Electronic Notes in Theoretical Computer Science 89 No. 1 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 15 pages

# Distributed Branching Bisimulation Reduction of State Spaces

Stefan Blom [1,3] Simona Orzan [2,4]

*Department of Software Engineering, CWI*
*Amsterdam, The Netherlands*

**Abstract**

Enumerative model checking tools are limited by the size of the state space to which they can be applied. Reduction modulo branching bisimulation usually results in a much smaller state space and therefore enables model checking of much larger state spaces. We present an algorithm for reducing state spaces modulo branching bisimulation which is suitable for distributed implementation. The target architecture is a cluster with a high bandwidth interconnect. The algorithm is based on partition refinement and it works on transition systems which contain cycles of invisible steps, without eliminating strongly connected components first. To avoid fine grained parallelism, the algorithm refines the whole partition instead of just a single block in the partition. We prove correctness and also present some experimental results obtained with single threaded and distributed prototypes.

## 1 Introduction

The size of systems that enumerative verification tools can handle is traditionally very small, and many interesting applications remain out of its scope. The ways to overcome this situation are the use of symbolic model checking and/or build more powerful tools. In the context of the latter approach, parallelization (use of shared memory machines) and distribution (use of clusters of workstations) of verification algorithms is an attractive line to follow. We mention some work done in this direction on both enumerative and symbolic tools: parallelization of the Mur$\phi$ verifier [15], distribution of the model checker UPPAAL [2], parallel state space generation [8], distributed LTL model checking [1], parallel $\mu$-calculus model checking [5].

[1] Email: sccblom@cwi.nl
[2] Email: simona@cwi.nl
[3] Partially funded by the "Systems Validation Centre" project.
[4] Funded by the STW-project CES.5009.

This paper presents a distributed message-passing algorithm for state space reduction modulo branching bisimulation, following previous work on distribution of strong bisimulation reduction algorithms [3], [4]. It is designed for a cluster of workstations, which is the most common and cheap architecture able to offer large memory and processing capabilities.

The most commonly used algorithm for branching bisimulation is the one of Groote and Vaandrager [10]. It is a very good algorithm, but there are two reasons why one does not really want to use it for developing a distributed tool. First, the natural parallelism in the algorithm is very fine grained, which is a bad idea on the cluster. The often large message latency leads to unacceptable performance. The second reason is that the Groote-Vaandrager algorithm works on transition systems that do not have cycles of silent steps. Cycle elimination requires detection of strongly connected components, which is a difficult problem to solve distributedly, although sequentially the well known Tarjan algorithm [16] solves it in linear time. Our current distributed algorithm does not rely on the absence of $\tau$ cycles, but we learn from sequential studies that it would perform better on a cycle-free state space. Therefore, it is interesting future work to integrate an initial distributed cycle elimination phase.

Stuttering equivalence on Kripke structures (see [6]) is similar to branching bisimulation on labeled transition systems. However, there are two differences. First, in Kripke structures the states are labeled instead of the edges. Second, stuttering equivalence distinguishes states where an infinite sequence of invisible steps is possible from states where such a sequence is impossible. Nevertheless, the algorithm presented by Browne, Clarke and Grumberg and our own algorithm are similar in the sense that both algorithms are partition refinement algorithms and apply more or less the same refinement strategy. However, the way in which the refinements are computed is different. More precisely, the Browne-Clarke-Grumberg algorithm calls for explicit computation of the transitive reflexive closure of silent steps whereas our algorithm avoids doing so.

**Overview.** The paper is organized as follows. Section 2 revisits some basic notions. Section 3 explains the theory behind our algorithms. Also, the correctness of our partition refinement strategy is proven. The single threaded and distributed implementations are commented in section 4 and their performance is discussed in section 5. We conclude (section 6) with a short overview and a discussion of future work.

## 2 Preliminaries

In this section we fix a notation for labeled transition systems, we recall the definition of branching bisimulation [17] and we introduce the terminology of partition refinement.

We consider transition systems with anonymous/unlabeled states and la-

beled edges. We use a fixed set of labels $\mathsf{Act}$. The silent action $\tau$ is a member of $\mathsf{Act}$.

**Definition 2.1** A labeled transition system ($LTS$) is a triple $(S, \rightarrow, s_0)$, consisting of a set of states $S$, a transition relation $\rightarrow \subseteq S \times \mathsf{Act} \times S$ and an initial state $s_0 \in S$.

We use the following notations:

$s \xrightarrow{a} t$ shorthand for $(s,a,t) \in \rightarrow$ ;
$\xrightarrow{a}\!\!\!\!\!\rightarrow$ the transitive reflexive closure of $\xrightarrow{a}$ ;
$\xrightarrow[R]{a} = R \cap \xrightarrow{a}$ for any equivalence relation $R$ ;
$\xrightarrow[R]{a}\!\!\!\!\!\rightarrow$ the transitive reflexive closure of $\xrightarrow[R]{a}$ .

