Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 128 (2005) 47–62

Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Distributed On-the-Fly Equivalence Checking

Christophe Joubert and Radu Mateescu[1]

INRIA Rhône-Alpes / VASY
655, av. de l'Europe, F-38330 Montbonnot St Martin, France

Abstract

On-the-fly equivalence checking consists in comparing two Labeled Transition Systems (LTSs) modulo a given equivalence relation by exploring them in a demand-driven way. Since it avoids the explicit construction of LTSs, this method is able to detect errors even in systems that are too large to fit in the memory of a computer. In this paper, we aim at further improving the performance of on-the-fly equivalence checking using several machines connected by a network. We propose DSOLVE, a new algorithm for distributed on-the-fly resolution of Boolean Equation Systems (BESs), which enables equivalence checking modulo various relations characterized in terms of BESs. DSOLVE serves as verification engine for the distributed version of BISIMULATOR, an on-the-fly equivalence checker developed within the CADP verification toolbox using the OPEN/CÆSAR environment. Our experimental measures show quasi-linear speedups and a good scalability of the distributed version of BISIMULATOR w.r.t. its sequential version.

Keywords: Bisimulation, boolean equation systems, labelled transition systems, distributed equivalence checking.

1 Introduction

Equivalence checking is a verification technique that consists in comparing the description of a system behavior (e.g., a *protocol*) with the description of its desired behavior (e.g., a *service*) modulo a suitable equivalence relation. Numerous equivalence relations (strong [24], branching [25], observational [22], $\tau^*.a$ [11], safety [8], etc.) were defined on Labeled Transition Systems (LTSs), which are the natural models for action-based description languages such as process algebras. There are basically two approaches for checking the equivalence of finite LTSs: *globally*, which requires the construction of the two LTSs

[1] {Christophe.Joubert, Radu.Mateescu}@inrialpes.fr

doi:10.1016/j.entcs.2004.10.018

before verification, and *locally* (or *on-the-fly*), which allows the Ltss to be constructed incrementally during verification. The on-the-fly approach has the ability to detect errors even when the Ltss are too large to be constructed explicitly, and therefore is more suitable for analyzing large systems.

During the past two decades, many sequential algorithms for global equivalence checking were designed and implemented in verification tools (see [10] for a survey). Most of these algorithms rely on partition refinement: starting with a state partition containing a single equivalence class, they iteratively refine it (by splitting classes which contain non equivalent states) until no further distinction between classes is possible according to a given equivalence relation. Recently, distributed global equivalence checking algorithms were proposed [5,6], showing effective behavior on medium and large-sized Ltss (dozens of millions of states and transitions). However, relatively little research effort was devoted to on-the-fly equivalence checking algorithms.

The first algorithms proposed for on-the-fly equivalence checking [11] and preorder checking [10] were based on the following principle: a forward, simultaneous exploration of the two Ltss is performed starting from their initial states, until either some execution pattern showing non equivalence (counterexample) is encountered, or the two Ltss have been entirely explored. Another approach for on-the-fly equivalence checking is based upon the characterizations of equivalence relations in terms of Boolean Equation Systems (Bess) [9,3], which allow to use efficient algorithms for on-the-fly Bes resolution [20]. In this way, the encoding of an equivalence relation and the Bes resolution algorithm are clearly separated, allowing them to be implemented and optimized independently. We followed this latter approach for developing the on-the-fly equivalence checker Bisimulator, which uses the generic Cæsar_Solve [20] Bes resolution library, built using the Open/Cæsar environment for on-the-fly Lts exploration [12] of the Cadp verification toolbox [13].

In this paper, we present the distributed version of Bisimulator, which has been obtained by devising DSolve, an algorithm for distributed on-the-fly Bes resolution. As far as we know, this is the first attempt to develop a distributed on-the-fly equivalence checker. DSolve is similar in spirit with the distributed model-checking algorithm proposed (in the setting of game graphs) in [7]: it performs a distributed forward traversal of the dependency graph of the Bes, combined with a backward propagation of stable variables (i.e., whose final value has been computed). It was implemented to run on commonly available loosely-coupled architectures such as networks of workstations (Nows) and clusters of Pcs. Our experiments show quasi-linear speedups of DSolve and a good scalability of its performance w.r.t. the problem size. DSolve was integrated to the generic Cæsar_Solve library and therefore allows to immediately obtain distributed versions of any other applications built using CÆSAR_SOLVE, such as alternation-free μ-calculus model-checking [20] and τ-confluence reduction [23].

The remainder of the paper is organized as follows. Section 2 recalls the definitions of BESS and the encodings of five widely-used equivalence relations in terms of BESS. Section 3 describes in detail the DSOLVE algorithm and Section 4 shows experimental data comparing the performance of the distributed and sequential versions of BISIMULATOR. Finally, Section 5 gives some concluding remarks and directions for future work.

2 Equivalence relations and boolean equation systems

An LTS is a quadruple $M = (Q, A, T, q_0)$, where: Q is the set of states, A is the set of actions ($A_\tau = A \cup \{\tau\}$ also contains 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 $q_1 \xrightarrow{a} q_2 \in T$ means that the system can move from state q_1 to state q_2 by executing action a. Given a language $l \subseteq {A_\tau}^*$, $q_1 \xrightarrow{l} q_2$ means that from q_1 to q_2 there is a sequence of transitions whose concatenated actions form a word of l. In the sequel, we consider two LTSS $M_i = (Q_i, A, T_i, q_{0i})$, $i \in \{1, 2\}$.

