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

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Distributed On-the-Fly Equivalence Checking

# Christophe Joubert and Radu Mateescu[1](#_bookmark0)

*INRIA Rhoˆne-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) mod- ulo a given equivalence relation by exploring them in a demand-driven way. Since it avoids the ex- plicit 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.

# 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. Nu- merous equivalence relations (strong [[24](#_bookmark34)], branching [[25](#_bookmark35)], observational [[22](#_bookmark32)], *τ* ∗*.a* [[11](#_bookmark21)], safety [[8](#_bookmark18)], 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 equiv- alence of finite Ltss: *globally*, which requires the construction of the two Ltss

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

1571-0661 © 2005 Elsevier B.V. Open access under [CC BY-NC-ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

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 equiv- alence checking were designed and implemented in verification tools (see [[10](#_bookmark20)] 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 equiva- lence relation. Recently, distributed global equivalence checking algorithms were proposed [[5](#_bookmark12),[6](#_bookmark13)], 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](#_bookmark21)] and preorder checking [[10](#_bookmark20)] were based on the following principle: a forward, si- multaneous exploration of the two Ltss is performed starting from their initial states, until either some execution pattern showing non equivalence (coun- terexample) is encountered, or the two Ltss have been entirely explored. An- other approach for on-the-fly equivalence checking is based upon the char- acterizations of equivalence relations in terms of Boolean Equation Systems (Bess) [[9](#_bookmark19),[3](#_bookmark14)], which allow to use efficient algorithms for on-the-fly Bes resolu- tion [[20](#_bookmark28)]. 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 devel- oping the on-the-fly equivalence checker Bisimulator, which uses the generic Cæsar Solve [[20](#_bookmark28)] Bes resolution library, built using the Open/Cæsar environ- ment for on-the-fly Lts exploration [[12](#_bookmark22)] of the Cadp verification toolbox [[13](#_bookmark23)]. 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](#_bookmark17)]: 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 worksta- tions (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 us- ing Cæsar Solve, such as alternation-free *µ*-calculus model-checking [[20](#_bookmark28)] and *τ* -confluence reduction [[23](#_bookmark33)].

The remainder of the paper is organized as follows. Section [2](#_bookmark1) recalls the definitions of Bess and the encodings of five widely-used equivalence relations in terms of Bess. Section [3](#_bookmark2) describes in detail the DSolve algorithm and Sec- tion [4](#_bookmark6) shows experimental data comparing the performance of the distributed and sequential versions of Bisimulator. Finally, Section [5](#_bookmark10) gives some conclud- ing remarks and directions for future work.

# 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 (*Aτ* = *A* ∪ {*τ* } also contains the invisible action *τ* ), *T* ⊆ *Q* × *Aτ* × *Q* is the transition relation, and *q*0 ∈ *Q* is the initial state. A transition *q*1 → *q*2 ∈ *T* means that the system can move from state *q*1 to state

*a*

*q*2 by executing action *a*. Given a language *l* ⊆ *A* ∗, *q*1 →*l*

*τ*

*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 *Mi* = (*Qi, A, Ti, q*0*i*), *i* ∈ {1*,* 2}. A Bes is a set of equations *B* = {*Xi* = *Xi*1 *opi* ··· *opi Xiki* }1≤*i*≤*n*, where

*Xi* are boolean variables and *opi* ∈ {∨*,* ∧}. For efficiency of resolution, we

consider *simple* Bess [[4](#_bookmark15)], whose right-hand sides of equations are pure dis-

junctive 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

Φ : *n* → *n*, Φ(*b*1*, ..., bn*) = ( **[***Xi*1 *opi* ··· *opi Xiki* ]][*b*1*/X*1*,... , bn/Xn*])1≤*i*≤*n*, where **[***ϕ*]]*δ* 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](#_bookmark29)].