**Definition 2.2** [branching bisimulation] Given a labeled transition system $\mathcal{S} \equiv (S, \rightarrow, s_0)$, a relation $R \subseteq S \times S$ is a branching bisimulation if $R$ is symmetric and $\forall s_1, s_2, t_1 \in S$ :

$$s_1 \; R \; s_2 \wedge s_1 \xrightarrow{a} t_1 \Longrightarrow \begin{cases} a \equiv \tau \wedge t_1 \; R \; s_2 \\ \vee \\ \exists s_2', t_2 \in S : s_2 \xrightarrow{\tau}\!\!\!\!\!\rightarrow s_2' \xrightarrow{a} t_2 \wedge s_1 \; R \; s_2' \wedge t_1 \; R \; t_2 \end{cases}$$

For $s, t \in S$, such that a branching bisimulation $R$ exists such that $s \; R \; t$, we write $s \underline{\leftrightarrow}_b t$.

**Lemma 2.3** *If $s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} s_2 \cdots s_n$ and $s_0 \underline{\leftrightarrow}_b s_n$ then $s_0 \underline{\leftrightarrow}_b s_i$, for all $i : 0 \leq i \leq n$.*

**Proof.** Follows from the stuttering lemma in [17].

□

**Definition 2.4** [partition] Given a set $S$, $\pi$ is a partition of $S$ if

$$\bigcup \pi = S \text{ and } \forall S', S'' \in \pi : S' \neq S'' \Longrightarrow S' \cap S'' = \emptyset \ .$$

A partition $\pi_1$ is a refinement of a partition $\pi_2$ if

$$\forall S_1 \in \pi_1 : \exists S_2 \in \pi_2 : S_1 \subset S_2 \ .$$

The elements of a partition are referred to as *blocks*. If $\pi$ is a partition then by $\pi(x)$ we denote the unique block $B$, such that $x \in B$. We view a partition $\pi$ as a relation, by abbreviating $\pi(x) = \pi(y)$ as $x \; \pi \; y$. So $\xrightarrow[\pi]{\tau}\!\!\!\!\!\rightarrow$ stands for a sequence of 0 or more $\tau$-steps within a block of $\pi$.

Given a LTS $\mathcal{S} \equiv (S, \rightarrow, s_0)$, let

$$\pi^b = \{\{s' \in S \mid s \underline{\leftrightarrow}_b s'\} \mid s \in S\} \ .$$

The LTS with the minimal number of states that is branching bisimilar to $\mathcal{S}$ is

$$\mathcal{S}^b \equiv (\pi^b, \{(\pi^b(s), a, \pi^b(t)) \mid s \xrightarrow{a} t \wedge (s \underline{\leftrightarrow}_b t \Longrightarrow a \neq \tau)\}, \pi^b(s_0)) \ .$$

## 3 Signature Refinement Theory

Our approach to the problem of state space minimization modulo branching bisimulation is inspired by previous work on strong bisimulation minimization [3], [4] and exploits the same basic idea: partition refinement based on *signature* computation.

In the remainder of the paper we work with a fixed LTS $(S, \rightarrow, s_0)$.

**Definition 3.1** [signature refinement]

$$\begin{aligned}
\mathrm{sig}(\pi) &: s \mapsto \{(a, \pi(t)) \mid \exists s' : s \xrightarrow[\pi]{\tau\!\!\!\!\!\!\rightarrow} s' \xrightarrow{a} t \wedge (a \neq \tau \vee \pi(s) \neq \pi(t))\} \ ; \\
\mathrm{sigref}(\pi) &= \{\{s' \in S \mid \mathrm{sig}(\pi)(s) = \mathrm{sig}(\pi)(s')\} \mid s \in S\} \ ; \\
\pi^0 &= \{S\} \ ; \\
\pi^{n+1} &= \mathrm{sigref}(\pi^n) \ .
\end{aligned}$$

A partition $\pi$ is *stable* if $\mathrm{sigref}(\pi) = \pi$.

The signature refinement algorithm iteratively computes $\pi^{n+1}$ (starting of course with $\pi^0$) until the stable partition is reached. We devote the rest of this section to thoroughly proving that this procedure correctly computes the minimal branching bisimulation. For this, the following three properties are necessary: the refinement steps must yield refinements; the refinement steps must keep bisimilar states in the same block; and a stable partition must be a bisimulation. First, let us see that every $\pi^{n+1}$ is a refinement of $\pi^n$.

