Distributed Branching Bisimulation Minimization by Inductive Signatures

Stefan Blom Jaco van de Pol
University of Twente, Formal Methods and Tools *
P.O.-box 217, 7500 AE, Enschede, The Netherlands
{sccblom,vdpol}@cs.utwente.nl

Technical Report TR-CTIT-09-37
October 2009

Abstract

We present a new distributed algorithm for state space minimization modulo branching bisimulation. Like its predecessor it uses signatures for refinement, but the refinement process and the signatures have been optimized to exploit the fact that the input graph contains no τ-loops.

The optimization in the refinement process is meant to reduce both the number of iterations needed and the memory requirements. In the former case we cannot prove that there is an improvement, but our experiments show that in many cases the number of iterations is smaller. In the latter case, we can prove that the worst case memory use of the new algorithm is linear in the size of the state space, whereas the old algorithm has a quadratic upper bound.

The paper includes a proof of correctness of the new algorithm and the results of a number of experiments that compare the performance of the old and the new algorithms.

This report is an extension of [10] with full proofs.

1 Introduction

The idea of distributed model checking of very large systems, is to store the state space in the collective memory of a cluster of workstations, and employ parallel algorithms to analyze the graph. One approach is to generate the graph in a distributed way, and on-the-fly (i.e. during generation) run a distributed model checking algorithm. This is what is done in the DiVinE toolset [4]. This is useful if the system is expected to contain bugs, because the generation can stop after finding the first bug.

Another approach is to generate the full state space in a distributed way, and subsequently run a distributed bisimulation reduction algorithm. The result is usually much smaller, and satisfies the same temporal logic properties. The minimized graph could be small enough to analyse with sequential model checkers. This approach is useful for certification, because many properties can be checked on the minimized graph. This paper contributes to the second approach.

*This work has been partially funded by the EU under grant number FP6-NEST STREP 043235 (EC-MOAN).

The process-algebraic way of abstracting from actions is to hide them by renaming them to the invisible action τ. To reason about equivalence of these abstracted models, branching bisimulation [21, 5] can be used. Because branching bisimulation is coarser than strong bisimulation, this leads to smaller state spaces modulo reduction.

Distributed minimization algorithms have been proposed in [8, 9] for strong bisimulation, and in [7] for branching bisimulation. These are signature-based algorithms, which work by successively refining the trivial partition, according to the (local) signature of states with respect to the previous partition.

The best-known sequential algorithm [14] for branching bisimulation reduction assumes that the state space has no τ-cycles. The idea is that any τ-cycles can be removed in linear time, by Tarjan's algorithm to detect (and eliminate) strongly connected components (SCC) [20]. Eliminating SCCs preserves branching bisimulation.

Because eliminating τ-cycles in distributed graphs seemed complicated, the algorithm in [7] works on *any* LTS, i.e. it doesn't assume the absence of τ-cycles. This generality came with a certain cost: signatures have to be transported over the transitive closure of silent τ-steps. For some cases this leads to increased time and memory usage.

Later, several distributed SCC detection (and elimination) algorithms have been developed [18, 17, 15, 3]. It has already been reported in [17] that running SCC elimination as a preprocessing step to the branching minimization algorithm of [7], reduces the overall time. Note that this gain was achieved even though the minimization algorithm doesn't assume that the input graph is τ-acyclic.

In this paper, we further improve this method, by exploiting the fact that the input graph of the minimization algorithm has no τ-cycles. Using this extra knowledge, we are able to develop a distributed minimization algorithm that runs in less time and memory.

At the heart of our improved method is a notion of *inductive signature*. Normally, during a round of signature computations, only the signatures of the previous round may be used. The basic idea of inductive signatures is that the new signature of a state may depend on the *current* signature of its $\xrightarrow{a}$-successors, provided a is guaranteed to terminate. We will first illustrate this notion for strong bisimulation, and then apply it to branching bisimulation, where τ is cycle-free, i.e. $\xrightarrow{\tau}$ is a terminating transition. Note that if all action labels are terminating, the graph is actually a directed acyclic graph, for which it is known that there is a linear algorithm for bisimulation reduction.

Overview. In the next section, we will explain the theory and prove the correctness of the improved signature bisimulation. In section 3, we explain how we turned the definition of inductive signature bisimulation onto a distributed algorithm and how we implemented it on top of the LTSmin toolset[1]. We show the results of running the tool on several problems in Section 4.

2 Theory

In this section, we start by recalling the basic definitions of LTS and bisimulation. Followed by the definitions of signature refinement from previous papers. Then we present inductive signatures for strong bismulation followed by inductive signatures for branching bisimulation. We end this section with the correctness proof for branching bisimulation.

