

Improved On-the-Fly Equivalence Checking using Boolean Equation Systems

Radu Mateescu and Emilie Oudot*

INRIA / VASY project-team

Faculté des Sciences Mirande, bât. LE21, F-21000 Dijon, France

Email: {Radu.Mateescu,Emilie.Oudot}@inria.fr

Abstract. Equivalence checking is a classical verification method for ensuring the compatibility of a finite-state concurrent system (*protocol*) with its desired external behaviour (*service*) by comparing their underlying labeled transition systems (LTSS) modulo an appropriate equivalence relation. The local (or *on-the-fly*) approach for equivalence checking combats state explosion by exploring the synchronous product of the LTSS incrementally, thus allowing an efficient detection of errors in complex systems. However, when the two LTSS being compared are equivalent, the on-the-fly approach is outperformed by the global one, which completely builds the LTSS and computes the equivalence classes between states using partition refinement. In this paper, we consider the approach based on translating the on-the-fly equivalence checking problem in terms of the local resolution of a boolean equation system (BES). We propose two enhancements of the approach in the case of equivalent LTSS: a new, faster encoding of equivalence relations in terms of BESS, and a new local BES resolution algorithm with a better average complexity. These enhancements were incorporated into the BISIMULATOR 2.0 equivalence checker of the CADP toolbox, and they led to significant performance improvements w.r.t. existing on-the-fly equivalence checking algorithms.

1 Introduction

Equivalence checking is a classical verification method for finite-state concurrent systems that consists in comparing the behaviour of the system under design (typically, a *protocol* or a low-level hardware description) with its desired external behaviour (typically, a *service* or a high-level hardware description) modulo a suitable equivalence relation. Protocol and service behaviours are usually represented as labeled transition systems (LTSS), and the relations most used for comparing them are the *bisimulations* defined in the framework of process algebras, such as CCS [36], CSP [8], or ACP [5] and of the formal specification languages inspired from them, such as LOTOS [24] or CHP [30]. In practice, LTSS are often represented in two complementary ways, which also determine the nature of equivalence checking algorithms: either *explicitly*, by their list of states

* This research was partially funded by the EC-MOAN project no. 043235 of the FP6-NEST-PATH-COM European program.

and transitions, or *implicitly*, by their “successor function” returning the set of transitions going out of a given state. The implicit and explicit LTS representations are suitable for protocols (which are usually large) and services (which are usually small), respectively.

There are basically two approaches for equivalence checking: the *global* one [14], which operates on explicit LTSS, computes the equivalence classes of states by using partition refinement and then checks whether the initial states of the two LTSS fall into the same equivalence class; and the *local* one [11], which operates on implicit LTSS, explores the synchronous product between the two LTSS and searches for mismatches indicating the non equivalence of their initial states. Global algorithms are more effective when the two LTSS are equivalent, but require their complete construction, which is limited for large systems by the amount of memory available. Local (or *on-the-fly*) algorithms are more effective when the LTSS are not equivalent, allowing the detection of errors in complex systems even when the global approach would fail. Therefore, on-the-fly algorithms are useful at the beginning of the verification process, when errors occur frequently and must be detected quickly, whereas global algorithms are more suitable at a later stage, once the formal descriptions of the protocol and the service are stable and their underlying LTSS become equivalent.

Our objective is to improve the performance of on-the-fly equivalence checking algorithms when the LTSS to be compared are equivalent, which is the worst case for this class of algorithms because it forces them to explore the synchronous product of the two LTSS entirely. This would combine the advantages of global and local verification, making the on-the-fly approach suitable throughout the verification process. We consider here the technique relying on the translation of on-the-fly equivalence checking to the local resolution of a boolean equation system (BES) [2, 34]. This technique involves two clearly separated aspects, namely the BES encodings of bisimulation relations and the local BES resolution algorithms, which can be developed and optimized independently. To improve performance, we seek to enhance both these aspects.

First, we devise new BES encodings of the branching [41] and weak [36] bisimulations, obtained by migrating a part of the computation of transitive reflexive closures over internal steps (τ -closures) into the boolean equations. This simplifies the structure of BES equations considerably and reveals to be faster than computing τ -closures separately by using specialized algorithms [33]. Second, we propose a new local BES resolution algorithm, which exhibits a smaller average complexity than previously published algorithms [1, 43, 16, 34]. Our algorithm is based on a suspend/resume depth first search (sr-DFS) of the dependencies between boolean variables, and stops as soon as the BES portion explored contains a single example or counterexample for the boolean variable to be solved, therefore being optimal from this point of view.

These two enhancements led to version 2.0 of the BISIMULATOR [34] equivalence checker of the CADP [21] verification toolbox. The tool was developed using the generic OPEN/CÆSAR [20] environment for on-the-fly manipulation of LTSS, and uses as verification engine the CÆSAR_SOLVE [34] library for on-the-fly

resolution of BESS. The enhancements led to a significant performance increase w.r.t. BISIMULATOR 1.0, as we observed on LTSS generated from protocol and hardware descriptions or taken from the VLTS benchmark suite [44].

Related work. On-the-fly equivalence checking algorithms [11] received relatively little attention from the verification community, the research being mainly focused on optimizing global algorithms based on partition refinement [14, 19]. Among the first on-the-fly equivalence checking algorithms were those proposed in [17] and subsequently implemented in the ALDÉBARAN tool [18]. Two different algorithms were elaborated: the first one compares deterministic LTSS by searching their synchronous product for a pair of non equivalent states, and the second one handles nondeterministic LTSS by assuming that certain couples of states are equivalent and by backtracking in the synchronous product whenever such an assumption turns out to be wrong. The verification technique based on BES resolution allows one to reproduce the first algorithm by observing that the BESS corresponding to bisimulations between deterministic LTSS are conjunctive, and by devising a specialized local resolution algorithm for this case [34]. The algorithm for the nondeterministic case is outperformed in practice by local BES resolution algorithms, as it was observed experimentally [4].

Another approach of checking the equivalence of two LTSS is to rephrase the problem as the model checking on one LTS of a *characteristic formula* [23] in modal μ -calculus derived from the other LTS. This approach was elegantly implemented in the Concurrency Workbench [12, 9], but was hampered in practice for large LTSS by the prohibitive size of characteristic formulas, which is at least of the same order as the LTS size. The quest for performance was pursued by considering other intermediate formalisms suitable for representing equivalence checking, such as the BESS, which are lower-level than the modal μ -calculus and therefore are likely to require less computation effort. Encodings of branching and weak bisimulation using BESS of alternation depth two were proposed in [2]. These BESS contain two mutually recursive equation blocks, a maximal fixed point one encoding the bisimulation relation, and a minimal fixed point one encoding the τ -closures to be computed in the input LTSS. The local resolution algorithms underlying this class of BESS have a quadratic complexity w.r.t. the BES size, which makes them impractical for large LTSS; no implementation of this approach was reported as far as we know. Simpler encodings of weak equivalences using alternation-free BESS, obtained by leaving the computation of τ -closures (possibly enhanced with on-the-fly τ -confluence reduction [37]) outside the BES, proved to be practically effective [34].

An alternative approach consists in formulating equivalence checking by means of Horn clauses [28], which can be solved using classical HORNSAT resolution algorithms [15, 27]. We believe that BES encodings provide a more direct way of connecting on-the-fly equivalence checking to graph exploration algorithms. In fact, local BES resolution algorithms, such as the one presented in this paper, can also be used for solving HORNSAT efficiently, by applying the translation from Horn clauses to BESS proposed in [26].