**Lemma 3.2** $\forall n :$ *$\pi^{n+1}$ is a refinement of $\pi^n$* .

**Proof.** By induction on $n$. As induction hypothesis suppose that for all $i < n$, we have that $\pi^{i+1}$ is a refinement of $\pi^i$. We must show that $\pi^{n+1}$ is a refinement of $\pi^n$. Given $s, t$, such that $\pi^n(s) \neq \pi^n(t)$. There exists $k < n$ such that $\pi^k(s) = \pi^k(t)$ and $\pi^{k+1}(s) \neq \pi^{k+1}(t)$. This means that $\mathrm{sig}(\pi^k)(s) \neq \mathrm{sig}(\pi^k)(t)$. So without loss of generality, there exists $(a, B) \in \mathrm{sig}(\pi^k)(s)$, such that $(a, B) \notin \mathrm{sig}(\pi^k)(t)$. Let us define two sets of blocks:

$$\begin{aligned}
T_1 &= \{u \in \pi^k(s) \mid (a, B) \in \mathrm{sig}(\pi^k)(u)\} \\
T_2 &= \{u \in \pi^k(s) \mid (a, B) \notin \mathrm{sig}(\pi^k)(u)\}
\end{aligned}$$

We make the following remarks:

$$T_1 \cap T_2 = \emptyset \wedge \forall u \in \pi^k(s) : \pi^{k+1}(u) \subset T_1 \vee \pi^{k+1}(u) \subset T_2$$

$$\forall u \in T_2 : \neg \exists u' \in B : u \xrightarrow{a} u' \tag{1}$$

$$\forall u \in T_2 : \neg \exists u' \in T_1 : u \xrightarrow{\tau} u' \tag{2}$$

$$\exists s_1, \cdots, s_q \in T_1 : s \xrightarrow{\tau} s_1 \xrightarrow{\tau} \cdots s_q \xrightarrow{a} s' \in B$$

There are two cases possible:

- If $s_1, \cdots, s_q \in \pi^n(s)$ then $(a, \pi^n(s')) \in \mathrm{sig}(\pi^{n+1})(s)$.
  Suppose that $\mathrm{sig}(\pi^{n+1})(s) = \mathrm{sig}(\pi^{n+1})(t)$. Then $t \xrightarrow[\pi^n]{\tau\!\!\!\!\!\!\rightarrow} t' \xrightarrow{a} t'' \in \pi^n(s')$. But since $t' \in T_2$ and $t'' \in B$, this contradicts 1.

- If $s_1, \cdots, s_r \in \pi^n(s)$ and $s_{r+1} \notin \pi^n(s)$ then $(\tau, \pi^n(s_{r+1})) \in \mathrm{sig}(\pi^{n+1})(s)$. Suppose that $\mathrm{sig}(\pi^{n+1})(s) = \mathrm{sig}(\pi^{n+1})(t)$. Then $t \xrightarrow[\pi^n]{\tau\!\!\!\!\rightarrow\!\!\!\!\rightarrow} t' \xrightarrow{\tau} t'' \in \pi^n(s_{r+1})$. We also have that $t' \in T_2$ and $t'' \in T_1$, which contradicts 2.

□

The following lemma states that refining a partition where bisimilar states are in the same block results in a partition where bisimilar states are still in the same block. (Note that saying that $\pi^b$ is a refinement of $\pi$ is equivalent to saying that bisimilar states are in the same block of $\pi$.)

**Lemma 3.3** *Given a partition $\pi$, if $\pi^b$ is a refinement of $\pi$ then $\pi^b$ is a refinement of* $\mathrm{sigref}(\pi)$.

**Proof.** We must show that for any $s_0, t_0$ such that $s_0\ \pi^b\ t_0$, we have that $s_0\ \mathrm{sigref}(\pi)\ t_0$. This means that we have to show that $\mathrm{sig}(\pi)(s_0) = \mathrm{sig}(\pi)(t_0)$, given that $s_0\ \pi'\ t_0$. Due to symmetry it suffices to show that $\mathrm{sig}(\pi)(s_0) \subseteq \mathrm{sig}(\pi)(t_0)$.

Given $(a, B) \in \mathrm{sig}(\pi)(s_0)$, we can find $s_0 \xrightarrow[\pi]{\tau} s_1 \xrightarrow[\pi]{\tau} \cdots s_n \xrightarrow{a} s'$, such that $\pi(s) = B$ and $(a \neq \tau \vee \pi(s_0) \neq \pi(s))$.

Given $t_i$ such that $t_i\ \pi^b\ s_i$ and $i < n$, we define $t_{i+1}$ by distinguishing two cases:

- If $s_{i+1}\ \pi^b\ s_i$ then let $t_{i+1} = t_i$. We have that $s_{i+1}\ \pi^b\ t_{i+1}$ and that $t_i \xrightarrow[\pi]{\tau\!\!\!\!\rightarrow\!\!\!\!\rightarrow} t_{i+1}$.
- Otherwise, due to bisimulation and the stuttering lemma we can find $t_{i+1}$ such that $t_i \xrightarrow[\pi^b]{\tau\!\!\!\!\rightarrow\!\!\!\!\rightarrow} t \xrightarrow{\tau} t_{i+1}$ and $s_{i+1}\ \pi^b\ t_{i+1}$. So for some $t'_i$, we have $t_i \xrightarrow[\pi^b]{\tau\!\!\!\!\rightarrow\!\!\!\!\rightarrow} \rightarrow t'_i \xrightarrow{\tau} t_{i+1}$. We now have that $t'_i\ \pi^b\ t_i\ \pi^b\ s_i\ \pi\ s_{i+1}\ \pi^b\ t_{i+1}$ .. Because $\pi^b$ is a refinement of $\pi$, we can conclude that $t'_i\ \pi\ t_{i+1}$. Thus, we have that $t'_i \xrightarrow{\tau} t_{i+1}$. Again because $b(\pi)$ is a refinement of $\pi$, we have that $t_i \xrightarrow[\pi']{\tau\!\!\!\!\rightarrow\!\!\!\!\rightarrow} t'_i$, so we have $t_i \xrightarrow[\pi']{\tau\!\!\!\!\rightarrow\!\!\!\!\rightarrow} t_{i+1}$.