[1] `http://fmt.cs.utwente.nl/tools/ltsmin/`

2.1 Preliminaries

First, we fix a notation for labeled transition systems and recall the definitions of strong bisimulation and branching bisimulation [21, 5]. Our transition systems are labeled with actions from a given set Act. The invisible action τ is a member of Act.

Definition 1 *(LTS) A labeled transition system (LTS) is a triple* $(S, \rightarrow, s^0)$*, consisting of a set of states* S*, transitions* $\rightarrow \subseteq S \times \mathsf{Act} \times S$ *and an initial state* $s^0 \in S$*.*

We write $s \xrightarrow{a} t$ for $(s, a, t) \in \rightarrow$, and use $\xrightarrow{a}^*$ to denote the transitive reflexive closure of $\xrightarrow{a}$.

Both strong and branching bisimulation can be defined in two ways. As a relation between two LTSs or as a relation on one LTS. We choose the latter.

Definition 2 (strong bisimulation) *Given an LTS* $(S, \rightarrow, s^0)$*. A symmetric relation* $R \subseteq S \times S$ *is a strong bisimulation if:*

$$\forall s, t, s' \in S : \ \forall a \in \mathsf{Act} : \ s\ R\ t \wedge s \xrightarrow{a} s' \Rightarrow \exists t' \in S : \ t \xrightarrow{a} t' \wedge s'\ R\ t' \ .$$

Definition 3 (branching bisimulation) *Given an LTS* $(S, \rightarrow, s^0)$*. A symmetric relation* $R \subseteq S \times S$ *is a branching bisimulation if:*

$$\forall s, t, s' \in S : \ \forall a \in \mathsf{Act} : \ s\ R\ t \wedge s \xrightarrow{a} s' \Rightarrow \begin{cases} (a \equiv \tau \wedge s'\ R\ t) \\ \vee \\ (\exists t', t'' \in S : \ t \xrightarrow{\tau}^* t' \wedge s\ R\ t' \wedge t' \xrightarrow{a} t'' \wedge s'\ R\ t'') \end{cases}$$

Two states $s, t \in S$ *are branching bisimilar (denoted* $s \leftrightarrow t$*) if there exists a branching bisimulation* R *such that* $s\ R\ t$*.*

For proving correctness, we will use a few properties:

Proposition 4 *Given an LTS:*

- *the relation* $\underline{\leftrightarrow}$ *is a branching bisimulation;*
- *if* R *is a branching bisimulation then* $R \subseteq \underline{\leftrightarrow}$*.*

For a proof see [21].

To talk about bisimulation reduction algorithms, we need the terminology of partition refinement. Given a set S.

- A set of sets $\{S_1, \cdots, S_N\}$ is a *partition* of S if $S = S_1 \cup \cdots \cup S_N$ and $\forall i \neq j : S_i \cap S_j = \emptyset$. Each set S_i is referred to as a *block* and must be non-empty.
- A partition $\{S_1, \cdots, S_N\}$ is a *refinement* of a partition $\{S'_1, \cdots, S'_M\}$ if $\forall i \exists j : S_i \subseteq S'_j$.
- Any partition $\{S_1, \cdots, S_N\}$ can be represented with an identity function $ID : S \rightarrow \mathbb{N}$, defined as $ID(s) = i$, if $s \in S_i$.

2.2 Signature Refinement

We continue with the previously published variant of signature refinement. Because many results are correct for finite LTSs only, we assume that both Act and all LTSs are finite for the remainder of the paper.

The signature of a state is computed with respect to a partition. Intuitively, the signature of a state is the set of possible moves (actions) that are possible in a state with respect to the paritition (represented by a number). Formally:

Definition 5

- *The set of signatures* Sig *is the set of* finite subsets *of* $\mathsf{Act} \times \mathbb{N}$.
- *A partition* π *of an LTS* $(S, \rightarrow, s^0)$ *is a function* $\pi : S \rightarrow \mathbb{N}$.
- *A signature function is a function* $sig : (S \rightarrow \mathbb{N}) \times S \rightarrow \mathsf{Sig}$*, such that for all isomorphisms* $\phi : \mathbb{N} \rightarrow \mathbb{N}$ *and all partitions* π*:*

$$\forall s \in S : sig(\phi \circ \pi, s) = \{(a, \phi(n)) \mid (a, n) \in sig(\pi, s)\}$$

