4:13–24, August 2002. Also available as INRIA Technical Report RT-0254

[23] H. Garavel and R. Mateescu. SEQ.OPEN: a tool for efficient trace-based verification. In S. Graf and L. Mounier (eds.), Proc. of the 11th International SPIN Workshop on Model Checking of Software SPIN'2004 (Barcelona, Spain), Lecture Notes in Computer Science 2989, Berlin, Heidelberg, New York: Springer-Verlag, April 2004, pp. 150–155

[24] J. F. Groote and M. Keinänen. Solving disjunctive/conjunctive boolean equation systems with alternating fixed points. In K. Jensen and A. Podelski (eds.), Proc. of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2004 (Barcelona, Spain), Lecture Notes in Computer Science 2988, Berlin, Heidelberg, New York: Springer-Verlag, March 2004, pp. 436–450

[25] J. F. Groote and J. van de Pol. State space reduction using partial τ-confluence. In M. Nielsen and B. Rovan (eds.), Proc. of the 25th International Symposium on Mathematical Foundations of Computer Science MFCS'2000 (Bratislava, Slovakia), Lecture Notes in Computer Science 1893, Berlin, Heidelberg, New York: Springer-Verlag, August 2000, pp. 383–393

[26] H. Hermanns and M. Siegle. Bisimulation algorithms for stochastic process algebras and their BDD-based implementation. In J.-P. Katoen (ed.), Proc. of the 5th International AMAST Workshop ARTS'99 (Bamberg, Germany), Lecture Notes in Computer Science 1601, Berlin, Heidelberg, New York: Springer-Verlag, May 1999, pp. 244–265

[27] A. J. Hu. Techniques for efficient formal verification using binary decision diagrams. PhD thesis, Stanford University, December 1995

[28] A. Ingolfsdottir and B. Steffen. Characteristic formulae for processes with divergence. Information and Computation, 110(1):149–163, June 1994

[29] O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. Journal of the ACM, 47(2):312–360, March 2000

[30] K. G. Larsen. Efficient local correctness checking. In G. v. Bochmann and D. K. Probst (eds.), Proc. of the 4th International Workshop in Computer Aided Verification CAV'92 (Montréal, Canada), Lecture Notes in Computer Science 663, Berlin, Heidelberg, New York: Springer-Verlag, June-July 1992, pp. 30–43

[31] X. Liu, C. R. Ramakrishnan, and S. A. Smolka. Fully local and efficient evaluation of alternating fixed points. In B. Steffen (ed.), Proc. of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal), Lecture Notes in Computer Science 1384, Berlin, Heidelberg, New York: Springer-Verlag, March 1998, pp. 5–19

[32] X. Liu and S. A. Smolka. Simple linear-time algorithms for minimal fixed points. In Proc. of the 25th International Colloquium on Automata, Languages, and Programming ICALP'98 (Aalborg, Denmark), Lecture Notes in Computer Science 1443, Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 53–66

[33] A. Mader. Verification of Modal Properties Using Boolean Equation Systems. VERSAL 8, Bertz Verlag, Berlin, 1997

[34] R. Mateescu. Efficient diagnostic generation for boolean equation systems. In S. Graf and M. Schwartzbach (eds.), Proc. of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2000 (Berlin, Germany), Lecture Notes in Computer Science 1785, Berlin, Heidelberg, New York: Springer-Verlag, March 2000, pp. 251–265. Full version available as INRIA Research Report RR-3861

[35] R. Mateescu. Local model-checking of modal mu-calculus on acyclic labeled transition systems. In J.-P. Katoen and P. Stevens (eds.), Proc. of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002 (Grenoble, France), Lecture Notes in Computer Science 2280, Berlin, Heidelberg, New York: Springer-Verlag, April 2002, pp. 281–295. Full version available as INRIA Research Report RR-4430

[36] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Science of Computer Programming, 46(3):255–281, March 2003

[37] R. Milner. Communication and Concurrency. Prentice-Hall, 1989

[38] R. De Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In Semantics of Concurrency, Lecture Notes in Computer Science 469, Berlin, Heidelberg, New York: Springer-Verlag, 1990, pp. 407–419