If $\pi(s_0) \neq \pi(s')$ then $\pi^b(s_n) \neq \pi^b(s')$, because $\pi(s_0) = \pi(s_n)$ and $\pi^b$ is a refinement of $\pi$. So we have $a \neq \tau \vee \pi^b(s_0) \neq \pi^b(s')$. We also have $s_n\ \pi^b\ t_n$, so by definition of bisimulation and the stuttering lemma there exists $t'$ such that $t_n \xrightarrow[\pi^b]{\tau\!\!\!\!\rightarrow\!\!\!\!\rightarrow} \xrightarrow{a} t'$ and $s'\ \pi^b\ t'$. As $\pi^b$ is a refinement of $\pi$ this implies that $t_n \xrightarrow[\pi]{\tau\!\!\!\!\rightarrow\!\!\!\!\rightarrow} \xrightarrow{a} t'$ and $s'\ \pi\ t'$. In turn this implies $t_0 \xrightarrow[\pi]{\tau\!\!\!\!\rightarrow\!\!\!\!\rightarrow} \xrightarrow{a} t'$, which implies that $(a, B) \in \mathrm{sig}(\pi)(t_0)$.

□

Finally, we need to establish that a stable partition is a branching bisimulation.

**Lemma 3.4** *If $\pi$ is a stable partition then $\pi$ is a branching bisimulation.*

**Proof.** Given $s\ \pi\ t$ and $s \xrightarrow{a} s'$. If $a = \tau$ and $s\ \pi\ s'$ then $s'\ \pi\ t'$. Otherwise $(a, \pi(s')) \in \mathrm{sig}(\pi)(s)$. Because the partition is stable we have $\mathrm{sig}(\pi)(s) =$

$\mathrm{sig}(\pi)(t)$, so for some $t'$ we have $t \xrightarrow[\pi]{\tau\ast} t' \xrightarrow{a} t''$ with $s' \;\pi\; t''$ and $s \;\pi\; t'$.

□

From these three lemmas, the correctness of the partition refinement algorithm for finite LTSs follows easily:

**Theorem 3.5** *Given a finite LTS, the following program computes $\pi^b$ in $\pi$:*

$\pi := \{S\}$
*repeat*
    $\pi' := \pi$
    $\pi := \mathrm{sigref}(\pi)$
*until* $\pi = \pi'$

**Proof.** After the $n^{\text{th}}$ iteration of the loop, the variable $\pi$ contains $\pi^n$. If the loop exits after $n$ iterations then the partition $\pi$ is stable. Due to Lemma 3.4 the resulting $\pi$ is a branching bisimulation. Due to Lemma 3.3 it must be $\pi^b$. From Lemma 3.2 we get that $\mathrm{sigref}(\pi)$ is a refinement of $\pi$. This means that if $\mathrm{sigref}(\pi)$ is not the same as $\pi$ then $\mathrm{sigref}(\pi)$ contains more blocks than $\pi$. As the number of blocks is limited by the number of states, termination of the loop is guaranteed.

□

For an LTS with $n$ states and $m$ transitions, the worst case complexity of our algorithm is $\mathcal{O}(n^2m)$ time and $\mathcal{O}(nm)$ space. This is much worse than the $\mathcal{O}(n(n+m))$ time and $\mathcal{O}(n+m)$ space complexity of the Groote-Vaandrager algorithm. However, we expect that for typical state spaces the expected complexity of both algorithms is $\mathcal{O}(\log(n)(n+m))$ time and $\mathcal{O}(n+m)$ space. Next we'll analyze the complexity of an example, which is near to the worst case.

**Example 3.6** Given a natural number $N$. Consider the the LTS with states $1, 1', 2, 2' \cdots, N, N'$, transitions $i \xrightarrow{a} i'$, $i+1 \xrightarrow{\tau} i$, $1 \xrightarrow{\tau} N$, $(i+1)' \xrightarrow{b} i'$ and initial state $N$. The signatures for this LTS are

$$\begin{aligned}
\mathrm{sig}(\pi^k)(i) &= \{(a, \pi^k(1')), \cdots, (a, \pi^k(N'))\} \\
\mathrm{sig}(\pi^k)(1') &= \emptyset \\
\mathrm{sig}(\pi^k)((i+1)') &= \{(b, \pi^k(i'))\}
\end{aligned}$$

and the partitions are

$$\begin{aligned}
\pi^0 &= \{\{1, 1', 2, 2' \cdots, N, N'\}\} \\
\pi^k &= \{\{1'\}, \cdots, \{k'\}, \{(k+1)', \cdots, N'\}, \{1, \cdots, N\}\}
\end{aligned}$$