Paper outline. Section 2 defines the class of BESS we use and illustrates the functioning of local resolution algorithms. Section 3 proposes new, faster BES encodings for branching and weak bisimulations. Section 4 describes our new local resolution algorithm, and Section 5 shows experimentally its performance when applied to equivalence checking. Finally, Section 6 gives some concluding remarks and directions for future work.

2 Background

A boolean equation system (BES) is a set of possibly recursive equations $B = \{X_i \stackrel{\sigma}{=} X_{i_1} \text{ op}_i \cdots \text{op}_i X_{i_{m_i}}\}_{1 \leq i \leq n}$, where $X_i \in \mathcal{X}$ are boolean variables, $\text{op}_i \in \{\vee, \wedge\}$ are disjunctive or conjunctive connectors, and $\sigma \in \{\mu, \nu\}$ is a minimal or maximal fixed point sign. An empty disjunction (resp. conjunction) is equivalent to the **false** (resp. **true**) constant. Each boolean variable occurring in the right-hand side of an equation must be defined by some equation of the BES. A variable X_i is said to be disjunctive (resp. conjunctive) iff $\text{op}_i = \vee$ (resp. \wedge). BESS of this kind are called *simple*, because each of their equations contains a single type of boolean connector (either \vee , or \wedge) in its right-hand side. Any BES containing arbitrary combinations of boolean connectors in the right-hand sides of its equations can be brought to the simple form with at most a linear blow-up in size, by introducing new equations to factor subformulas [3]. We focus our attention on BESS with a single equation block, since they are suitable for encoding equivalence checking problems [34]; more general BESS with multiple blocks are used for encoding model checking problems [13, 35]. In-depth presentations of the theory and applications of BESS can be found in [1, 29].

For each equation i of a BES, the evaluation of the formula in its right-hand side yields a boolean value defined as follows:

$$\llbracket X_{i_1} \text{ op}_i \cdots \text{op}_i X_{i_{m_i}} \rrbracket \delta = \delta(X_{i_1}) \text{ op}_i \cdots \text{op}_i \delta(X_{i_{m_i}}).$$

where the context $\delta : \mathcal{X} \rightarrow \text{Bool}$ is a partial function assigning boolean values to all variables occurring in the formula. The solution of a BES is a vector $\langle v_1, \dots, v_n \rangle \in \text{Bool}^n$ equal to the fixed point $\sigma \Phi$ of the functional $\Phi : \text{Bool}^n \rightarrow \text{Bool}^n$ associated to the BES:

$$\Phi(b_1, \dots, b_n) = \langle \llbracket X_{i_1} \text{ op}_i \cdots \text{op}_i X_{i_{m_i}} \rrbracket [b_1/X_1, \dots, b_n/X_n] \rangle_{1 \leq i \leq n}$$

where $[b_1/X_1, \dots, b_n/X_n]$ is the context assigning the boolean value b_i to variable X_i for $1 \leq i \leq n$. Since the boolean formulas in a BES do not contain negation operators, the functional Φ is monotonic, which ensures the existence of its minimal and maximal fixed points on $\langle \text{Bool}^n, \text{false}^n, \text{true}^n, \vee^n, \wedge^n \rangle$, the pointwise extension of the boolean lattice [25]. In the sequel, we consider only maximal fixed point BESS (i.e., with $\sigma = \nu$), which allow to encode equivalence checking.

The local resolution of a BES B , which corresponds to on-the-fly verification, amounts to computing the solution v_i of a particular variable X_i by solving as few equations of B as possible. Local resolution algorithms are easier

to devise and understand by representing BESS as *boolean graphs* [1]. Given a BES $B = \{X_i \stackrel{\sigma}{=} X_{i_1} op_i \cdots op_i X_{i_{m_i}}\}_{1 \leq i \leq n}$, its associated boolean graph $G = (V, E, L)$ is defined as follows: $V = \{X_1, \dots, X_n\}$ is the set of vertices (boolean variables), $E = \{(X_i, X_j) \mid 1 \leq i \leq n \wedge j \in \{i_1, \dots, i_{m_i}\}\}$ is the set of edges (dependencies between variables), and $L : V \rightarrow \{\vee, \wedge\}$, $L(X_i) = op_i$ for $1 \leq i \leq n$ is the labeling of vertices as disjunctive or conjunctive. The constant false (resp. true) is represented as a sink \vee -vertex (resp. \wedge -vertex). The local resolution of a vertex X_i consists in two activities performed simultaneously [1, 43, 34]: a forward exploration of the boolean graph along its edges, starting at X_i ; and a backward propagation of the *stable* variables found, i.e., whose boolean value has been determined. An example of local BES resolution is shown on Figure 1. The local resolution algorithm used is based on a depth-first search (DFS) of the boolean graph, starting at the variable of interest X_1 . The light grey area delimits the boolean subgraph explored during resolution. Black (resp. white) vertices correspond to variables whose solution is true (resp. false).

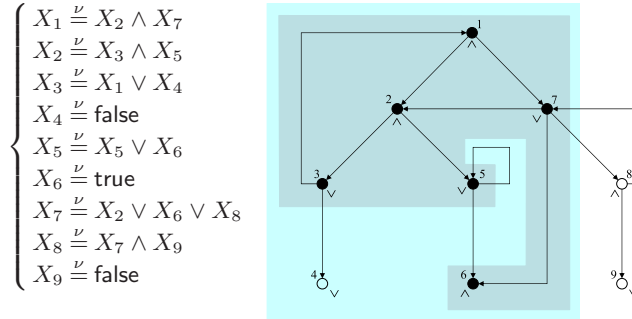


Fig. 1. Boolean graph-based local resolution of variable X_1

The solution of a BES can also be characterized by interpreting on its boolean graph the following *example formula* [31] written in modal μ -calculus:

$$\phi_{ex} = \nu X.(P_{\vee} \wedge \langle - \rangle X) \vee (P_{\wedge} \wedge [-] X)$$

where the atomic propositions P_{\vee} and P_{\wedge} denote \vee -vertices and \wedge -vertices, respectively. The solution v_i of a variable X_i is true iff the corresponding vertex X_i satisfies ϕ_{ex} in the boolean graph. A positive diagnostic (or *example*) for vertex X_i is a boolean subgraph that contains X_i and is a model for ϕ_{ex} . The dark grey area shown on Figure 1 delimits an example for X_1 , generated by traversing again the boolean subgraph explored during resolution [31].

3 Encoding Bisimulation Relations as BESs

Labeled transition systems (LTSS) are the semantic model underlying process algebras [6] and the related languages, such as LOTOS [24] and CHP [30]. An

LTS is a quadruple $M = \langle Q, A, T, q_0 \rangle$, where Q is the set of states, A is the set of actions (including the internal action τ), $T \subseteq Q \times A \times Q$ is the transition relation, and $q_0 \in Q$ is the initial state. A transition $\langle p, a, q \rangle \in T$ (also written $p \xrightarrow{a} q$) indicates that the system can move from state p to state q by performing action a . The notation is extended to transition sequences: $p \xrightarrow{l} q$ denotes the existence of a sequence going from p to q and whose concatenated labels form a word of the language $l \subseteq A^*$.