A BES is a set of equations $B = \{X_i = X_{i1}\ op_i \cdots op_i\ X_{ik_i}\}_{1 \le i \le n}$, where X_i are boolean variables and $op_i \in \{\vee, \wedge\}$. For efficiency of resolution, we consider *simple* BESS [4], whose right-hand sides of equations are pure disjunctive or conjunctive formulas (boolean constants F and T are encoded as empty disjunctions and conjunctions, respectively). The semantics of a BES is given by the maximal fixed point of the associated vectorial functional $\Phi : \mathbb{B}^n \to \mathbb{B}^n$, $\Phi(b_1, ..., b_n) = ([\![X_{i1}\ op_i \cdots op_i\ X_{ik_i}]\!][b_1/X_1, \ldots, b_n/X_n])_{1 \le i \le n}$, where $[\![\varphi]\!]\delta$ is the interpretation of a boolean formula φ in a context δ that assigns boolean values to variables. The theory underlying BESS is extensively developed in [18].

Various equivalence relations between LTSS were characterized in terms of BESS [9,3]. The table below shows the encodings of five widely-used equivalences: strong [24], branching [25], observational [22], $\tau^*.a$ [11], and safety [8]. Each relation is represented as a BES whose variables $X_{p,q}$ indicate whether the states $p \in Q_1$ and $q \in Q_2$ are equivalent or not ($a \in A$ and $b \in A_\tau$). For each equivalence, the corresponding preorder (in *grey*) is obtained by deleting either the 2nd conjunct (for strong, $\tau^*.a$, safety, and branching), or the 3rd and 4th conjuncts (for observational) in the right-hand sides of the equations.

RELATION	ENCODING
Strong	$\left\{ X_{p,q} = \left(\bigwedge_{p \xrightarrow{b} p'} \bigvee_{q \xrightarrow{b} q'} X_{p',q'}\right) \wedge \left(\bigwedge_{q \xrightarrow{b} q'} \bigvee_{p \xrightarrow{b} p'} X_{p',q'}\right) \right\}$
Branching	$\left\{ \begin{array}{rl} X_{p,q} = & \bigwedge_{p \xrightarrow{b} p'} \left((b = \tau \wedge X_{p',q}) \vee \bigvee_{q \xrightarrow{\tau^*} q' \xrightarrow{b} q''} (X_{p,q'} \wedge X_{p',q''})\right) \wedge \\ & \bigwedge_{q \xrightarrow{b} q'} \left((b = \tau \wedge X_{p,q'}) \vee \bigvee_{p \xrightarrow{\tau^*} p' \xrightarrow{b} p''} (X_{p',q} \wedge X_{p'',q'})\right) \end{array} \right\}$
Observational	$\left\{ \begin{array}{rl} X_{p,q} = & \left(\bigwedge_{p \xrightarrow{\tau} p'} \bigvee_{q \xrightarrow{\tau^*} q'} X_{p',q'}\right) \wedge \left(\bigwedge_{p \xrightarrow{a} p'} \bigvee_{q \xrightarrow{\tau^* a \tau^*} q'} X_{p',q'}\right) \wedge \\ & \left(\bigwedge_{q \xrightarrow{\tau} q'} \bigvee_{p \xrightarrow{\tau^*} p'} X_{p',q'}\right) \wedge \left(\bigwedge_{q \xrightarrow{a} q'} \bigvee_{p \xrightarrow{\tau^* a \tau^*} p'} X_{p',q'}\right) \end{array} \right\}$
$\tau^*.a$	$\left\{ X_{p,q} = \left(\bigwedge_{p \xrightarrow{\tau^* a} p'} \bigvee_{q \xrightarrow{\tau^* a} q'} X_{p',q'}\right) \wedge \left(\bigwedge_{q \xrightarrow{\tau^* a} q'} \bigvee_{p \xrightarrow{\tau^* a} p'} X_{p',q'}\right) \right\}$
Safety	$\left\{ \begin{array}{rl} X_{p,q} = & Y_{p,q} \wedge Y_{q,p} \\ Y_{p,q} = & \left(\bigwedge_{p \xrightarrow{\tau^* a} p'} \bigvee_{q \xrightarrow{\tau^* a} q'} Y_{p',q'}\right) \end{array} \right\}$

All BESS shown in the table above can be made simple (at the price of a linear blow-up in size) by introducing additional variables such that the right-hand sides of equations become either disjunctive, or conjunctive formulas (e.g., the BES for strong equivalence is transformed into $\{X_{p,q} = \bigwedge_{p \xrightarrow{b} p'} Y_{p',b,q} \wedge \bigwedge_{q \xrightarrow{b} q'} Z_{p,b,q'}, Y_{p',b,q} = \bigvee_{q \xrightarrow{b} q'} X_{p',q'}, Z_{p,b,q'} = \bigvee_{p \xrightarrow{b} p'} X_{p',q'}\}$). The on-the-fly resolution of the resulting BESS consists in solving the variable $X_{q_{01},q_{02}}$ (which denotes the equivalence of the initial states of the two LTSS) by constructing the BES incrementally; this amounts to a demand-driven exploration of both LTSS, since the formulas in the right-hand sides of equations are evaluated by traversing the LTS transitions in a forward manner.