Various equivalence relations between Ltss were characterized in terms of Bess [[9](#_bookmark19),[3](#_bookmark14)]. The table below shows the encodings of five widely-used equiva- lences: strong [[24](#_bookmark34)], branching [[25](#_bookmark35)], observational [[22](#_bookmark32)], *τ* ∗*.a* [[11](#_bookmark21)], and safety [[8](#_bookmark18)]. Each relation is represented as a Bes whose variables *Xp,q* indicate whether the states *p* ∈ *Q*1 and *q* ∈ *Q*2 are equivalent or not (*a* ∈ *A* and *b* ∈ *Aτ* ). For each equivalence, the corresponding preorder (in *grey*) is obtained by deleting either the 2nd conjunct (for strong, *τ* ∗*.a*, safety, and branching), or the 3rd and 4th conjuncts (for observational) in the right-hand sides of the equations.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Relation | Encoding | | | | | | | | |
| Strong | n *Xp,q*  8 | = |  | “ V *b* ' W *b* ' *Xp*' *,q*' ”  *p*→*p q*→*q*  V “ | ∧ “ V *b* ' W *b* ' *Xp*' *,q*' ” o  *q*→*q p*→*p*  W ” 9 | | | | |
| Branching | < *Xp,q* | = |  | *p*→*b p*' (*b* = *τ* ∧ *Xp ,q* ) ∨ *τ*∗ ' *b* '' (*Xp,q* ∧ *Xp ,q* )  ' ' ' ''  *q* →*q* →*q*  V “ W ” | | | | ∧ = | |
| 8: | | *q*→*b q*' (*b* = *τ* ∧ *Xp,q* ) ∨ *τ*∗ ' *b* '' (*Xp ,q* ∧ *Xp ,q* )  ' ' '' '  *p*→*p* →*p*  “ V W ” “ V W ” 9 | | | | | | ; |
| Observational | < *Xp,q* | = |  | *p τ p*' *τ* ∗ *Xp*' *,q*' ∧ *a* ' *τ* ∗ *aτ* ∗ *Xp*' *,q*'  → *q* →*q*' *p*→*p q* −→ *q*'  “ V W ” “ V W ” | | | ∧ = | | |
| n: | | *q*→*τ q*' *τ* ∗ ' *Xp*' *,q*' ∧ *q*→*a q*' *τ* ∗ *aτ* ∗ ' *Xp*' *,q*' ;  *p*→*p p* −→ *p*  “ V W ” “ V W ” o | | | | | |  |
| ∗  *τ .a* | *Xp,q*  8 | = |  | *τ* ∗ *a τ* ∗ *a Xp*' *,q*'  *p*−→*p*' *q*−→*q*' | | ∧ *τ* ∗ *a τ* ∗ *a Xp*' *,q*'  *q*−→*q*' *p*−→*p*'  9 | | | |
| Safety | < *Xp,q* =  : *Yp,q* = | | *Y*“ V ∧ *Y* W ” =  *p,q q,p*  *τ* ∗ *a τ* ∗ *a Yp*' *,q*' ;  *p*−→*p*' *q*−→*q*' | | | | | |  |

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 con-

junctive formulas (e.g., the

Bes

for strong equivalence is transformed into

{*Xp,q* =

*p*→*b p*

' *Yp*'*,b,q* ∧

*q*→*b q*

' *Zp,b,q*' *, Yp*'*,b,q* =

*q*→*b q*

' *Xp*'*,q*' *, Zp,b,q*' =

*p*→*b p*

' *Xp*'*,q*' }).

The on-the-fly resolution of the resulting Bess consists in solving the variable

*Xq*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 ex-

ploration of both Ltss, since the formulas in the right-hand sides of equations are evaluated by traversing the Lts transitions in a forward manner.

# Distributed resolution algorithm

The architecture adopted for distributed Bes resolution consists of *P worker* nodes of index *i* ∈ [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](#_bookmark5), Sec. [3.3](#_bookmark4)), 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](#_bookmark3)).