To compare the LTSS modeling the behaviour of concurrent systems, various equivalence relations were proposed (see [42] for a survey), among which *bisimulations* are most useful in practice due to their congruence properties w.r.t. the parallel composition operators of process algebras. We consider here three widely-used bisimulations: strong [38], branching [41], and weak [36], the last two being originally proposed as native equivalence relations for ACP [5] and CCS [36], respectively. Given two LTSS $M_i = \langle Q_i, A_i, T_i, q_{0i} \rangle$ with $i \in \{1, 2\}$, a bisimulation $\approx \subseteq Q_1 \times Q_2$ is a relation such that $p \approx q$ iff $\forall p \xrightarrow{a} p'. \exists q \xrightarrow{a} q'. p' \approx q'$ and $\forall q \xrightarrow{a} q'. \exists p \xrightarrow{a} p'. p' \approx q'$, where $p, p' \in Q_1$, $a \in A_1 \cup A_2$, and $q, q' \in Q_2$. Bisimulations are closed under union, and the strong bisimulation \approx_s is defined as the greatest one, i.e., the union of all bisimulations. M_1 is strongly equivalent to M_2 (notation $M_1 \approx_s M_2$) iff $q_{01} \approx_s q_{02}$.

A basic encoding of this mathematical definition as a maximal fixed point BES is shown in Table 1 (upper part, first row). The fact that $p \approx_s q$ is encoded as a boolean variable X_{pq} defined by an equation whose right-hand side boolean formula is directly derived from the two bisimulation conditions. The correctness of this encoding scheme relies on a bijection between the set of bisimulations and the set of fixed point solutions of the functional associated to the BES. To obtain a simple BES compliant with the definition given in Section 2, we introduce the new variables $Y_{p'qa}$ and $Z_{pq'a}$ such that each right-hand side formula contains a single type of boolean connector (second row). The BES for the strong preorder relation \preceq_s is obtained by keeping only the coloured parts of the equations. Checking the strong bisimilarity of M_1 and M_2 amounts to solving the variable $X_{q_{01}q_{02}}$ of this BES, which can be carried out using a local resolution algorithm. The evaluation of the boolean formulas in the right-hand sides of the equations defining X_{pq} , $Y_{p'qa}$, and $Z_{pq'a}$ triggers a forward exploration of transitions in M_1 and M_2 , which enables an incremental construction of both LTSS, and therefore an on-the-fly verification.

Similar encoding schemes hold for the branching (\approx_b) and weak (\approx_w) bisimulations, as shown in Table 1 (middle and lower parts, first rows). The important difference w.r.t. strong bisimulation is the presence of transitive reflexive closures over τ -transitions, which correspond to the abstraction of internal activity done by these two bisimulations. The simple BESS derived from these encodings by introducing new variables (similarly to strong bisimulation as shown above) were successfully used as basis for on-the-fly equivalence checking in conjunction with linear-time local BES resolution algorithms [34]. For LTSS with a high percentage of τ -transitions, the encodings of weak bisimulations lead to relatively small BESS; however, practical usage revealed that the computation of the

Table 1. Basic and full BES encodings of three widely-used bisimulations

Strong bisimulation		
$X_{pq} \stackrel{\nu}{=} \bigwedge_{p \xrightarrow{a} p'} \bigvee_{q \xrightarrow{a} q'} X_{p'q'} \wedge \bigwedge_{q \xrightarrow{a} q'} \bigvee_{p \xrightarrow{a} p'} X_{p'q'}$		
$X_{pq} \stackrel{\nu}{=} \bigwedge_{p \xrightarrow{a} p'} Y_{p'qa}$	$\wedge \bigwedge_{q \xrightarrow{a} q'} Z_{pq'a}$	$Y_{p'qa} \stackrel{\nu}{=} \bigvee_{q \xrightarrow{a} q'} X_{p'q'}$
$Z_{pq'a} \stackrel{\nu}{=} \bigvee_{p \xrightarrow{a} p'} X_{p'q'}$		
Branching bisimulation		
$X_{pq} \stackrel{\nu}{=} \bigwedge_{p \xrightarrow{a} p'} ((a = \tau \wedge X_{p'q}) \vee \bigvee_{q \xrightarrow{\tau^*} q'} \bigvee_{q \xrightarrow{a} q''} (X_{p'q'} \wedge X_{p''q''})) \wedge$ $\bigwedge_{q \xrightarrow{a} q'} ((a = \tau \wedge X_{pq'}) \vee \bigvee_{p \xrightarrow{\tau^*} p'} \bigvee_{p \xrightarrow{a} p''} (X_{p'q} \wedge X_{p''q'}))$		
$X_{pq} \stackrel{\nu}{=} \bigwedge_{p \xrightarrow{a} p'} Y_{pp'qa}$	$\wedge \bigwedge_{q \xrightarrow{a} q'} Z_{ppq'a}$	$Y_{pp'qa} \stackrel{\nu}{=} (a = \tau \wedge X_{p'q}) \vee U_{pp'qa}$
$Z_{ppq'a} \stackrel{\nu}{=} (a = \tau \wedge X_{pq'}) \vee V_{ppq'a}$	$U_{pp'qa} \stackrel{\nu}{=} \bigvee_{q \xrightarrow{a} q'} W_{pp'qq'}$	$\vee \bigvee_{q \xrightarrow{\tau} q'} U_{pp'q'a}$
$V_{ppq'a} \stackrel{\nu}{=} \bigvee_{p \xrightarrow{a} p'} W_{pp'qq'}$	$\vee \bigvee_{p \xrightarrow{\tau} p'} V_{p'qq'a}$	$W_{pp'qq'} \stackrel{\nu}{=} X_{pq} \wedge X_{p'q'}$
Weak bisimulation		
$X_{pq} \stackrel{\nu}{=} \bigwedge_{p \xrightarrow{a} p'} ((a = \tau \wedge \bigvee_{q \xrightarrow{\tau^*} q'} X_{p'q'}) \vee \bigvee_{q \xrightarrow{\tau^*} q'} \bigvee_{q \xrightarrow{a} q''} X_{p'q''}) \wedge$ $\bigwedge_{q \xrightarrow{a} q'} ((a = \tau \wedge \bigvee_{p \xrightarrow{\tau^*} p'} X_{p'q'}) \vee \bigvee_{p \xrightarrow{\tau^*} p'} \bigvee_{p \xrightarrow{a} p''} X_{p''q'})$		
$X_{pq} \stackrel{\nu}{=} \bigwedge_{p \xrightarrow{a} p'} Y_{p'qa} \wedge \bigwedge_{q \xrightarrow{a} q'} Z_{pq'a}$		
$Y_{p'qa} \stackrel{\nu}{=} (a = \tau \wedge U_{p'q}) \vee V_{p'qa}$	$U_{p'q} \stackrel{\nu}{=} X_{p'q} \vee \bigvee_{q \xrightarrow{\tau} q'} U_{p'q'}$	
$V_{p'qa} \stackrel{\nu}{=} \bigvee_{q \xrightarrow{a} q'} U_{p'q'} \vee \bigvee_{q \xrightarrow{\tau} q'} V_{p'q'a}$	$Z_{pq'a} \stackrel{\nu}{=} (a = \tau \wedge W_{pq'}) \vee T_{pq'a}$	
$W_{pq'} \stackrel{\nu}{=} X_{pq'} \vee \bigvee_{p \xrightarrow{\tau} p'} W_{p'q'}$	$T_{pq'a} \stackrel{\nu}{=} \bigvee_{p \xrightarrow{a} p'} W_{p'q'} \vee \bigvee_{p \xrightarrow{\tau} p'} T_{p'q'a}$	

various forms of τ -closures, even using optimized algorithms [33], is the most time-consuming part of the verification process.