3 Distributed resolution algorithm

The architecture adopted for distributed BES resolution consists of P *worker* nodes of index $i \in [1..P]$ and one *coordinator* node of index 0, all nodes being connected by a network. In addition to the *distributed termination detection* (DTD) task (shown on Fig. 2, Sec. 3.3), the coordinator is also responsible for other activities, such as monitoring the progression of BES resolution, collecting statistics about the BES structure, handling early termination requested by the user or urgent termination caused by remote hardware or software failures. These features are implemented by appropriate extensions of DSOLVE (omitted in Fig. 1).

The DSOLVE algorithm is devised in terms of the boolean graph (V, E, L) [2] defined as follows: $V = \{X_1, ..., X_n\}$ is the set of vertices (boolean variables), $E = \{(X_i, X_j) | X_j \in \{X_{i1}, ..., X_{ik_i}\}\}_{1 \le i \le n}$ is the set of edges (dependencies between boolean variables), and $L : V \to \{\wedge, \vee\}$, $L(X_i) = op_i$ is the vertex labeling (boolean operator in the right-hand side of the equation). An instance of DSOLVE runs on each worker (task partitioning) and data (boolean variables) is distributed among workers by means of message passing according to a static hash function $h : V \to [1..P]$ as defined in [14]. Solving a Bes on-the-fly (i.e., computing the value of a variable X_k) consists in performing a forward exploration of the boolean graph starting at X_k, intertwined with a backward propagation of variables whose value is F (these variables are stable, since they reached their final value in a maximal fixed point computation). The resolution terminates either when X_k is stabilized to F (a counterexample was found), or when the graph portion reachable from X_k is entirely explored (the two Ltss are equivalent).

3.1 Bes *resolution*

Three aspects are covered by Fig.1: Bes resolution, communication, and termination detection. Bes resolution is defined by the following primitives:

DSolve. Each worker i executes an instance of DSolve on its own data structures. No variables are shared among processes. After a phase of initialization (lines 2-11), three activities take place: backward propagation of stable variables is given the highest priority (lines 14-26), then comes the forward exploration of boolean graph (V, E, L) (lines 27-36), and finally the reception of remote data is achieved (lines 43-44). Bes resolution begins with the initiator worker, of index $i{=}h(x)$, which expands the globally known variable of interest $x \in V$. Subsequently, the successor variables $E(x_i)$ generated by expanding variables at a worker are distributed to specific workers according to the hash function h (lines 27-36). If necessary, messages $Exp(x_i, y_i)$ are sent to remote workers with index $h(y_i) \neq i$ (line 34). During execution, all workers receive variables sent by other workers (lines 43-44). Symmetrically, stabilized variables ($c(x_i) = 0$) at a worker are backward propagated to predecessor variables $d(x_i)$ saved during expansion, whose corresponding specific workers are determined by h (lines 14-26). For remote workers ($h(w_i) \neq i$), messages $Evl(w_i, x_i)$ are sent (line 22). Bes resolution stops either when x becomes stable (line 91), or all variables reachable from x have been explored (line 111). DSolve returns the value of x (i.e., F if $c(x) = 0$). Orthogonally to the Bes resolution, specific variable dependencies $s(x_i)$ are saved during backward propagation of stable variables (line 88), in order to generate a diagnostic (counterexample) in case the variable x is stabilized to F (meaning that the two Ltss are not equivalent), following the approach presented in [19].

Expand. The routine Expand is called to update local data structures for forward exploration of the boolean graph: the working set W_i, the backward stabilization set B_i, and the search set S_i (lines 59-83).

Stabilize. The routine Stabilize stabilizes predecessor variables w_i by decre-

```
function DSolve$(x, (V, E, L), h, i)$ $: \mathbb{B}$ is
  if $h(x) = i$ then
    if $L(x) = \vee$ then
      $c(x) := |E(x)|$
    else
      $c(x) := 1$
    endif;
    $d(x) := \emptyset; W_i := \{x\}; S_i := \{x\}; B_i := \emptyset$
  else
    $W_i := \emptyset; S_i := \emptyset; B_i := \emptyset$
  endif;
  $term_i := inactive_i :=$ F; $sent_i := recv_i := 0$;
  while $\neg term_i$ do
    if $B_i \neq \emptyset$ then
      while $B_i \neq \emptyset$ do
        $x_i := choose(B_i)$;
        $B_i := B_i \setminus \{x_i\}$;
        forall $w_i \in d(x_i)$ do
          if $h(w_i)$ = $i$ then
            Stabilize$(w_i, x_i)$
          else
            Sending$(h(w_i), Evl(w_i, x_i))$
          endif
        endfor;
        $d(x_i) := \emptyset$
      endwhile
    elsif $W_i \neq \emptyset$ then
      $x_i := choose(W_i)$;
      $W_i := W_i \setminus \{x_i\}$;
      forall $y_i \in E(x_i)$ do
        if $h(y_i) = i$ then
          Expand$(x_i, y_i)$
        else
          Sending$(h(y_i), Exp(x_i, y_i))$
        endif
      endfor
    else
      if $\neg inactive_i$ then
        $inactive_i := true$;
        $sent_i := sent_i + 1$;
        Send$(coord, Idl(sent_i - recv_i))$
      endif;
      Receive$(sender_i, msg_i)$;
      Read$(sender_i, msg_i)$
    endif
  endwhile;
  return $c(x) = 0$
end

procedure Sending$(node_j, msg_j)$ is
  while $\neg$ISend$(node_j, msg_j) \wedge \neg term_i$ do
    if IReceive$(sender_i, msg_i)$ then
      Read$(sender_i, msg_i)$
    else
      WaitEvent$(\{0..P\}, node_j)$
    endif
  endwhile;
  $sent_i := sent_i + 1$
end

procedure Expand$(x_i, y_i)$ is
  if $y_i \notin S_i$ then
    $S_i := S_i \cup \{y_i\}$;
    $d(y_i) := \emptyset$;
    if $L(y_i) = \vee$ then
      $c(y_i) := |E(y_i)|$
    else
      $c(y_i) := 1$
    endif;
    if $c(y_i) \neq 0$ then
      $W_i := W_i \cup \{y_i\}$
    endif
  endif;

  if $c(y_i) = 0$ then
    if $h(x_i) = i$ then
      Stabilize$(x_i, y_i)$
    else
      $B_i := B_i \cup \{y_i\}$;
      $d(y_i) := d(y_i) \cup \{x_i\}$
    endif
  else
    $d(y_i) := d(y_i) \cup \{x_i\}$
  endif
end

procedure Stabilize$(w_i, y_i)$ is
  $c(w_i) := c(w_i) - 1$;
  if $c(w_i) = 0$ then
    if $L(y_i) = \wedge$ then
      $s(w_i) := y_i$
    endif;
    $B_i := B_i \cup \{w_i\}$;
    $term_i :=$ $c(x) = 0$
  endif
end

procedure Read$(sender_i, msg_i)$ is
  $recv_i := recv_i + 1$;
  if $sender_i \neq coord \wedge inactive_i$ then
    $inactive_i := false$;
    $sent_i := sent_i + 1$;
    Send$(coord, Act)$
  endif;
  case $msg_i$ is
    $Evl(x_i, y_i) \rightarrow$
      Stabilize$(x_i, y_i)$
    $Exp(x_{sender_i}, y_i) \rightarrow$
      Expand$(x_{sender_i}, y_i)$
    $Ack(stamp) \rightarrow$
      if $inactive_i$ then
        $sent_i := sent_i + 1$;
        Send$(coord, Ack(stamp))$
      endif
    $Trm \rightarrow$ $term_i := true$
  endcase
end
```