This means that $N+1$ iterations are needed to get to a stable refinement of $\pi^0$. The cost of computing the signature of $i'$ is constant in each iteration because the signature size is constant. However, the cost of computing the signature of $i$ in the $k^{\text{th}}$ linearly grows with $k$ because the size of the signature

Table 1
Single threaded version of the algorithm.

```
reduce()
    for all states s do pi[s]:=0 end for
    repeat
        // compute signatures
        for all states s do sig[s]:=∅ end for
        for all transitions (s,a,t) do
            if not (a=τ and pi[s]=pi[t]) then insert(s,a,pi[t]) end if
        end for
        // reassign pi according to sig
        hashtable := ∅
        count:=0
        for all states s do
            if not sig[s] in keys(hashtable) then
                insert(hashtable,sig[s],count)
                inc(count)
            end if
        end for
        for all states s do pi[s]:=lookup(hashtable,sig[s]) end for
    until pi is stable

insert(t,a,ID)
    if not((a,ID) ∈ sig[t]) then
        sig[t]:=sig[t] ∪ {(a,ID)}
        for all s such that s -τ-> t and pi[s]=pi[t] do
            insert(s,a,ID)
        end for
    end if
```

is $k$. As we have $N$ signatures of each kind, we get time complexity $\mathcal{O}(N^3)$ and space complexity $O(N^2)$.

## 4 Signature Refinement Algorithms

In this section, we discuss a few algorithms for branching bisimulation minimization based on signature refinement. These algorithms are similar to those for strong bisimulation in [3] and [4]. So for details about data distribution and computation of partitions from signatures, we refer to those papers.

### *4.1 Single Threaded*

We now describe a single threaded implementation (depicted in Table 1) of the algorithm outlined in Theorem 3.5. To represent partitions we assign a unique (integer) identifier to each block and then represent the partition as an

Table 2
Distributed version of the algorithm.

```
1   reduce()
2       for all states s parallel do pi[s]:=0 end for
3       repeat
4           for all states s parallel do
5               sig[s]:={(a, pi[t]) | s -a-> t ∧ (a ≠ τ ∨ pi[s] ≠ pi[t])}
6               pred[s] := {t | t -τ-> s ∧ pi[s] = pi[t]}
7           end for
8           new:=sig
9           repeat
10              for all states s parallel do nextnew[s]:=∅ end for
11              for all states s parallel do
12                  for all states t in pred[s] do
13                      nextnew[t]:=nextnew[t] ∪ (new[s] \ sig[t])
14                      sig[t]:=sig[t] ∪ new[s]
15                  end for
16              end for
17              new:=nextnew
18          until ∀s : new[s] = ∅
19          reassign pi according to sig
20      until pi is stable
```

array of block identifier, which is indexed by states. Thus, the initial partition can be represented as an array of zeros. The definition of signature consider transitions of all states reachable by $\tau$-steps within blocks. Explicitly computing sets of reachable states should be avoided because this would require too much time and memory. So instead of starting at a state and searching the reachable states for information, we start with the information and propagate it back along the $\tau$-steps within blocks using a depth first traversal. Then once all signatures have been computed, unique identifiers are assigned to signatures and from these identifiers the next partition is built. Based on the number of identifiers, we can decide if the partition is stable and iterate if necessary.

### *4.2 Distributed*

Let us now see how a distributed version of the algorithm can be implemented.

The single threaded algorithm uses sequential depth first traversal for propagating signature information. As the order of signature propagation is irrelevant, we chose breadth first propagation for the distributed algorithm in Table 2. In order to present the global picture in a clear way, we write it as a shared memory algorithm and abstracts away the actual location of data.