The DSolve algorithm is devised in terms of the boolean graph (*V* , *E*, *L*) [[2](#_bookmark16)] defined as follows: *V* = {*X*1*, ..., Xn*} is the set of vertices (boolean variables), *E* = {(*Xi, Xj*)|*Xj* ∈ {*Xi*1*, ..., Xiki* }}1≤*i*≤*n* is the set of edges (dependencies between boolean variables), and *L* : *V* → {∧*,* ∨}, *L*(*Xi*) = *opi* 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* → [1*..P* ] as defined in [[14](#_bookmark24)]. Solving a Bes on-the-fly (i.e., computing the value of a variable *Xk*) consists in performing a forward exploration of the boolean graph starting at *Xk*, 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 *Xk* is stabilized to F (a counterexample was found), or when the graph portion reachable from *Xk* is entirely explored (the

two Ltss are equivalent).

* 1. Bes *resolution*

Three aspects are covered by Fig.[1](#_bookmark3): Bes resolution, communication, and ter-

mination detection. Bes resolution is defined by the following primitives:

DSolve. Each worker *i* executes an instance of DSolve on its own data struc- tures. No variables are shared among processes. After a phase of initializa- tion (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 recep- tion 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* ∈ *V* . Subsequently, the successor variables *E*(*xi*) generated by expanding variables at a worker are distributed to specific workers accord- ing to the hash function *h* (lines 27-36). If necessary, messages *Exp*(*xi, yi*) are sent to remote workers with index *h*(*yi*) /= *i* (line 34). During execution, all workers receive variables sent by other workers (lines 43-44). Symmetri- cally, stabilized variables (*c*(*xi*) = 0) at a worker are backward propagated to predecessor variables *d*(*xi*) saved during expansion, whose correspond- ing specific workers are determined by *h* (lines 14-26). For remote workers (*h*(*wi*) /= *i*), messages *Evl*(*wi, xi*) are sent (line 22). Bes resolution stops ei- ther 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*(*xi*) 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](#_bookmark30)].

Expand. The routine Expand is called to update local data structures for for- ward exploration of the boolean graph: the working set *Wi*, the backward stabilization set *Bi*, and the search set *Si* (lines 59-83).

Stabilize. The routine Stabilize stabilizes predecessor variables *wi* by decre-

1: function DSolve(*x,* (*V, E, L*)*, h, i*) :is

2: if *h*(*x*)= *i* then

3: if *L*(*x*)= ∨ then

4: *c*(*x*) := |*E*(*x*)|

5: else

6: *c*(*x*) :=1

7: endif;

8: *d*(*x*) := ∅; *Wi* := {*x*}; *Si* := {*x*}; *Bi* := ∅

9: else

10: *Wi* := ∅; *Si* := ∅; *Bi* := ∅

11: endif;

12: *termi* := *inactivei* := F; *senti* := *recvi* := 0;

13: while ¬*termi* do

14: if *Bi* /= ∅ then

15: while *Bi* /= ∅ do

16: *xi* := *choose*(*Bi*);

17: *Bi* := *Bi* \ {*xi*};

59: procedure Expand(*xi, yi*) is

60: if *yi* ∈*/ Si* then

61: *Si* := *Si* ∪ {*yi*};

62: *d*(*yi*) := ∅;

63: if *L*(*yi*)= ∨ then

64: *c*(*yi*) := |*E*(*yi*)|

65: else

66: *c*(*yi*) := 1

67: endif;

68: if *c*(*yi*) /=0 then

69: *Wi* := *Wi* ∪ {*yi*}

70: endif

71: endif;

72:

73: if *c*(*yi*)=0 then

74: if *h*(*xi*)= *i* then

18: forall *wi* ∈ *d*(*xi*) do

75:

Stabilize(*xi, yi*)

19: if *h*(*wi*) = *i* then

76: else

20:

Stabilize(*wi, xi*)

77: *Bi* := *Bi* ∪ {*yi*};

21: else

78: *d*(*yi*) := *d*(*yi*) ∪ {*xi*}

22:

Sending(*h*(*wi*)*,Evl*(*wi, xi*))