Fig. 1. Distributed local resolution of a Bes using its boolean graph

menting the counter of unstable successors $c(w_i)$ and updates the stabilization set B_i (lines 84-93).

3.2 Synchronization and communication

Apart from local computations, nodes exchange data by means of Receive and Send operations, thus redistributing work for better processor utilization, and for detecting termination of the distributed resolution. Adding to initial architectural choices (bidirectional channel between any two nodes, static hash function for data distribution, and mono-threaded algorithm), we aim at further improving the performance of DSolve resolution by using a communication layer that enables:

(i) reducing memory consumption;
(ii) maximizing the overlapping of communication and computations;
(iii) avoiding busy waiting on emission failures;
(iv) preventing deadlocks during communication between workers.

Point (i) can be solved by bounding the size of emission and reception buffers. However, this requires to deal with emission and reception failures (point (iii)), due to full buffers or empty buffers. Point (ii) requires asynchronous and non-blocking communication operations both in emission and in reception. Point (iii) suggests the combination of non-blocking and blocking communication. Finally, point (iv) can be addressed by allowing blocking communication only when workers are idle (i.e., no more local activity to be done, $B_i = W_i = \emptyset$).

Since our goal was to obtain an implementation of DSolve which can be easily integrated and released within the Cadp toolbox, we did not consider general message-passing environments such as Mpi, but preferred instead to use Cæsar_Network, a fine-tuned loosely coupled distributed communication library based on Unix sockets with bounded buffers and Tcp/Ip protocol developed according to a study made in [17]. By considering emission / reception failures and full communication buffers, and by introducing both blocking and non-blocking communication primitives, the complexity of the algorithm is slightly increased. However, this enables a fine-grained flow control of communication and reduces memory consumption related to emission and reception buffers.

The Cæsar_Network primitives used by DSolve are the following:

- Receive (line 43) enables blocking reception of a message from a node;
- IReceive (line 51) enables immediate (i.e., non-blocking) reception, and returns T if the message is received successfully, or F if the reception buffers

are empty;

- Send (line 41, 99 and 109) enables blocking emission of a message to a node;
- ISend (line 50) enables immediate (i.e., non-blocking) emission, and returns T if the message is sent successfully, or F if the emission buffers are full;
- WaitEvent (line 54) enables blocking waiting on the detection of communication events on the local reception and emission buffers associated to nodes in $\{0..P\}$.

3.3 Termination detection

The boolean variable $term_i$ is set to T when termination of the distributed Bes resolution is detected. Conditions of termination are: either the variable of interest x has been explicitly stabilized ($c(x) = 0$) during backward propagation of stable variables, or the boolean graph has been completely explored, i.e., all local working sets of variables are empty ($\forall i \in [1..P].W_i = B_i = \emptyset$), and no more messages are transiting through the network. The first condition is detected by the *initiator* worker, whose index is $h(x)$, when back propagating boolean values up to x (line 91). The second condition requires a Dtd algorithm.

We have used an algorithm derived from a combination of Dtd algorithms [16] and [21]. Our Dtd algorithm relies upon the coordinator node (of index *coord*=0), which is usually the end-user node from which the distributed Bes resolution is launched.