[39] R. De Nicola, U. Montanari, and F. Vaandrager. Back and forth bisimulations. Computer Science Report CS R9021, Centrum voor Wiskunde en Informatica, Amsterdam, May 1990

[40] G. Pace, F. Lang, and R. Mateescu. Calculating τ-confluence compositionally. In Jr W. A. Hunt and F. Somenzi (eds.), Proc. of the 15th International Conference on Computer Aided Verification CAV'2003 (Boulder, Colorado, USA), Lecture Notes in Computer Science 2725, Berlin, Heidelberg, New York: Springer-Verlag, July 2003, pp. 446–459. Full version available as INRIA Research Report RR-4918

[41] D. Park. Concurrency and automata on infinite sequences. In P. Deussen (ed.), Theoretical Computer Science, Lecture Notes in Computer Science 104, Berlin, Heidelberg, New York: Springer-Verlag, March 1981, pp. 167–183

[42] B. Steffen, A. Classen, M. Klein, J. Knoop, and T. Margaria. The fixpoint analysis machine. In J. Lee and S. A. Smolka (eds.), Proc. of the 6th International Conference on Concurrency Theory CONCUR'95 (Philadelphia, Pennsylvania, USA), Lecture Notes in Computer Science 962, Berlin, Heidelberg, New York: Springer-Verlag, August 1995, pp. 72–87

[43] B. Steffen, T. Margaria, A. Classen, and V. Braun. The METAFrame'95 environment. In R. Alur and T. A. Henzinger (eds.), Proc. of the 8th Conference on Computer-Aided Verification CAV'96 (New Brunswick, New Jersey, USA), Lecture Notes in Computer Science 1102, Berlin, Heidelberg, New York: Springer-Verlag, August 1996, pp. 450–453

[44] B. Steffen, T. Margaria, and V. Braun. The Electronic Tool Integration platform: concepts and design. Springer International Journal on Software Tools for Technology Transfer (STTT), 1–2(1):9–30, December 1997

[45] P. Stevens and C. Stirling. Practical model-checking using games. In B. Steffen (ed.), Proc. of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal), Lecture Notes in Computer Science 1384, Berlin, Heidelberg, New York: Springer-Verlag, March 1998, pp. 85–101

[46] R. E. Tarjan. Depth first search and linear graph algorithms. SIAM Journal of Computing, 1(2):146–160, 1972

[47] M. Thorup. Near-optimal fully-dynamic graph connectivity. In Proc. of the 32nd ACM Symposium on the Theory of Computing STOC'2000 (Portland, Oregon), 2000

[48] R. J. van Glabbeek and W. P. Weijland. Branching-time and abstraction in bisimulation semantics (extended abstract). Computer Science Report CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in Proc. of IFIP 11th World Computer Congress (San Francisco, USA), 1989

[49] B. Vergauwen and J. Lewi. A linear algorithm for solving fixed-point equations on transition systems. In Proc. of the 17th Colloquium on Trees in Algebra and Programming CAAP'92 (Rennes, France), Lecture Notes in Computer Science 581, Berlin, Heidelberg, New York: Springer-Verlag, February 1992, pp. 322–341

[50] B. Vergauwen and J. Lewi. Efficient local correctness checking for single and alternating boolean equation systems. In S. Abiteboul and E. Shamir (eds.), Proc. of the 21st International Colloquium on Automata, Logic, and Programming ICALP'94 (Vienna, Austria), Lecture Notes in Computer Science 820, Berlin, Heidelberg, New York: Springer-Verlag, July 1994, pp. 304–315

[51] B. Vergauwen, J. Wauman, and J. Lewi. Efficient fixpoint computation. In Proc. of the 1st International Static Analysis Symposium SAS'94 (Namur, Belgium), Lecture Notes in Computer Science 864, Berlin, Heidelberg, New York: Springer-Verlag, September 1994, pp. 314–328

[52] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R.K. Ranjan, and F. Somenzi. A performance study of BDD-based model-checking. In Proc. of FMCAD'98, Lecture Notes in Computer Science 1522, Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 255–289

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopˆole de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399