79: endif

23: endif

24: endfor;

25: *d*(*xi*) := ∅

26: endwhile

27: elsif *Wi* /= ∅ then

28: *xi* := *choose*(*Wi*);

29: *Wi* := *Wi* \ {*xi*};

30: forall *yi* ∈ *E*(*xi*) do

31: if *h*(*yi*)= *i* then

80: else

81: *d*(*yi*) := *d*(*yi*) ∪ {*xi*}

82: endif

83: end

84: procedure Stabilize(*wi, yi*) is

85: *c*(*wi*) := *c*(*wi*) − 1;

86: if *c*(*wi*)= 0 then

87: if *L*(*yi*)= ∧ then

32:

Expand(*xi, yi*)

88: *s*(*wi*) := *yi*

33: else

89: endif;

34:

Sending(*h*(*yi*)*,Exp*(*xi, yi*))

90: *Bi* := *Bi* ∪ {*wi*};

35: endif

36: endfor

37: else

38: if ¬*inactivei* then

39: *inactivei* := *true*;

40: *senti* := *senti* + 1;

91: *termi* := *c*(*x*)= 0

92: endif

93: end

94: procedure Read(*senderi, msgi*) is

41:

Send(*coord, Idl*(*senti* − *recvi*))

95: *recvi* := *recvi* + 1;

42: endif;

96: if *senderi* /= *coord* ∧ *inactivei* then

43:

44:

Receive(*senderi, msgi*);

Read(*senderi, msgi*)

97: *inactivei* := *false*;

98: *senti* := *senti* + 1;

45: endif

99:

Send(*coord, Act*)

46: endwhile;

47: return *c*(*x*)=0

48: end

100: endif;

101: case *msgi* is

102: *Evl*(*xi, yi*) →

103:

Stabilize(*xi, yi*)

49: procedure Sending(*nodej , msgj* ) is

104: *Exp*(*xsenderi , yi*) →

50: while ¬ISend(*nodej , msgj* ) ∧ ¬*termi* do

105:

Expand(*xsenderi , yi*)

51: if IReceive(*senderi, msgi*) then

106: *Ack*(*stamp*) →

if *inactivei* then

52:

Read(*senderi, msgi*)

107:

53: else

108: *senti* := *senti* + 1;

54:

WaitEvent({0*..P* }*, nodej* )

109:

Send(*coord, Ack*(*stamp*))

55: endif

56: endwhile;

57: *senti* := *senti* +1

58: end

110: endif

111: *Trm* → *termi* := *true*

112: endcase

113:end

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

menting the counter of unstable successors *c*(*wi*) and updates the stabiliza- tion set *Bi* (lines 84-93).

* 1. *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 ini- tial 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 communi- cation layer that enables:

* + 1. reducing memory consumption;
    2. maximizing the overlapping of communication and computations;
    3. avoiding busy waiting on emission failures;
    4. 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 asyn- chronous and non-blocking communication operations both in emission and in reception. Point (iii) suggests the combination of non-blocking and block- ing 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, *Bi* = *Wi* = ∅).

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 devel- oped according to a study made in [[17](#_bookmark26)]. 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 commu- nication 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 re- turns 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 communi- cation events on the local reception and emission buffers associated to nodes in {0*..P* }.
  1. *Termination detection*

The boolean variable *termi* 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 propa-

gation of stable variables, or the boolean graph has been completely explored, i.e., all local working sets of variables are empty (∀*i* ∈ [1*..P* ]*.Wi* = *Bi* = ∅), 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 propagat- ing 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](#_bookmark27)] and [[21](#_bookmark31)]. 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 coor- dinator (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 coor- dinator, two counters *senti* (or *sent*) and *recvi* (or *recv*) keep the number of ex- changed messages in emission and in reception. When a worker *i* becomes idle, it sends an *Idl*(*senti* − *recvi*) 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., ∀*i* ∈ [1*..P* ]*.inactivei*=T

and *nb idle*=*P* ), it also verifies that no messages are still in transit (i.e., in-

variant *total msg*=Σ*P* (*senti* − *recvi*) and *total msg* +(*sent*−*recv*)= 0). If