An alternative solution for computing τ -closures would be to encode them directly using boolean equations, yielding the BESs shown on Table 1 (middle and lower part, second rows); however, this works only for LTSS without τ -cycles. To see this, consider the two LTSS $M_1 = \langle \{p_0, p_1\}, \{a, \tau\}, \{p_0 \xrightarrow{\tau} p_0, p_0 \xrightarrow{a} p_1\}, p_0 \rangle$ and $M_2 = \langle \{q_0, q_1\}, \{b, \tau\}, \{q_0 \xrightarrow{\tau} q_0, q_0 \xrightarrow{b} q_1\}, q_0 \rangle$, which are obviously not equivalent modulo any of the three bisimulations considered since they have different action sets. The comparison of M_1 and M_2 modulo weak bisimulation yields the BES below:

$$\begin{array}{lll}
X_{p_0q_0} \stackrel{\nu}{=} Y_{p_0q_0\tau} \wedge Y_{p_1q_0a} \wedge Z_{p_0q_0\tau} \wedge Z_{p_0q_1b} & & \\
Y_{p_0q_0\tau} \stackrel{\nu}{=} U_{p_0q_0} \vee V_{p_0q_0\tau} & U_{p_0q_0} \stackrel{\nu}{=} X_{p_0q_0} \vee U_{p_0q_0} & V_{p_0q_0\tau} \stackrel{\nu}{=} V_{p_0q_0\tau} \\
Z_{p_0q_0\tau} \stackrel{\nu}{=} W_{p_0q_0} \vee T_{p_0q_0\tau} & W_{p_0q_0} \stackrel{\nu}{=} X_{p_0q_0} \vee W_{p_0q_0} & T_{p_0q_0\tau} \stackrel{\nu}{=} T_{p_0q_0\tau} \\
Y_{p_1q_0a} \stackrel{\nu}{=} V_{p_1q_0a} & V_{p_1q_0a} \stackrel{\nu}{=} V_{p_1q_0a} & Z_{p_0q_1b} \stackrel{\nu}{=} T_{p_0q_1b} \quad T_{p_0q_1b} \stackrel{\nu}{=} T_{p_0q_1b}
\end{array}$$

We can easily compute the maximal fixed point solution of this BES by using the iterative characterization [25], which consists in initializing all variables to true and repeatedly evaluating the right-hand sides of equations until the values

of all variables become stable; the process converges in one iteration and all variables remain `true`, erroneously indicating that $M_1 \approx_w M_2$. The problem here is that τ -closures express the existence of *finite* τ -sequences in the LTSS, and hence they correspond to *minimal* fixed point computations, which cannot be done accurately by solving the equations of a *maximal* fixed point BES. On the other hand, if we eliminate the two τ -loops in M_1 and M_2 , the BES becomes:

$$X_{p_0q_0} \stackrel{\nu}{=} Y_{p_1q_0a} \wedge Z_{p_0q_1b} \quad Y_{p_1q_0a} \stackrel{\nu}{=} \text{false} \quad Z_{p_0q_1b} \stackrel{\nu}{=} \text{false}$$

and yields the correct solution `false` for the variable $X_{p_0q_0}$. This is a consequence of the fact that minimal and maximal fixed points have the same interpretation on acyclic models, as shown in [32] for modal μ -calculus formulas. Thus, if the LTSS being compared do not contain τ -cycles, the encoding of τ -closures using maximal fixed point equations is correct. The elimination of τ -cycles by collapsing their states (also called τ -compression), which preserves both branching and weak bisimulation, can be performed in linear-time during an on-the-fly LTS exploration [33], using an adaptation of Tarjan’s algorithm [39] for detecting strongly connected components (SCCs). To make the BES encodings in Table 1 correct, it is therefore sufficient to reduce both LTSS on-the-fly by applying τ -compression simultaneously with the local BES resolution.

The new BESS obtained in this way for branching and weak bisimulation have a size comparable with the BESS resulting from the previous encodings in which τ -closures were computed separately by specialized algorithms [33]; however, we observed experimentally that their resolution (using the same algorithms) is about one order of magnitude faster. This is due to the fact that intermediate results of τ -closure computations are stored as values of the boolean variables used to encode τ -closures (e.g., variables $U_{pp'qa}$ and $V_{ppq'a}$ of the BES encoding branching bisimulation), which are retrieved immediately if needed again during resolution; the only risk with this scheme was a too important increase of the number of such variables, which did not occur in practice. We also encoded as BESS, using similar schemes, the $\tau^*.a$ [17] and safety [7] equivalences, which are weaker than branching bisimulation and slightly less used in practice.

4 Local BES Resolution based on Suspend/Resume DFS

Several local BES resolution algorithms with a linear-time complexity are available [1, 43, 16, 34], typically based on DFS or breadth-first search (BFS) strategies for exploring the dependencies between boolean variables, i.e., the edges of the boolean graph. Here we aim to satisfy the following optimality criterion for local BES resolution algorithms, based on the notion of diagnostic [31]: the resolution must stop as soon as the boolean subgraph already explored contains exactly one diagnostic (example or counterexample) for the variable of interest. To our knowledge, all existing algorithms satisfy only a half of this criterion, i.e., they detect optimally either the presence of examples, or of counterexamples, but not of both of them. The LMC algorithm proposed in [16], based on a DFS traversal of the boolean graph with computation of SCCs, detects counterexamples

optimally and speeds up the search of examples (in maximal fixed point BESS) without attempting their optimal detection.

In the BESS produced from equivalence checking problems, false constants (sink \vee -vertices) denote couples of non equivalent states; if their backward propagation along edges in the boolean graph is done as soon as these vertices are encountered, it leads to an optimal detection of counterexamples, as in the A0 algorithm proposed in [34]. However, when the LTSSs being compared are equivalent, the variable of interest is true and the associated diagnostic is an *example*, which must be detected as soon as possible during resolution. Using the characterization of examples induced by the μ -calculus formula ϕ_{ex} given in Section 2, we can draw an alternative graph-based characterization: an example for vertex X is a subgraph containing X in which every \vee -vertex (resp. \wedge -vertex) must have exactly one successor (resp. all its successors) contained in the example. Each example can be split into maximal SCCs, which are connected acyclically; in the sequel, we denote them as *pseudo-SCCs*, since they are special cases of SCCs in the boolean graph (for instance, a trivial SCC containing a single sink \vee -vertex denotes a false constant, which is neither an example, nor a pseudo-SCC). These pseudo-SCCs are the smallest “building blocks” of the examples, and therefore their presence in the boolean subgraph already explored must be determined as soon as possible in order to achieve an optimal detection of examples.

To detect pseudo-SCCs, one can adapt Tarjan’s algorithm [39], which relies on a DFS traversal. The problem is that a classical DFS of the boolean graph does not allow the detection of pseudo-SCCs as soon as they occur, because Tarjan’s algorithm identifies SCCs only when their root vertex is popped from the DFS stack, meaning that the subgraph reachable from the root has been entirely explored; this subgraph may very well contain other pseudo-SCCs, which could make several examples to be contained in the boolean subgraph explored at the end of the resolution, i.e., when the variable of interest will be popped in turn from the DFS stack (if it evaluates to true, this variable is the root of the last pseudo-SCC identified). In order to detect the first pseudo-SCC encountered, it is necessary to *suspend* the DFS for each \vee -vertex when one of its successors was already visited; if this successor turns out to be false at some later stage (and thus not part of a pseudo-SCC), and the \vee -vertex is encountered again, it is necessary to *resume* the DFS by considering some other successor of the \vee -vertex that may belong to a pseudo-SCC. The exploration of \wedge -vertices is done as in the classical DFS, since for each \wedge -vertex, all its successors must be visited before attempting to detect a pseudo-SCC containing it.