The Dtd consists of two phases: detection of global inactivity by the coordinator (i.e., $trm\_status{=}DETECT$), and confirmation of local inactivity by all workers (i.e., $trm\_status{=}CONF$). On each worker as well as on the coordinator, two counters $sent_i$ (or $sent$) and $recv_i$ (or $recv$) keep the number of exchanged messages in emission and in reception. When a worker i becomes idle, it sends an $Idl(sent_i - recv_i)$ message to the coordinator (lines 38-42). When it goes back to activity, the worker sends an *Act* message to the coordinator (lines 96-100). The coordinator keeps track for each worker i of the amount of messages transmitted minus those received ($nb\_msg(i)$, line 153). Thus, when the coordinator detects that all workers are idle (i.e., $\forall i \in [1..P].inactive_i{=}\mathsf{T}$ and $nb\_idle{=}P$), it also verifies that no messages are still in transit (i.e., invariant $total\_msg{=}\sum_{i=1}^{P}(sent_i - recv_i)$ and $total\_msg + (sent - recv) = 0$). If both conditions are respected (lines 156-157), then a phase of inactivity confirmation, indexed by a counter *stamp*, is started. The coordinator broadcasts to all workers an $Ack(stamp)$ message (lines 124-128), thus flushing possible residual messages transiting between workers and the coordinator. Each inactive worker acknowledges the reception of an $Ack(stamp)$ message by sending

```
114: procedure Coordinator is
115:   $trm\_status := DETECT$;
116:   $sent := recv := 0$;
117:   $stamp := 0$;
118:   $total\_msg := nb\_idle := nb\_ack := 0$;
119:   forall $i$ in $[1..P]$ do
120:     $nb\_msg(i) := 0$
121:   endfor;
122:   while $trm\_status \neq TERM$ do
123:     case $trm\_status$ is
124:     $CONF \rightarrow$ while $bcast\_node \leq P$ $\wedge$
125:        ISend($bcast\_node, Ack(stamp)$) do
126:          $bcast\_node := bcast\_node + 1$;
127:          $sent := sent + 1$
128:        endwhile
129:     $STOP \rightarrow$ while $bcast\_node \leq P$ $\wedge$
130:        ISend($bcast\_node, Trm$) do
131:          $bcast\_node := bcast\_node + 1$
132:        endwhile;
133:        if $bcast\_node > P$ then
134:          $trm\_status := TERM$
135:        endif
136:     endcase;
137:     if $trm\_status = DETECT$ then
138:        Receive($msg, sender$);
139:        ReadCoord($msg, sender$)
140:     elsif IReceive($msg, sender$) then
141:        ReadCoord($msg, sender$)
142:     endif
143:   endwhile
144: end

145: procedure ReadCoord($m, s$) is
146:   $recv := recv + 1$;
147:   case $m$ is
148:     $Act \rightarrow nb\_idle := nb\_idle - 1$;
149:       $total\_msg := total\_msg - nb\_msg(s)$;
150:       if $trm\_status = CONF$ then
151:         $trm\_status := DETECT$
152:       endif
153:     $Idl(k) \rightarrow nb\_msg(s) := k$;
154:       $nb\_idle := nb\_idle + 1$;
155:       $total\_msg := total\_msg + nb\_msg(s)$;
156:       if $total\_msg = -(sent - recv)$
157:       $\wedge$ $nb\_idle = P$ then
158:         $trm\_status := CONF$;
159:         $bcast\_node := 1$; $nb\_ack := 0$;
160:         $stamp := stamp + 1$
161:       endif
162:     $Ack(k) \rightarrow$ if $k = stamp$ then
163:         if $trm\_status = DETECT$ then
164:           if $total\_msg = -(sent - recv)$
165:           $\wedge$ $nb\_idle = P$ then
166:             $trm\_status := CONF$;
167:             $bcast\_node := 1$; $nb\_ack := 0$;
168:             $stamp := stamp + 1$
169:           endif
170:         elsif $trm\_status = CONF$ then
171:           $nb\_ack := nb\_ack + 1$;
172:           if $total\_msg = -(sent - recv)$
173:           $\wedge$ $nb\_ack = P$ then
174:             $trm\_status := STOP$;
175:             $bcast\_node := 1$
176:           endif
177:         endif
178:       endif
179:   endcase
180: end
```

Fig. 2. Termination detection algorithm (coordinator node)

back the same $Ack(stamp)$ message to the coordinator (lines 106-110). If a worker is active upon reception of an $Ack(stamp)$ message, it simply ignores it. In that case, an Act message from that worker must eventually arrive to the coordinator. Finally, the coordinator detects the global termination if it receives P $Ack(stamp)$ messages (i.e., $nb\_ack = P$, lines 162-178). It can then broadcast this termination detection (i.e., $trm\_status = STOP$) to all workers (lines 129-135).

3.4 Correctness and complexity

Our distributed Bes resolution algorithm is based on the theory of boolean graphs underlying the sequential algorithms [2,26]. It is composed of two intertwined graph traversals (forward and backward), whose worst-case time complexity is $O(|V|+|E|)$. The same bound applies for memory complexity, because of the dependencies $d(y)$ stored during graph exploration. Assuming a perfect partition function, the message complexity is $O(2 \cdot |E| \cdot (P-1)/P)$, the worst-case being obtained with two messages (expansion and stabilization) exchanged per edge. Theoretically, our DTD algorithm has a complexity $O(|E|)$, but practically it reveals to be very efficient, with only 0.01% of total exchanged messages used for termination detection. Indeed, the coordinator has a sufficiently accurate and up to date image of the distributed computation status to perform the DTD with a small number of attempts.