*i*=1

both conditions are respected (lines 156-157), then a phase of inactivity con- firmation, 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 inac- tive worker acknowledges the reception of an *Ack*(*stamp*) message by sending

114:procedure Coordinator is 115: *trm status* := *DET ECT* ; 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* /= *TERM* do

123: case *trm status* is

124: *CONF* → while *bcast node* ≤ *P* ∧

145:procedure ReadCoord(*m, s*) is

146: *recv* := *recv* + 1;

147: case *m* is

148: *Act* → *nb idle* := *nb idle* − 1;

149: *total msg* := *total msg* − *nb msg*(*s*);

150: if *trm status* = *CONF* then

151: *trm status* := *DET ECT*

152: endif

153: *Idl*(*k*) → *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: ∧ *nb idle* = *P* then

125:

ISend(*bcast node, Ack*(*stamp*)) do

158: *trm status* := *CONF* ;

126: *bcast node* := *bcast node* + 1;

127: *sent* := *sent* +1

128: endwhile

129: *ST OP* → while *bcast node* ≤ *P* ∧

159: *bcast node* := 1; *nb ack* := 0;

160: *stamp* := *stamp* +1

161: endif

162: *Ack*(*k*) → if *k* = *stamp* then

130:

ISend(*bcast node, T rm*) do

163: if *trm status* = *DET ECT* then

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* = *DET ECT* then

164: if *total msg* = −(*sent* − *recv*)

165: ∧ *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

138:

Receive(*msg, sender*);

171: *nb ack* := *nb ack* + 1;

139: ReadCoord(*msg, sender*)

140: elsif IReceive(*msg, sender*) then

141: ReadCoord(*msg, sender*)

142: endif

143: endwhile

144:end

172: if *total msg* = −(*sent* − *recv*)

173: ∧ *nb ack* = *P* then

174: *trm status* := *ST OP* ;

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).

* 1. *Correctness and complexity*

Our distributed Bes resolution algorithm is based on the theory of boolean graphs underlying the sequential algorithms [[2](#_bookmark16),[26](#_bookmark36)]. 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 · |*E*|· (*P* − 1)*/P* ), the worst-case being obtained with two messages (expansion and stabiliza- tion) 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.

# 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](#_bookmark28)] devel- oped using the Open/Cæsar environment [[12](#_bookmark22)]. Hence we immediately obtained a distributed version of the Bisimulator [[20](#_bookmark28)] on-the-fly equivalence checker of the Cadp verification toolbox [[13](#_bookmark23)], 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 *Xp,q* denot- ing 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 re- lation (see the table in Section [2](#_bookmark1)). Note that for weak equivalence relations (branching, observational, *τ* ∗*.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](#_bookmark11)], which is designed to be a reference criterion for scientific assessment of algorithms and tools operating on large graphs, such as dis- tributed 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.

* 1. *Speedup*

One way to quantify the efficiency of a parallel algorithm is to compute the absolute speedup *S* = *T*1*/Tp* by using as baseline the uniprocessor time *T*1 for the best known uniprocessor (sequential) algorithm, and the time *Tp* with *P* workers. Fig. [3](#_bookmark7) 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 *MR*, 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 sys- tem behavior (*protocol* ) and its external behavior (*service*), which correspond here to *M* and *MR*; (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 *MR*. We also per- formed 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](#_bookmark7)(*a*) shows the speedups obtained for strong equiv- alence checking with distributed Bisimulator on a set of examples, ordered by increasing sizes, from 9*.*757 · 103 states and 24*.*352 · 103 transitions (*dle*10)