The local resolution algorithm sr-DFS that we propose, based on this suspend/resume DFS, is illustrated below. Taking as input a boolean graph $G = (V, E, L)$ represented implicitly (i.e., by its successor function) and a variable of interest x , the algorithm performs iteratively a forward search of G starting at vertex x . Visited vertices are stored in a set $A \subseteq V$. The DFS stack is stored in a variable *dfs* and the stack used for detecting pseudo-SCCs is stored in a variable *scc*. The variable *count* keeps a global counter allowing the assignment

of unique DFS numbers to visited vertices. To each vertex v are associated the following fields:

- a counter $c(v)$ which counts the number of remaining successors of v to visit in order to stabilize v ;
- a number $p(v)$ recording the index of the next successor of v to be visited (the successors in $E(v)$ are supposed to be indexed from 0 to $|E(v)| - 1$);
- a number $n(v)$ representing the DFS number of v ;
- a number $l(v)$ representing the “lowlink” number [39] of v , used to detect if a vertex is the root of a pseudo-SCC;
- a set $d(v)$ containing the vertices that currently depend upon v ;
- a boolean $on_scc(v)$ which is **true** if v is on the *scc* stack and false otherwise;
- a boolean $stop(v)$ which is **true** if the DFS must be suspended for v (i.e., v is a \vee -vertex and one of its successors has been visited);
- a boolean $stable(v)$, which is **true** if v is stable;
- a boolean $value(v)$, which represents the value of v (this field is of interest only if v is stable).

At each iteration of the main while-loop (lines 20–122), the vertex y at the top of the *dfs* stack is explored. If y is stable, or the DFS must be suspended for y (that is, y is a \vee -vertex and one of its successors has already been visited), the value of y is back-propagated along its predecessors d (lines 30–62). For each vertex w which is not stabilized by the back-propagation, the algorithm must keep on visiting its successors, if w will be visited again during the DFS (the variable $stop(w)$ becomes false). Due to the suspend/resume principle, this propagation phase may influence the contents of the pseudo-SCC currently stored on the *scc* stack. Indeed, each \vee -vertex which is visited during the propagation phase is stored on the *scc* stack. The definition of pseudo-SCCs requires that each \vee -vertex must have exactly one successor contained in its pseudo-SCC. But, as the vertex was not stabilized by the value of the successor which was propagated, it does not meet any more the definition of the pseudo-SCC. Since the *scc* does not contain a pseudo-SCC anymore, it must be cleared. A variable called *purge* is used for this purpose and becomes true when the *scc* stack must be cleared (two variables *min* and *max* are used to determine if *scc* must be cleared: *min* represents the least DFS number among all the DFS numbers of vertices stabilized during the propagation phase and *max* represents the greatest DFS number among all \vee -vertices towards which a false value has been propagated).

If the vertex y at the top of the *dfs* stack is unstable or that the exploration must continue for this vertex, its next unexplored successor $z = E(y)_{p(y)}$ is visited. Before that, the *scc* stack is cleared if needed (i.e., if a back-propagation of a false value took place at some previous iteration of the main while-loop). If z is a new vertex (lines 96-107), it is pushed on the *dfs* stack. If z is an already explored vertex, two cases may appear. Either z is on the *scc* stack (lines 87-90) and therefore its lowlink number must be updated, or it is not on the stack (lines 91-95), and therefore it must be explored as if it was a new vertex (i.e., z was popped from the *scc* stack after a propagation phase which induced a clearing of this stack). Finally, if y is unstable and all its successors have been visited,

Algorithm 1 Local BES resolution based on suspend/resume DFS

```

1: function sr-DFS ( $x, (V, E, L)$ ) : Bool is
2: var  $A, B : 2^V ; d : V \rightarrow 2^V ;$ 
3:    $u, w, y, z : V ; dfs, scc : V^* ;$ 
4:    $c, p, n, l : V \rightarrow \mathbf{Nat} ;$ 
5:    $stop, stable : V \rightarrow \mathbf{Bool} ;$ 
6:    $value, on\_scc : V \rightarrow \mathbf{Bool} ;$ 
7:    $count, max, min : \mathbf{Nat} ;$ 
8:    $purge : \mathbf{Bool} ;$ 
9: if  $L(x) = \wedge$  then
10:   $c(x) := |E(x)|$ 
11: else
12:   $c(x) := 1$ 
13: end if
14:  $p(x) := 0 ; stable(x) := \text{false} ;$ 
15:  $d(x) := \emptyset ; value(x) := \text{false} ;$ 
16:  $A := \{x\} ; count := 0 ;$ 
17:  $dfs := push(x, nil) ;$ 
18:  $scc := push(x, nil) ;$ 
19:  $on\_scc(x) := \text{true} ;$ 
20: while  $dfs \neq nil$  do
21:   $y := top(dfs) ;$ 
22:   $n(y) := count ;$ 
23:   $l(y) := n(y) ;$ 
24:   $count := count + 1 ;$ 
25:   $max := 0 ;$ 
26:   $min := n(y) ;$ 
27:  if  $stable(y) \vee stop(y)$  then
28:   if  $d(y) \neq \emptyset$  then
29:     $B := \{y\} ;$ 
30:    while  $B \neq \emptyset$  do
31:     let  $u \in B ;$ 
32:      $B := B \setminus \{u\} ;$ 
33:     for all  $w \in d(u)$  do
34:      if  $\neg stable(w)$  then
35:       if  $((L(w) = \vee) \wedge value(u)) \vee$ 
36:          $((L(w) = \wedge) \wedge \neg value(u))$ 
37:         then
38:           $c(w) := 0$ 
39:        else
40:          $c(w) := c(w) - 1$ 
41:        end if
42:        if  $c(w) = 0$  then
43:          $stable(w) := \text{true} ;$ 
44:          $value(w) := value(u) ;$ 
45:          $B := B \cup \{w\} ;$ 
46:         if  $n(w) < min$  then
47:           $min := n(w)$ 
48:        end if
49:        else
50:          $stop(w) := \text{false} ;$ 
51:         if  $L(w) = \wedge$  then
52:           $c(w) := 1 ;$ 
53:           $p(w) := 0$ 
54:        else
55:          $E(w) := E(w) \setminus \{u\} ;$ 
56:         if  $n(w) > max$  then
57:           $max := n(w)$ 
58:        end if
59:        end if
60:      end if
61:    end for
62:  end while
63:   $dfs := pop(dfs) ;$ 
64:  if  $dfs \neq nil$  then
65:    $l(top(dfs)) := \min(l(top(dfs)), l(y))$ 
66:  end if
67: end if
68: else
69:  if  $purge$  then
70:   while  $top(scc) \neq top(dfs)$  do
71:     $scc := pop(scc)$ 
72:   end while
73:   $purge := \text{false}$ 
74: end if
75: if  $p(y) < |E(y)|$  then
76:  if  $L(y) = \vee$  then
77:    $stop(y) := \text{true}$ 
78:  end if
79:   $z := (E(y))_{p(y)} ;$ 
80:   $p(y) := p(y) + 1 ;$ 
81:   $d(z) := d(z) \cup \{y\} ;$ 
82:  if  $z \in A$  then
83:   if  $on\_scc(z)$  then
84:    if  $n(z) < n(y)$  then
85:      $l(y) := \min(n(z), n(y))$ 
86:    end if
87:   else
88:     $dfs := push(z, dfs) ;$ 
89:     $scc := push(z, scc) ;$ 
90:     $on\_scc(z) := \text{true}$ 
91:   end if
92:  else
93:   if  $L(z) = \wedge$  then
94:     $c(z) := |E(z)|$ 
95:   else
96:     $c(z) := 1$ 
97:   end if
98:    $p(z) := 0 ;$ 
99:    $A := A \cup \{z\} ;$ 
100:   $dfs := push(z, dfs) ;$ 
101:   $scc := push(z, scc) ;$ 
102:   $on\_scc(z) := \text{true}$ 
103: end if
104: else
105:  if  $(l(y) = n(y)) \wedge (top(scc) \neq y)$ 
106:  then
107:   repeat
108:     $z := top(scc) ;$ 
109:     $c(z) := 0 ;$ 
110:     $scc := pop(scc)$ 
111:   until  $top(scc) = y$ 
112:  end if
113:   $dfs := pop(dfs) ;$ 
114:  if  $dfs \neq nil$  then
115:    $l(top(dfs)) := \min(l(top(dfs)), l(y))$ 
116:  end if
117: end if
118: end while
119: return  $value(x)$ 