4 Implementation and experiments

Our implementation of DSolve and Coordinator (8500 lines of C code) has been integrated to the generic Bes resolution library Cæsar_Solve [20] developed using the Open/Cæsar environment [12]. Hence we immediately obtained a distributed version of the Bisimulator [20] on-the-fly equivalence checker of the Cadp verification toolbox [13], which uses Cæsar_Solve as verification engine. This tool architecture is highly modular, allowing to separate the front-end (encoding of the equivalence relations as Bess) from the back-end (Bes resolution). To compute the successors of a boolean variable $X_{p,q}$ denoting the equivalence of states p and q modulo a given relation, the front-end, which is called sequentially and independently on each worker, explores the two Ltss forward starting at p and q, according to the definition of that relation (see the table in Section 2). Note that for weak equivalence relations (branching, observational, $\tau^*.a$, safety), the front-end must perform transitive closures on τ-transitions in both Ltss.

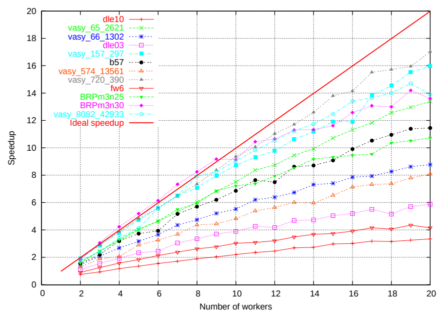
We have carried out an extensive set of experiments on a cluster of 20 Xeon 2.4 GHz Linux Pcs, with 1.5 GB of main memory, interconnected by a Gigabit network. The Ltss considered were mainly extracted from the Vlts benchmark suite [1], which is designed to be a reference criterion for scientific assessment of algorithms and tools operating on large graphs, such as distributed equivalence checkers. Only a dozen of experiments that took at least few seconds of computation are shown in this section. Note that to obtain an accurate image of the performances, in the experimental results described below we excluded the fixed costs of system-dependent activities (loading of code on remote nodes, initialization of connections, and copying of Lts files), and we kept only the costs of distributed resolution and termination detection. We performed each experiment ten times. Each point on each curve represents the average of the eight values corresponding to the measurements obtained excluding the maximum and minimum values.

4.1 Speedup

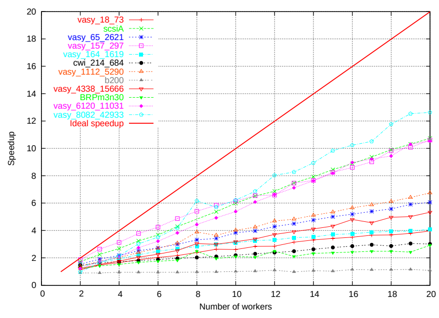
One way to quantify the efficiency of a parallel algorithm is to compute the absolute speedup $S = T_1/T_p$ by using as baseline the uniprocessor time T_1 for the best known uniprocessor (sequential) algorithm, and the time T_p with P workers. Fig. 3 shows experimental data comparing the performance of the distributed version of Bisimulator (based on DSolve) and its sequential version (based on a breadth-first search algorithm of Cæsar_Solve). For each equivalence relation R and Lts M, the experiments concern the comparison modulo R of M with M_R, its minimized version w.r.t. R. The choice of this comparison was motivated by two reasons: (a) it reproduces a situation frequently encountered in practice, when a designer specifies both the system behavior (*protocol*) and its external behavior (*service*), which correspond here to M and M_R; (b) it represents a worst-case behavior for on-the-fly equivalence checking, since the algorithm must explore the Bes (and the two Ltss) entirely before deciding the equivalence of M and M_R. We also performed various experiments comparing non-equivalent Ltss: in all cases, both the distributed and sequential versions of Bisimulator were extremely fast in discovering counterexamples.

Strong equivalence. Fig. 3(a) shows the speedups obtained for strong equivalence checking with distributed Bisimulator on a set of examples, ordered by increasing sizes, from $9.757 \cdot 10^3$ states and $24.352 \cdot 10^3$ transitions ($dle10$) to $8.082 \cdot 10^6$ states and $42.933 \cdot 10^6$ transitions ($vasy\_8082\_42933$). Strong equivalence is well-suited for distribution: there is very few time spent in the front-end (no transitive closure on τ-transitions needed), and curves show linear speedups from low (still better than the sequential times) to nearly optimal. Moreover, speedup gets better when the Lts size increases. For example, the sequential check of experiment $BRPm3n30$ (Bounded Retransmission Protocol with 3 retransmissions and packet length 30, i.e. $5.957 \cdot 10^6$ states, $9.225 \cdot 10^6$ transitions) took 332.53 seconds, whereas the parallel check with 13 workers took 29.06 seconds (speedup 11.5).

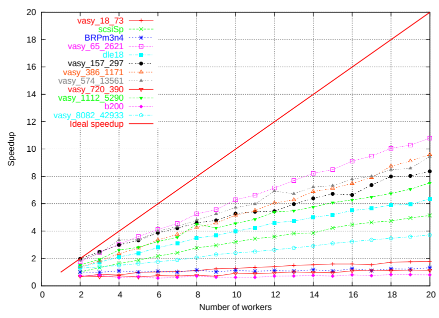
$\boldsymbol{\tau^*.a}$ and safety equivalences. Fig. 3(b) shows the speedups obtained for $\tau^*.a$ equivalence on a similar set of examples to the one used for strong equivalence (safety equivalence shows a similar behavior). The computations of these equivalences involves extensive transitive closures on τ-transitions (performed sequentially by the front-end present on each worker) and very small Bess in the case of Ltss containing many τ-transitions.