In our distributed memory implementation, the states are divided among

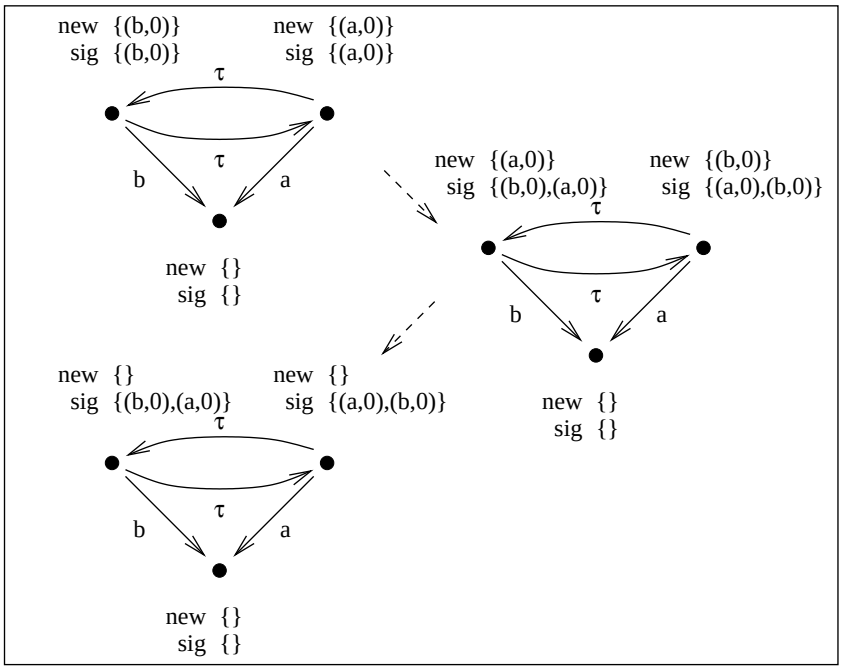


Fig. 1. The iterations of signature computation.

a set of workers and we may apply the function owner to a state to get the worker which owns the state. Each worker stores the parts of the arrays corresponding to its owned states. This means that the underlined references to arrays are potentially remote references. There are remote references to three arrays: `pi`, `newsig` and `sig`. In our distributed implementation remote references to `pi` are made local by copying the relevant parts of `pi`. That is, every worker keeps not only the `pi` data for its owned states, but also for the successor states of its owned states. Remote access to `newsig` and `sig` is solved in a different way. Instead of letting the owner of the `new` array perform the assignments, we let the owner of the `new` array send a message containing `s`, `t`, and `new[s]` to the owner of the `newsig` and `sig` arrays. Upon receiving such a message the owner of the `newsig` and `sig` arrays will perform the assignments. This is correct because the order of the assignments to `nextnew` and `sig` does not matter as long as they are atomic (no other assignments carried out in between).

Message passing replacements for lines 4-7 and 11-16 can be found in tables 3 and 4, respectively. These replacements consist of multiple threads which are separated by ‖. The receive statement blocks until there is a message returning true or until there are no further messages in the system and no further sends can be initiated in which case they return false. For performance reasons the actual implementation buffers a few kB worth of small messages before sending.

In Fig. 1 we have illustrated the process of signature computation. When the computation starts every state is in partition 0. Initialy, the signature sets contain the transition,ID pairs which are possible in every state and the new

Table 3
Message replacement for lines 4-7.

```
for all states t parallel do sig[t]:=∅ end for
for all states t parallel do
    for all s,a such that s -a-> t do
        send ["pi",s,a,t,pi[t]] to owner(s)
    end for
end for
||
while receive ["pi",s,a,t,id]
    pi[t]:=id
    if a = τ and pi[s] = pi[t] then
        send["pred",t,s] to owner(t)
    else
        sig[s]:=sig[s]∪{(a,ID)}
    end if
end while
||
while receive ["pred",t,s]
    pred[t]:=pred[t]∪{s}
end while
```

sets are set to the same value. In every iteration, the new sets are forwarded along the inverse of the invisible $\tau$-steps, added to the signature sets and the new elements are added to the new sets. So for example in the first iteration (b,0) is sent along the top edge, inserted in the signature of the right state and because it is new it is also put into new. In the second iteration it is sent along the bottom $\tau$-edge and inserted, but because it was already present it is not added to new. This forwarding continues until the new sets are empty.

We have omitted the code for reassigning pi according to sig, because the distributed assignment works the same as the single threaded, with the exception that hashtable lookups are performed by means of message passing rather than by means of memory access.

## 5 Experiments

We have built prototype implementations of both sequential and distributed branching bisimulation minimization algorithms. The distributed implementation uses MPI (Message Passing Interface) for communication. The tests were made on a cluster of 8 dual AMD Athlon MP1600+ machines with 2G memory each, running Linux and connected by gigabit ethernet.

The examples used are the state space of the Firewire Link Layer protocol [11] (1394-LL), the Firewire Leader Election protocol [14] with 14 nodes (1394-LE), a cache coherence protocol [13] (CCP-2p3t ), and a distributed lift system

Table 4
Message passing code for lines 11 - 16.

```
for all states t parallel do
    for all states s in pred[t] do
        for all (a,ID) in new[t] do
            send ["new",s,a,ID] to owner(s)
        end for
    end for
end for
||
while receive ["new",s,a,ID]
    if not (a,ID) in sig[s] then
        sig[s]:=sig[s]∪{(a,ID)}
        nextnew[s]:=nextnew[s]∪{(a,ID)}
    end if
end while
```

Table 5
A comparison of sequential implementations.