```

the algorithm watches if y is the root of a pseudo-SCC (lines 108–120). If this is the case, the *scc* stack is cleared from its top until y and each vertex of the pseudo-SCC is stabilized. Then, y is popped from the *dfs* stack. Finally, after termination of the main while-loop, the value computed for x is returned.

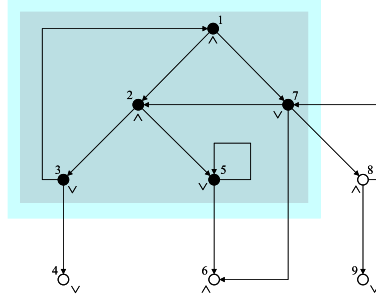


Fig. 2. Local resolution of variable X_1 using the sr-DFS algorithm

The execution of algorithm sr-DFS on the boolean graph considered in Section 2 is illustrated on Figure 2. We observe on this example an optimal behaviour of sr-DFS: due to the suspension of the DFS for the \vee -vertices X_3 , X_5 , and X_7 when one of their successors was visited, the subgraph explored by the algorithm (light grey area) coincides with the example found for vertex X_1 (dark grey area), made of the pseudo-SCCs $\{X_1, X_2, X_3, X_7\}$ and $\{X_5\}$. The resolution previously shown on Figure 1 was done using the algorithm A0 [34], which is based on a classical DFS without computation of SCCs, and therefore explores a larger subgraph than sr-DFS in order to find another, larger example for X_1 .

Complexity. For boolean graphs $G = (V, E, L)$ without sink \vee -vertices (i.e., for maximal fixed point BESS without **false** constants in the right-hand sides of their equations), the sr-DFS algorithm has a linear-time complexity $O(|V| + |E|)$. The presence of **false** constants could trigger the reexploration of some vertices (those present on the portions of the *scc* stack that were cleared after back-propagation of **false** constants), increasing the complexity of the algorithm towards quadratic-time $O((|V| + |E|)^2)$, which is the theoretical worst-case. This is the price to pay for achieving an optimal detection of examples and counterexamples in the boolean graph. However, the behaviour of the sr-DFS algorithm that we observed in practice for equivalence checking (by measuring the number of variables explored and reexplored) shows that its complexity is close to linear-time.

5 Implementation and Experiments

The `CÆSAR_SOLVE` [34] library of CADP [21] provides a generic implementation of several local BES resolution algorithms. The library was developed using the

Table 2. Algorithms of `CÆSAR_SOLVE` and their application to equivalence checking

Alg.	BES type	Strategy	Time	Memory	Condition
A0	general	Dfs	$O(V + E)$	$O(V + E)$	nondeterministic LTSS
A1		Bfs			
A2	acyclic	Dfs		$O(V)$	one LTS acyclic
A3	disjunctive				—
A4	conjunctive				one LTS deterministic and τ -free
A5	general	Bfs		$O(V + E)$	nondeterministic LTSS
A6	disjunctive				—
A7	conjunctive		one LTS deterministic and τ -free		

`OPEN/CÆSAR` [20] generic environment for LTS manipulation, which offers many primitives dedicated to graph exploration (stacks, hash tables, edge lists, etc.). BESs are handled by `CÆSAR_SOLVE` by means of their corresponding boolean graphs, represented implicitly in a way similar to LTSS in `OPEN/CÆSAR`. This representation is application-independent, allowing to employ the resolution algorithms as computing engines for several on-the-fly verification tools of CADP: the model checker `EVALUATOR` [35], the equivalence checker `BISIMULATOR` [4, 34], and the `REDUCTOR` tool for LTS generation equipped with partial order reductions.

Table 2 summarizes the local resolution algorithms currently available in `CÆSAR_SOLVE` and their application for equivalence checking within `BISIMULATOR`. All algorithms have a linear complexity w.r.t. the size of boolean graphs (number of vertices and edges). Algorithms A0, A1, and A5 can solve general BESs (without constraints on the structure of equations), A1 being BFS-based and thus able to produce small-depth diagnostics. When one LTS is deterministic (for strong equivalence) and τ -free (for weak equivalences), the resulting BES is conjunctive and can be solved using the memory-efficient algorithm A4 [34], which stores only the vertices of the boolean graph (and not its edges), i.e., only the states of the LTSS (and not their transitions). Also, when one LTS is acyclic, the resulting BES is also acyclic and can be solved using the memory-efficient algorithm A2. The BFS-based algorithm A7, recently added to the library, can be applied to conjunctive BESs and combines the advantages of algorithms A1 (small-depth diagnostics) and A4 (low memory consumption) when one LTS is deterministic and τ -free.

The version `BISIMULATOR 2.0` includes the new BES encodings of weak equivalences defined in Section 3 and the new resolution algorithm `sr-Dfs` given in Section 4, which was recently added to `CÆSAR_SOLVE` with the number A8. In the sequel, we present various performance measures showing the effect of these two enhancements. The LTSS considered were generated from the demo examples of CADP (specifications of communication protocols and asynchronous circuits) or taken from the VLTS benchmark suite [44].

New encodings of weak equivalences. The new BES encodings of weak equivalences that we proposed in Section 3 compute τ -closures by means of BES equations instead of relying on external, dedicated graph algorithms as the previous encodings used in BISIMULATOR 1.0. Figure 3(a)–(b) compares the performance of the two encodings for branching bisimulation as regards the size of the underlying BESS and their resolution time using algorithm A0. As expected, the BESS produced by the new encoding are larger (more variables but less operators) because intermediate results of τ -closure computations are stored as boolean variables, but they are solved faster due to the simpler structure of boolean equations. Of course, what matters from the end-user point of view is the overall performance of using sr-DFS in conjunction with the new BES encoding; this is illustrated below.