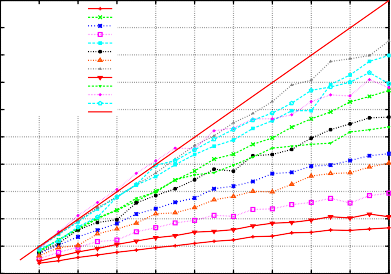
to 8*.*082 · 106 states and 42*.*933 · 106 transitions (*vasy* 8082 42933). Strong equivalence is well-suited for distribution: there is very few time spent in the front-end (no transitive closure on *τ* -transitions needed), and curves show linear speedups from low (still better than the sequential times) to nearly

optimal. Moreover, speedup gets better when the Lts size increases. For

example, the sequential check of experiment *BRPm*3*n*30 (Bounded Retrans- mission Protocol with 3 retransmissions and packet length 30, i.e. 5*.*957 · 106 states, 9*.*225 · 106 transitions) took 332.53 seconds, whereas the parallel check with 13 workers took 29.06 seconds (speedup 11.5).

τ ∗.a and safety equivalences. Fig. [3](#_bookmark7)(*b*) shows the speedups obtained for *τ* ∗*.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.

20



dle10 vasy\_65\_2621 vasy\_66\_1302

dle03 vasy\_157\_297

b57 vasy\_574\_13561 vasy\_720\_390

fw6 BRPm3n25 BRPm3n30

vasy\_8082\_42933 Ideal speedup

18

16

14

12

Speedup

10

8

6

4

2

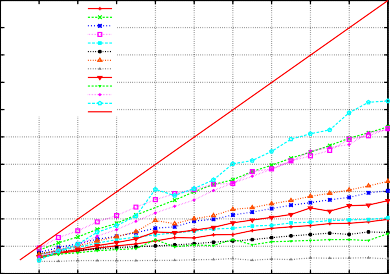
0

0 2 4 6 8 10 12 14 16 18 20

Number of workers

1. Strong equivalence

20



vasy\_18\_73 scsiA vasy\_65\_2621 vasy\_157\_297 vasy\_164\_1619 cwi\_214\_684 vasy\_1112\_5290

b200

vasy\_4338\_15666 BRPm3n30

vasy\_6120\_11031 vasy\_8082\_42933 Ideal speedup

18

16

14

12

Speedup

10

8

6

4

2

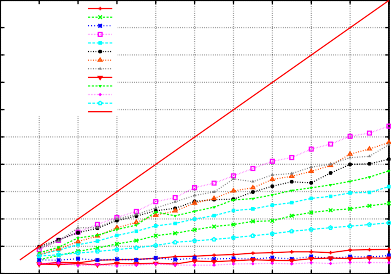
0

0 2 4 6 8 10 12 14 16 18 20

Number of workers

1. *τ* ∗*.a* equivalence

20



vasy\_18\_73 scsiSp BRPm3n4

vasy\_65\_2621

dle18 vasy\_157\_297 vasy\_386\_1171 vasy\_574\_13561 vasy\_720\_390 vasy\_1112\_5290

b200

vasy\_8082\_42933 Ideal speedup

18

16

14

12

Speedup

10

8

6

4

2

0

0 2 4 6 8 10 12 14 16 18 20

Number of workers

1. 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](#_bookmark7)(*c*) shows the speedups obtained for observational equivalence (branching equivalence shows a similar behavior). Con- trary to *τ* ∗*.a* and safety equiva- lences, the Bess encoding observa- tional and branching equivalences are much larger, and therefore dis- tributed 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 observa- tions can be drawn w.r.t. the nature of Ltss being checked. Three factors influence the performance of dis- tributed Bisimulator: size of Ltss, percentage of *τ* -transitions, and de- gree of nondeterminism. Hence, when neither *τ* -transitions nor non- determinism 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 *τ* ∗*.a* and safety equivalences (because of expensive front-end computations), but still good speedups for strong and observational equivalences (because of impor-

tant Bes sizes), as illustrated by experiments *BRPm*3*n*30 on Fig. [3](#_bookmark7). 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 *b*57 on Fig. [3](#_bookmark7)(*a*). Weak equivalences either could not terminate (e.g., observa- tional equivalence for *b*57), or they showed no speedup (e.g., *τ* ∗*.a* and safety equivalence for *b*200).