(a) Strong equivalence



(b) $\tau^*.a$ equivalence



(c) Observational equivalence

Fig. 3. Speedup for three equivalences

Hence, the speedups observed are lower than for strong equivalence, and start to be high on large LTSs such as *vasy*_8082_42933, where speedup grows up to 8.22 with 13 workers.

Branching and observational equivalences. Fig. 3(*c*) shows the speedups obtained for observational equivalence (branching equivalence shows a similar behavior). Contrary to $\tau^*.a$ and safety equivalences, the BESS encoding observational and branching equivalences are much larger, and therefore distributed resolution has a stronger impact on performance. Hence the curves show generally better speedups, in particular for LTSs with few τ-transitions or deterministic behavior, such as *vasy*_65_2621, where speedup grows up to 7.86 with 13 workers. Global observations can be drawn w.r.t. the nature of LTSs being checked. Three factors influence the performance of distributed BISIMULATOR: size of LTSs, percentage of τ-transitions, and degree of nondeterminism. Hence, when neither τ-transitions nor nondeterminism are present in the LTSs, then good speedups are achieved for all equivalence relations, as shown by experiments *vasy*_1112_5290, *vasy*_574_13561, *vasy*_65_2621, or *vasy*_8082_42933. On the contrary, an increased percentage of τ-transitions results in low speedups for $\tau^*.a$ and safety equivalences (because of expensive front-end computations), but still good speedups for strong and observational equivalences (because of important Bes sizes), as illustrated by experiments $BRPm3n30$ on Fig. 3. Similarly, increasing both nondeterminism and percentage of τ-transitions yields large Bess. In this cases, only strong equivalence can terminate in reasonable time (less than 45 minutes in sequential) and shows high speedups with experiment $b57$ on Fig. 3(a). Weak equivalences either could not terminate (e.g., observational equivalence for $b57$), or they showed no speedup (e.g., $\tau^*.a$ and safety equivalence for $b200$).

4.2 Scalability

Interesting insights into DSolve characteristics are provided by the above experimental measures together with the scalability results shown on Fig. 4.

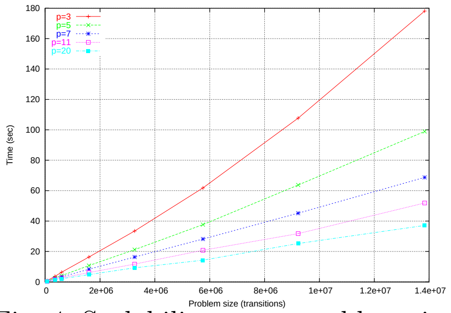


Fig. 4. Scalability w.r.t. problem size

Each curve on Fig. 4 represents the time needed for experiment $BRPm3nK$ (Bounded Retransmission Protocol with 3 retransmissions and packet length K varying from 4 to 35) using strong equivalence on a fixed number P of Xeon workers (between 3 and 20). The linear progression of the curves indicates that DSolve is well-adapted to increases in problem size, making an efficient use of memory and Cpu. As for another large example, DSolve handles the strong equivalence checking of experiment $b200$ (Alternating Bit Protocol with 200 different messages), whose generated Bes size is $2.4 \cdot 10^8$ variables, in about 24 minutes with 15 workers, whereas the sequential Bes resolution fails to achieve it due to current implementation restrictions on Bes size (maximum of $1.6 \cdot 10^7$ variables).

4.3 Memory

We have shown that performance is reasonable with respect to run times.

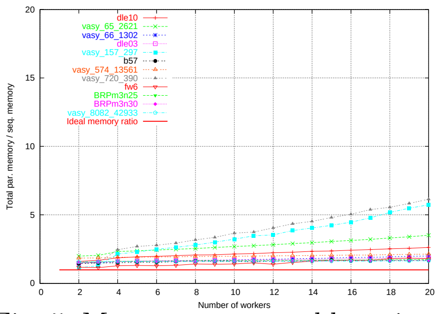


Fig. 5. Memory w.r.t. problem size

However, memory limitation of existing sequential algorithms is the main motivation for distribution. Fig. 5 sustains by practical experiments that DSolve makes an efficient use of memory. It presents results obtained for strong equivalences on a dozen of Vlts benchmarks sorted by increasing size, from $9 \cdot 10^3$ states, $25 \cdot 10^3$ transitions to $8 \cdot 10^6$ states, $43 \cdot 10^6$ transitions Ltss and with increasing number of nodes (from 2 to 20). We take into account only the data structures used by the DSolve algorithm, which include the hash tables used for storing boolean variables, and by the Cæsar_Network library, which include communication buffers. The impact of adding more workers is rather low, which is shown by a ratio, between total distributed memory consumption and corresponding sequential memory consumption, that is hardly increasing. The bigger is the Lts to be checked, the lower is the ratio.

5 Conclusion and future work

We presented DSolve, a new algorithm for on-the-fly distributed resolution of Bess using several machines connected by a network. DSolve serves as verification engine in the distributed version of Bisimulator, an on-the-fly equivalence checker developed within the Cadp toolbox [13] using the Open/Cæsar environment for Lts exploration [12]. The experiments we carried out on a Pc cluster using benchmark examples and five widely-used equivalence relations showed quasi-linear speedups and a good scalability of the distributed version w.r.t. the sequential version of Bisimulator.