| problem | size | bcg-min 1.4 | lts-min cycle | lts-min dfs | lts-min iter | lts-min mark | number of iterations |
|---|---|---|---|---|---|---|---|
| | states<br>transitions | time<br>mem | time<br>mem | time<br>mem | time<br>mem | time<br>mem | |
| 1394-LL | 0.37 $10^6$<br>0.68 $10^6$ | 2.27s<br>2.2M | 0.98s<br>2.8M | 0.97s<br>3.5M | 2.5s<br>3.5M | 1.16s<br>4M | 6 |
| lift5 | 2.2 $10^6$<br>8.7 $10^6$ | 2m42s<br>174M | 1m18s<br>108M | 1m20s<br>152M | 9m03s<br>116M | 2m30s<br>410M | 16 |
| 1394-LE | 2.5 $10^6$<br>17.6 $10^6$ | 1m18s<br>316M | 1m11s<br>220M | 1m08s<br>411M | 1m25s<br>220M | 1m14s<br>340M | 2 |
| CCP-2p3t | 7.8 $10^6$<br>59 $10^6$ | 19m26s<br>1051M | 22m50s<br>736M | 62m52s<br>968M | -<br>- | -<br>- | 46 |

with 5 and 6 legs [9] (lift5, lift6).

### *5.1 Single-threaded implementations*

In order to investigate possibilities, we have implemented four variants of the branching reduction scheme based on signatures. The one called *cycle* eliminates the $\tau$ cycles before starting the iterations series, while *dfs* and *iter* do not. Further, *iter* computes the signatures by performing propagation sub-iterations, as done in the distributed implementation. Finally, *mark* employs a marking procedure that proved helpful in the strong bisimulation reduction

case [4]. Its basic idea is to restrict the signature recomputation effort of an iteration to those signatures that changed for sure.

Table 5 displays the total run times (read, reduction and write) of these implementations and the maximum amount of memory occupied. To show that our signature refinement scheme is comparable to the block based refinement scheme, we include *bcg_min* (the reduction tool belonging to the CADP toolset [7]) in this brief comparison. For the CCP-2p3t example, the *iter* implementation takes too much time and *mark* runs out of memory. The reason for the *iter* implementation taking too much time was diagnosed as an inefficient implementation of one sub-routine. Thus, we could avoid making the same mistake in the distributed implementation. We stopped the single threaded tool after more than 24 hours, with only half the job completed. The distributed tool completes the task in roughly 12 minutes on 16 processors.

The first conclusion of this sequential study is that the signature based reduction algorithm works for branching bisimulation. The cycle elimination seems to be an advantage (cycle vs. dfs), therefore it might be interesting to use it also in the distributed version. From the performance data of *iter* it is clear that there is no serious efficiency loss by using mechanisms specific to a distributed implementation. The marking procedure does not deliver spectacular improvements, in fact no improvements at all. The explanation is that this procedure is efficient in the iterations when few changes happen – typically towards the end of the reduction process. But the branching bisimulation algorithm usually stabilizes in a rather small number of iterations, therefore the administrative penalties paid in the first iterations are not regained later. (1394-LL, for instance, stabilizes in 73 iterations for strong and in 6 for branching; lift5 in 86 for strong, 16 for branching; 1394-LE in 51 for strong and only 2 for branching.)

### *5.2 Distributed implementation*

Figure 5.2 shows the speedup of the distributed prototype (we name it DBBR - Distributed Branching Bisimulation Reduction) when run with 8,10,12,14 and 16 processors. For comparison, we also show the speedup of a similar distributed tool developed for strong bisimulation reduction, DSBR (Distributed Strong Bisimulation Reduction). That tool uses essentially the same idea – compute all states' signatures and perform block splitting based on them – and is described in detail in [3].

We also gathered data on the approximate total memory used by the two distributed tools when reducing lift6. DBBR 's memory needs grow slowly from 7232M for 8 processors to 7656M for 16 processors, while DSBR used 5752M (8 processors) up to 5958M (16). This shows that the memory use per worker decreases almost linearly with the number of workers.

As mentioned in the previous subsection, the stable partition with respect to branching bisimulation is most of the time reached in (a lot) less iterations

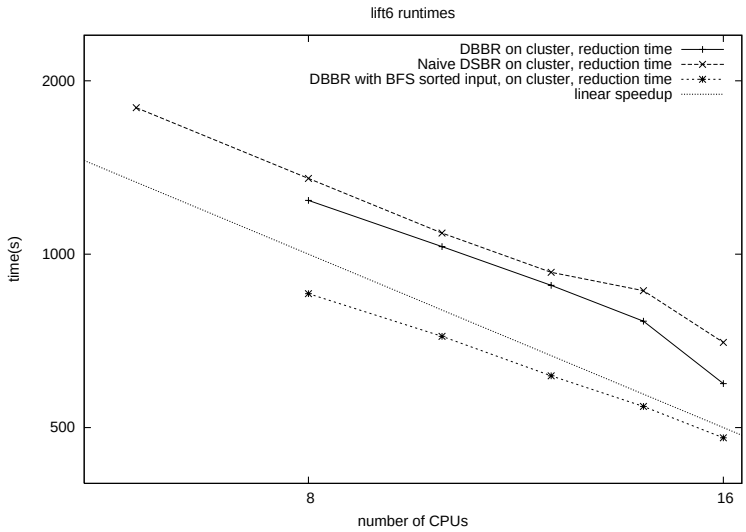


Fig. 2. Speedup for the reduction of lift6 (34 mil. states, 165 mil. transitions)

than the stable partition with respect to strong bisimulation. This explains why, although a DBBR iteration takes longer than a DSBR one, DBBR needs on the whole less time. As regard to memory use, DBBR is in all cases more expensive than DSBR . This is due to two factors. Firstly, the signatures for the branching bisimulation case are in general larger, since the signatures of a state $x$ must include the signatures of all states reachable by silent steps. And secondly, our current implementation is a first prototype, not yet optimized for memory use. We expect that a more careful implementation will visibly reduce this difference.