* 1. *Scalability*

Interesting insights into DSolve characteristics are provided by the above ex- perimental measures together with the scalability results shown on Fig. [4](#_bookmark8).

180

160

140

120

100

Time (sec)

80

60

40

20

0

p=3 p=5 p=7 p=11

p=20

0 2e+06 4e+06 6e+06 8e+06 1e+07 1.2e+07 1.4e+07

Problem size (transitions)

Each curve on Fig. [4](#_bookmark8) represents the time needed for experiment *BRPm*3*nK* (Bounded Retransmis- sion 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-

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

adapted to increases in problem size,

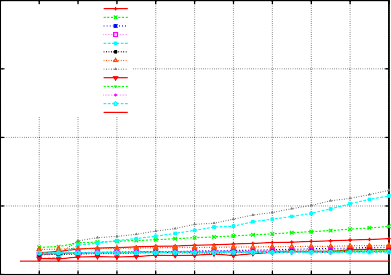
making an efficient use of memory and Cpu. As for another large example, DSolve handles the strong equivalence checking of experiment *b*200 (Alter- nating Bit Protocol with 200 different messages), whose generated Bes size is

2*.*4 · 108 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 · 107 variables).

* 1. *Memory*

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

20



dle10 vasy\_65\_2621 vasy\_66\_1302

dle03 vasy\_157\_297

b57 vasy\_574\_13561 vasy\_720\_390

fw6 BRPm3n25 BRPm3n30

vasy\_8082\_42933 Ideal memory ratio

15

Total par. memory / seq. memory

10

5

0

0 2 4 6 8 10 12 14 16 18 20

Number of workers

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

However, memory limitation of exist- ing sequential algorithms is the main motivation for distribution. Fig. [5](#_bookmark9) sustains by practical experiments that DSolve makes an efficient use of mem- ory. It presents results obtained for strong equivalences on a dozen of Vlts benchmarks sorted by increasing size, from 9 · 103 states, 25 · 103 transitions to 8·106 states, 43·106 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 con- sumption and corresponding sequential memory consumption, that is hardly increasing. The bigger is the Lts to be checked, the lower is the ratio.

# 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 verifica- tion engine in the distributed version of Bisimulator, an on-the-fly equivalence checker developed within the Cadp toolbox [[13](#_bookmark23)] using the Open/Cæsar envi- ronment for Lts exploration [[12](#_bookmark22)]. 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 inte- grated in the generic Cæsar Solve library [[20](#_bookmark28)], 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 us- ing Cæsar Solve, such as alternation-free *µ*-calculus model-checking [[20](#_bookmark28)] and *τ* -confluence reduction [[23](#_bookmark33)]. We also plan to extend Bisimulator with other equivalence relations, such as Markovian bisimulation [[15](#_bookmark25)].

# 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. Crubill´e. 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.

1. 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.
2. A. Bouajjani, J-C. Fernandez, S. Graf, C. Rodr´ıguez, and J. Sifakis. Safety for Branching Time Semantics. In *Proc. of 18th ICALP*, LNCS vol. 510. Springer Verlag.
3. R. Cleaveland and B. Steffen. Computing Behavioural Relations, Logically. In *Proc. of the 18th ICALP*, LNCS vol. 510, pp. 127–138. Springer Verlag.
4. 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.
5. 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.
6. 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.

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. A. Mader. *Veriﬁcation of Modal Properties Using Boolean Equation Systems*. VERSAL 8, Bertz Verlag, Berlin, 1997.
7. 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.

1. 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.
2. F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2:161– 175, 1987.
3. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
4. 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.

1. D. Park. Concurrency and Automata on Infinite Sequences. In P. Deussen, editor, *Theoretical Computer Science*, LNCS vol. 104, pp. 167–183. Springer Verlag.
2. 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.
3. 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.