The implementation of DSolve is application-independent and was integrated in the generic Cæsar_Solve library [20], which already provides four different sequential algorithms for on-the-fly Bes resolution. We are currently using DSolve to obtain distributed versions for other applications built using Cæsar_Solve, such as alternation-free μ-calculus model-checking [20] and τ-confluence reduction [23]. We also plan to extend Bisimulator with other equivalence relations, such as Markovian bisimulation [15].

References

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

[2] H. R. Andersen. Model Checking and Boolean Graphs. *Theoretical Computer Science*, 126(1):3–30, April 1994.

[3] H. R. Andersen and B. Vergauwen. Efficient Checking of Behavioural Relations and Modal Assertions using Fixed-Point Inversion. In P. Wolper, editor, *Proc. of the 7th International Conference on Computer Aided Verification CAV'95 (Liege, Belgium)*, LNCS vol. 939, pp. 142–154. Springer Verlag.

[4] A. Arnold and P. Crubillé. A Linear Algorithm to Solve Fixed-Point Equations on Transition Systems. *Information Processing Letters*, 29:57–66, 1988.

[5] S. Blom and S. Orzan. A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. In L. Brim and O. Grumberg, editors, *Proc. of the 1st International Workshop on Parallel and Distributed Model Checking PDMC'02 (Brno, Czech Republic)*, ENTCS 68(4). Elsevier, 2002.

[6] S. Blom and S. Orzan. Distributed Branching Bisimulation Reduction of State Spaces. In L. Brim and O. Grumberg, editors, *Proc. of the 2nd International Workshop on Parallel and Distributed Model Checking PDMC'03 (Boulder, Colorado, USA)*, ENTCS 89(1). Elsevier, 2003.

[7] B. Bollig, M. Leucker, and M. Weber. Local Parallel Model Checking for the Alternation Free Mu-Calculus. In D. Bonaki and S. Leue, editors, *Proc. of the 9th International SPIN Workshop on Model Checking of Software SPIN'2002 (Grenoble, France)*, LNCS vol. 2318, pp. 128–147. Springer Verlag.

[8] A. Bouajjani, J-C. Fernandez, S. Graf, C. Rodríguez, and J. Sifakis. Safety for Branching Time Semantics. In *Proc. of 18th ICALP*, LNCS vol. 510. Springer Verlag.

[9] R. Cleaveland and B. Steffen. Computing Behavioural Relations, Logically. In *Proc. of the 18th ICALP*, LNCS vol. 510, pp. 127–138. Springer Verlag.

[10] R. Cleaveland and O. Sokolsky. *Equivalence and Preorder Checking for Finite-State Systems.* In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 6, pages 391–424. North-Holland, 2001.

[11] J-C. Fernandez and L. Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In K. G. Larsen and A. Skou, editors, *Proc. of the 3rd Workshop on Computer-Aided Verification CAV'91 (Aalborg, Denmark)*, LNCS vol. 575. Springer Verlag.

[12] H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In B. Steffen, editor, *Proc. of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, LNCS vol. 1384, pp. 68–84. Springer Verlag. Full version available as INRIA Research Report RR-3352.

[13] 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.

[14] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In Matthew B. Dwyer, editor, *Proc. of the 8th International SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada)*, LNCS vol. 2057, pp. 217–234. Springer Verlag. Revised version available as INRIA Research Report RR-4341.

[15] H. Hermanns and M. Siegle. Bisimulation Algorithms for Stochastic Process Algebras and their BDD-based Implementation. In J-P. Katoen, editor, *Proc. of the 5th International AMAST Workshop ARTS'99 (Bamberg, Germany)*, LNCS vol. 1601, pp. 244–265. Springer Verlag.

[16] S. T. Huang and P. W. Kao. Detecting Termination of Distributed Computations by External Agents. *Journal of Information Science and Engineering*, 7(2):187–201, 1991.

[17] C. Joubert. Distributed Model Checking: From Abstract Algorithms to Concrete Implementations. In L. Brim and O. Grumberg, editors, *Proc. of the 2nd International Workshop on Parallel and Distributed Model Checking PDMC'03 (Boulder, Colorado, USA)*, ENTCS 89(1). Elsevier, 2003.

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

[19] R. Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In S. Graf and M. Schwartzbach, editors, *Proc. of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2000 (Berlin, Germany)*, LNCS vol. 1785, pp. 251–265. Springer Verlag. Full version available as INRIA Research Report RR-3861.

[20] R. Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In H. Garavel and J. Hatcliff, editors, *Proc. of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003 (Warsaw, Poland)*, LNCS vol. 2619, pp. 81–96. Springer Verlag. Full version available as INRIA Research Report RR-4711.

[21] F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2:161–175, 1987.

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

[23] G. Pace, F. Lang, and R. Mateescu. Calculating τ-Confluence Compositionally. In Jr W. A. Hunt and F. Somenzi, editors, *Proc. of the 15th International Conference on Computer Aided Verification CAV'2003 (Boulder, Colorado, USA)*, LNCS vol. 2725, pp. 446–459. Springer Verlag. Full version available as INRIA Research Report RR-4918.

[24] D. Park. Concurrency and Automata on Infinite Sequences. In P. Deussen, editor, *Theoretical Computer Science*, LNCS vol. 104, pp. 167–183. Springer Verlag.

[25] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[26] B. Vergauwen and J. Lewi. Efficient Local Correctness Checking for Single and Alternating Boolean Equation Systems. In S. Abiteboul and E. Shamir, editors, *Proc. of the 21st ICALP (Vienna)*, LNCS vol. 820, pp. 304–315. Springer Verlag.