A more interesting comparison is between the run times of DBBR for random and for sorted input. (Random meaning a copy without caring about the order and sorted means sorted into the same BFS order written by our distributed state space generation tool.) The data indicates a much better performance in the case when the distribution of the LTSs states to the workers is done on BFS order. This means that we should investigate whether other orders exist, which can easily be computed and show even better performance.

## 6 Conclusion

The work presented here continues the series of distributed minimization algorithms started with [3], [4]. In this paper we considered *branching bisimulation* as reduction relation and we developed a signature based partition refinement algorithm for it, that works on LTSs with cycles of invisible steps. We proved its correctness, briefly described its implementation and showed by some experimental results that it scales up both in time and memory use.

As future work, we mention the possibility of introducing a distributed $\tau$-cycle elimination preprocessing phase, from which the efficiency of the current tool could benefit. Another possible direction is the development of

(distributed) signature refinement algorithms for minimization modulo weak bisimulation and safety equivalence. We also consider building a distributed model checker in the style of XTL (see [12]) in order to have a complete distributed model checking solution for our $\mu$CRL toolset.

## References


[1] Barnat, J., L. Brim and J. Stříbrná, *Distributed LTL model-checking in SPIN*, in: M. B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop (SPIN'01)*, LNCS **2057** (2001), pp. 200–216.

[2] Behrmann, G., T. Hune and F. Vaandrager, *Distributed timed model checking - How the search order matters*, in: A. Emerson and A. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, LNCS **1855** (2000), pp. 216–231.

[3] Blom, S. and S. Orzan, *A distributed algorithm for strong bisimulation reduction of state spaces*, in: L. Brim and O. Grumberg, editors, *Proceedings of PDMC'02*, ENTCS **68** (2002).

[4] Blom, S. and S. Orzan, *Distributed state space minimization*, in: F. Aagesen, T. Arts and W. Fokkink, editors, *Proceedings of FMICS'03*, ENTCS **80** (2003), to appear.

[5] Bollig, B., M. Leucker and M. Weber, *Parallel model checking for the alternation free $\mu$-calculus*, in: T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, LNCS **2031** (2001), pp. 543–558.

[6] Browne, M., E. Clarke and O. Grumberg, *Characterizing finite Kripke structures in propositional temporal logic*, Theoretical Computer Science **59** (1988), pp. 115–131.

[7] Fernandez, J.-C., H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu and M. Sighireanu, *CADP – a protocol validation and verification toolbox*, in: *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*, LNCS **1102** (1996), pp. 437–440.

[8] Garavel, H., R. Mateescu and I. Smarandache, *Parallel state space construction for model-checking*, in: M. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, LNCS **2057** (2001), pp. 217–234.

[9] Groote, J., J. Pang and A. Wouters, *Analysis of a distributed system for lifting trucks*, Journal of Logic and Algebraic Programming **55** (2003), pp. 21–56.

[10] Groote, J. and F. Vaandrager, *An efficient algorithm for branching bisimulation and stuttering equivalence*, in: M.S.Paterson, editor, *Proceedings of the 17th ICALP*, LNCS **443** (1990), pp. 626–638.

[11] Luttik, S., *Description and formal specification of the Link Layer of P1394*, in: I. Lovrek, editor, *Proceedings of the 2nd International Workshop on Applied Formal Methods in System Design*, 1997.

[12] Mateescu, R. and H. Garavel, *XTL: A meta-language and tool for temporal logic model-checking*, in: T. Margaria and B. Steffen, editors, *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT'98)*, number NS-98-4 in BRICS Notes Series, 1998.

[13] Pang, J., W. Fokkink, R. Hofman and R. Veldema, *Model checking a cache coherence protocol for a java DSM implementation*, in: *Proceedings of the 8th International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'03)* (2003).

[14] Romijn, J., *Model checking the HAVi leader election protocol*, Technical Report SEN-R9915, CWI (1999).

[15] Stern, U. and D. Dill, *Parallelizing the Mur$\phi$ verifier*, in: O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, LNCS **1254**, 1997, pp. 256–278.

[16] Tarjan, R., *Depth-first search and linear graph algorithms*, SIAM Journal of Computing **1** (1972), pp. 146–160.

[17] van Glabbeek, R. and W. Weijland, *Branching time and abstraction in bisimulation semantics*, Journal of the ACM **43(3)** (1996), pp. 555–600.