Resolution using the sr-DFS algorithm. The series of experiments shown in Figure 3(c)–(f) compare the behaviour of BISIMULATOR 1.0 (algorithm A0 and previous BES encoding) w.r.t. version 2.0 (algorithm sr-DFS and new BES encoding) for branching bisimulation. To improve readability, we separated the LTSS in two groups according to their number of transitions. When applying version 2.0, we observed reductions of both the number of vertices visited and edges explored, which determine the memory consumption and the execution time, respectively. These reductions become more important as the LTS size increases, as indicated by curves (e) and (f); in particular, the number of transitions traversed can decrease by a factor 8. It is worth noticing that some of the LTSS compared were not equivalent (e.g., certain erroneous variants of a leader election protocol examined in [22]), showing that version 2.0 of the tool exhibits a good behaviour also for counterexample detection. These experimental results indicate that the increase in BES size induced by the new encoding of weak equivalences is compensated by the reduction achieved using sr-DFS, leading to an overall improvement of the on-the-fly verification procedure.

Further optimizations. Algorithm sr-DFS always visits a number of vertices (boolean variables) smaller or equal to that of algorithm A0 (assuming that the successors of every vertex in the boolean graph are visited in the same order). On the other hand, the number of edges (boolean operators) traversed by sr-DFS may increase w.r.t. those traversed by A0 because of possible reexplorations of certain vertices. We observed this phenomenon in some cases, e.g., when comparing Philips’ Bounded Retransmission Protocol (BRP) with its service modulo branching bisimulation. To remedy this situation, we generalized sr-DFS in order to explore k successors of each \vee -vertex before suspending the DFS, and to wait until all these successors become false before resuming the DFS from that vertex (the sr-DFS algorithm given in Section 4 corresponds to $k = 1$). We studied how the number of edges traversed by sr-DFS varies with the threshold k , and found out that the optimal value of k is not necessarily 1, as shown on Figure 3(g) for an instance of BRP with messages of length 8 and 6 retransmissions. Here the minimal number of edges explored is obtained for $k = 3$, a further increase of k leading gradually to the number of edges explored

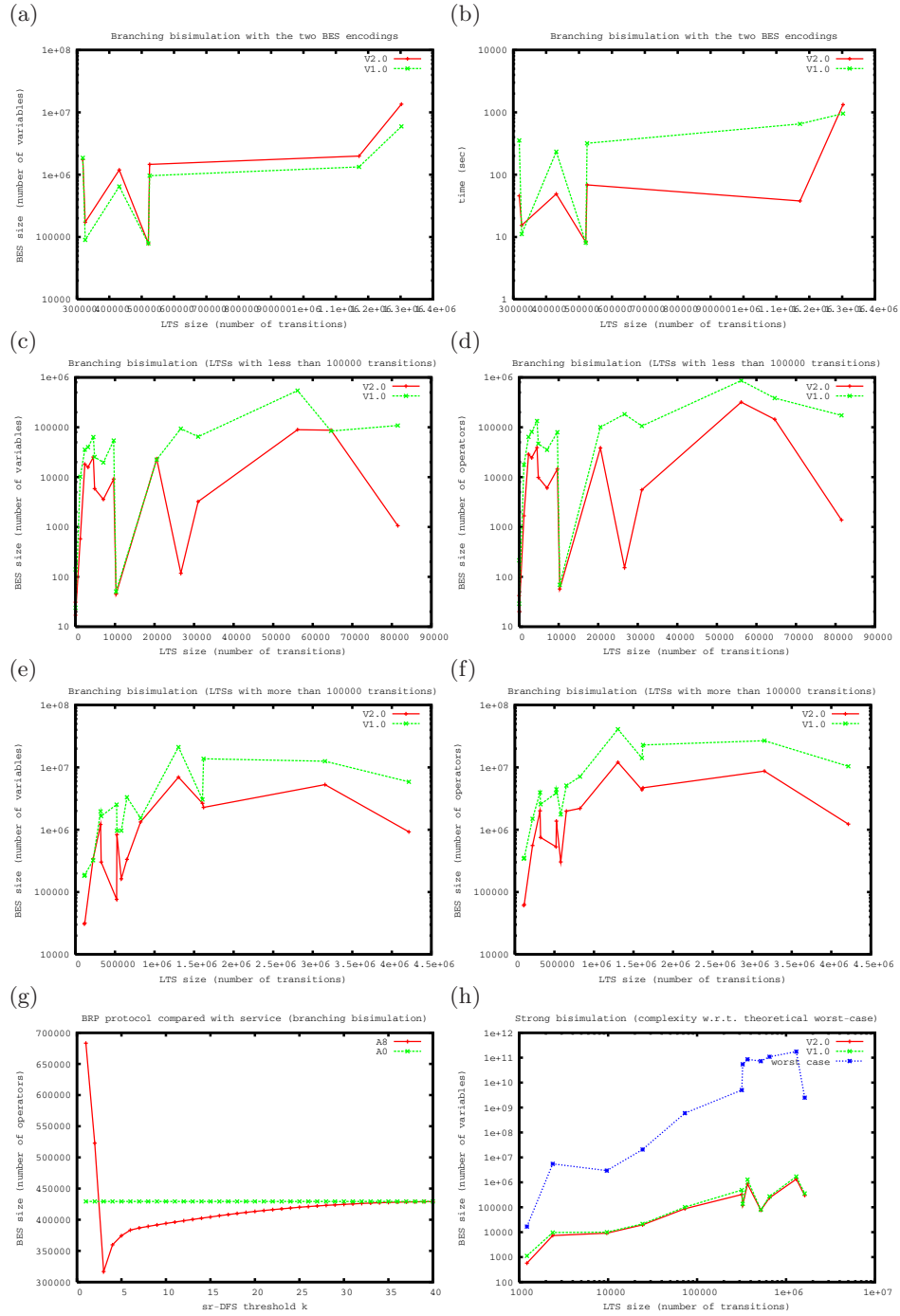


Fig. 3. Performance of equivalence checking using BISIMULATOR 1.0 and 2.0

by A0 (which can be seen as a variant of sr-DFS with $k = \infty$). Finding the optimal value of k would require a detailed knowledge of the LTSS being compared, which is not available in the setting of on-the-fly verification. We are currently experimenting adaptive schemes for increasing k dynamically according to the number of vertices reexplored so far, which attempt to trade off the number of visited vertices and the amount of reexplorations.

Complexity w.r.t. theoretical worst-case. As pointed out in [17], the on-the-fly comparison of two nondeterministic LTSS $M_1 = \langle Q_1, A_1, T_1, q_{01} \rangle$ and $M_2 = \langle Q_2, A_2, T_2, q_{02} \rangle$ has a worst-case complexity $O((|Q_1| \cdot |T_2|) + (|Q_2| \cdot |T_1|))$. Considering the BES formulation of the problem, this complexity can be estimated in terms of BES size: the BESS given in Table 1 have a maximum number of boolean variables proportional to the size of the synchronous product between the two LTSS. However in practice, the BESS produced from equivalence checking have a much smaller size (several orders of magnitude) than the theoretical worst-case, as it is illustrated in Figure 3(h) for strong bisimulation. This also holds for weak equivalences, in particular for branching bisimulation.

6 Conclusion and Future Work

Building efficient software tools for on-the-fly equivalence checking between LTSS is a difficult and time-consuming task. The usage of intermediate formalisms, such as BESS, allows one to separate the concerns of phrasing the verification problem and of solving it, leading to highly modular verification tools [35, 4]. The two optimizations we proposed, namely the new encodings of weak equivalences by applying τ -compression on the input LTSS and computing τ -closures using boolean equations (Section 3) and the new sr-DFS local BES resolution algorithm (Section 4) significantly increased the performance of on-the-fly equivalence checking w.r.t. existing approaches.

These optimizations are at the core of the new version 2.0 of the BISIMULATOR equivalence checker [34] of the CADP toolbox [21]. The sr-DFS algorithm was integrated to the generic CÆSAR_SOLVE library [34] for on-the-fly BES resolution, which is part of the generic OPEN/CÆSAR environment [20] for LTS manipulation. Local BES resolution proved to be a suitable alternative way for computing τ -closures on LTSS produced from protocols and distributed systems, competing favourably with general transitive closure algorithms. The sr-DFS algorithm is able to detect optimally the presence of both examples and counterexamples in the boolean graph, and appears to be quite effective for comparing LTSS modulo weak equivalences.

We plan to continue our work along two directions. First, the range of equivalences and preorders already available in BISIMULATOR 2.0 (strong, branching, weak, $\tau^*.a$, safety, trace, and weak trace) could be extended by devising BES encodings for other weak equivalences, such as CFFD [40] and testing equivalence [10], following the scheme in Section 3. Next, we will pursue experimenting the sr-DFS algorithm and study its applicability for solving BESS coming from

other verification problems, such as on-the-fly LTS reduction modulo partial order relations (e.g., τ -confluence, τ -inertness, etc.) as formulated in [37].

References

1. H. R. Andersen. Model checking and boolean graphs. *TCS*, 126(1):3–30, 1994.
2. H. R. Andersen and B. Vergauwen. Efficient checking of behavioural relations and modal assertions using fixed-point inversion. *Proc. of CAV'95*, LNCS vol. 939, pp. 142–154, 1995.
3. A. Arnold and P. Crubillé. A linear algorithm to solve fixed-point equations on transition systems. *IPL*, 29(1):57–66, 1988.
4. D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. Bisimulator: A modular tool for on-the-fly equivalence checking. *Proc. of TACAS'2005*, LNCS vol. 3440, pp. 581–585, 2005.
5. J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60, 1984.
6. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.
7. A. Bouajjani, J-C. Fernandez, S. Graf, C. Rodríguez, and J. Sifakis. Safety for branching time semantics. In *Proc. of ICALP'91*, 1991.
8. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *JACM*, 31(3):560–599, 1984.
9. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based verification tool for finite state systems. *ACM TOPLAS*, 15(1):36–72, 1993.
10. R. Cleaveland and M. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5(1):1–20, 1993.
11. R. Cleaveland and O. Sokolsky. Equivalence and Preorder Checking for Finite-State Systems. In *Handbook of Process Algebra*, chapter 6, pp. 391–424, 2001.
12. R. Cleaveland and B. Steffen. Computing behavioural relations, logically. *Proc. of ICALP'91*, LNCS vol. 510, pp. 127–138, 1991.
13. R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *FMSD*, 2(2):121–147, 1993.
14. A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *TCS*, 311(1–3):221–256, 2004.
15. W.F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *J. of Logic Programming*, 3:267–284, 1984.
16. X. Du, S. A. Smolka, and R. Cleaveland. Local model checking and protocol analysis. *Springer Int. Journal on Software Tools for Technology Transfer (STTT)*, 2(3):219–241, 1999.
17. J-C. Fernandez and L. Mounier. Verifying bisimulations “on the fly”. *Proc. of FORTE'90*, 1990.
18. J-C. Fernandez and L. Mounier. A tool set for deciding behavioral equivalences. *Proc. of CONCUR'91*, 1991.
19. K. Fisler and M. Y. Vardi. Bisimulation minimization and symbolic model checking. *FMSD*, 21(1):39–78, 2002.
20. H. Garavel. Open/cæsar: An open software architecture for verification, simulation, and testing. *Proc. of TACAS'98*, LNCS vol. 1384, pp. 68–84, 1998.

21. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. *Proc. of CAV'2007*, LNCS vol. 4590, pp. 158–163, 2007.
22. H. Garavel and L. Mounier. Specification and verification of various distributed leader election algorithms for unidirectional ring networks. *SCP*, 29(1–2):171–197, 1997.
23. A. Ingólfssdóttir and B. Steffen. Characteristic formulae for processes with divergence. *Inf. and Computation*, 110(1):149–163, 1994.
24. ISO/IEC. Lotos — a formal description technique based on the temporal ordering of observational behaviour. Int. Standard 8807, ISO, Genève, 1989.
25. S. C. Kleene. *Introduction to Metamathematics*. North-Holland, 1952.
26. X. Liu and S. A. Smolka. Simple linear-time algorithms for minimal fixed points. *Proc. of ICALP'98*, LNCS vol. 1443, pp. 53–66, 1998.
27. G. Ausiello and G. F. Italiano. On-line algorithms for polynomially solvable satisfiability problems. *J. Log. Program.*, 10(1/2/3&4):69–90, 1991.
28. S. K. Shukla and H. B. Hunt III and D. J. Rosenkrantz. Hornsat, model checking, verification and games. *Proc. of CAV'96*, LNCS vol. 1102, pp. 99–110, 1996.
29. A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. VERSAL 8, Bertz Verlag, Berlin, 1997.
30. A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.
31. R. Mateescu. Efficient diagnostic generation for boolean equation systems. *Proc. of TACAS'2000*, LNCS vol. 1785, pp. 251–265, 2000.
32. R. Mateescu. Local model-checking of modal mu-calculus on acyclic labeled transition systems. *Proc. of TACAS'2002*, LNCS vol. 2280, pp. 281–295, 2002.
33. R. Mateescu. On-the-fly state space reductions for weak equivalences. *Proc. of FMICS'05*, pp. 80–89. ACM Press, 2005.
34. R. Mateescu. Caesar_solve: A generic library for on-the-fly resolution of alternation-free boolean equation systems. *Springer Int. Journal on Software Tools for Technology Transfer (STTT)*, 8(1):37–56, 2006.
35. R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *SCP*, 46(3):255–281, 2003.
36. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
37. G. Pace, F. Lang, and R. Mateescu. Calculating τ -confluence compositionally. *Proc. of CAV'2003*, LNCS vol. 2725, pp. 446–459, 2003.
38. D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Th. Comp. Sci.*, LNCS vol. 104, pp. 167–183, 1981.
39. R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. of Computing*, 1(2):146–160, 1972.
40. A. Valmari and M. Tienari. Compositional failure-based semantics models for basic lotos. *Formal Aspects of Computing*, 7(4):440–468, 1995.
41. R. J. van Glabbeek and W. P. Weijland. Branching-time and abstraction in bisimulation semantics. Proc. IFIP 11th World Computer Congress, 1989.
42. R. van Glabbeek. The Linear Time — Branching Time Spectrum I. The Semantics of Concrete, Sequential Processes. In *Handbook of Process Algebra*, chapter 1, pp. 3–100, 2001.
43. B. Vergauwen and J. Lewi. Efficient local correctness checking for single and alternating boolean equation systems. *Proc. of ICALP'94*, LNCS vol. 820, pp. 304–315, 1994.
44. VASY. The VLTS benchmark suite. <http://www.inrialpes.fr/vasy/cadp/resources/benchmark.html>.