The last clause is to ensure that the equality on signatures is independent of how numbers are chosen to represent partitions. This is important because we want to do a refinement process, where based on a partition, we compute signatures, which we turn into a partition, for which we compute signatures, etc. until the partition is stable. This requires translating signatures (or better pairs of previous partition numbers and signatures) to integers, which we do by means of given isomorphisms:

$$h_1, h_2, \cdots : \mathbb{N} \times \mathsf{Sig} \rightarrow \mathbb{N} \ .$$

These isomorphisms exist due to the fact that signatures are finite, which implies that the set of signatures is countable. The actual refinement process works as follows:

- Given an initial partition π_0 of S.
- Given a signature function sig.
- Define $\pi_{i+1}(s) = h_{i+1}(\pi_i(s), sig(\pi_i, s))$
- Define the relation $\pi_i \subseteq S \times S$ as $s \ \pi_i \ t$, if $\pi_i(s) = \pi_i(t)$.
- There exists $N \in \mathbb{N}$ such that the relation $\pi_N = \pi_{N+1}$. Define $\pi_0^{sig} = \pi_N$.

Note that although the definitions of the functions π_{i+1} depend on the choice of the isomorphisms h_{i+1}, the relations π_i will be the same regardless of the choice of h_{i+1}, due to the third clause of Definition 5. This definition is turned into an algorithm by starting with π_i for $i = 0$, and computing π_{i+1} from π_i until the partition is stable ($\pi_{i+1} \equiv \pi_i$).

For the computed refinement to make sense, we need notions of signatures that correspond to meaningful equivalences. For example, the signatures of a state according to strong bisimulation and branching bisimulation are

Definition 6 (classic signatures)

$$\begin{array}{rcl} \mathrm{sig_s}(\pi, s) & = & \{(a, \pi(t)) \mid s \xrightarrow{a} t\} \\ \mathrm{sig_b}(\pi, s) & = & \{(a, \pi(t)) \mid s \xrightarrow{\tau} s_1 \cdots \xrightarrow{\tau} s_n \xrightarrow{a} t, \pi(s) = \pi(s_i) \wedge (a \neq \tau \vee \pi(s) \neq \pi(t))\} \end{array}$$

The signature of a state says which equivalence classes are reachable from the state by performing an action. For example in strong bisimulation, if there is an a step from a state s to a state t then the equivalence class of t is reachable by means of an a step form s which is expressed by putting the pair $(a, \pi(t))$ in the signature of s.

The case for branching bisimulation is more complicated. The set of actions includes the *invisible action* τ. The intent of this label is that whatever happens is unimportant. Thus τ steps are ignored, except if they change the *branching behaviour*. An ignored τ step is called *silent*. More formally a τ step is silent with respect to a partition if it is between states in the same equivalence class.

See [8] and [7] for more explanation.

2.3 Inductive signatures for strong bisimulation

In the classical definition of the strong bisimulation signature, the signatures depend on the previous partition only. One may wonder if in some cases the current partition can be used. The answer is yes. If for each label you consistently use the old partition or consistently use the new partition then it still works. Of course if we use the current partition then we must ensure that all signatures are well defined. This is ensured if the subgraph of edges for which we use the current partition is acyclic. This is guaranteed if we have a *well-founded partition* of the set of actions:

Effectively, we assume a partition $A_?, A_>$ of the set of actions, such that the relation $\{(s,t) \mid s \xrightarrow{a} t \wedge a \in A_>\}$ is well-founded.

Definition 7 *A pair $\langle A_?, A_> \rangle$ is a well founded partition of* Act *for an LTS $(S, \rightarrow, s^0)$ if $A_? \cap A_> = \emptyset$, $A_? \cup A_> = \mathsf{Act}$ and the LTS is $A_>$ cycle free. The order $> \subseteq S \times S$ is defined by $> \equiv \left(\cup_{a \in A_>} \xrightarrow{a}\right)^+$.*

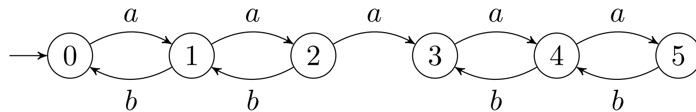
Based on the well-founded order $>$ we can give inductive definitions and proofs. For example, we can define inductive strong bisimulation signatures:

Definition 8 (inductive strong bisimulation) *Given an LTS $(S, \rightarrow, s^0)$, a well founded partition $\langle A_?, A_> \rangle$ for it, an initial partitition function $\pi_0 : S \rightarrow \mathbb{N}$ and isomorphisms $h_1, h_2, \cdots : \mathbb{N} \times \mathsf{Sig} \rightarrow \mathbb{N}$. Define*

$$\begin{array}{rcl} sig_{i+1}(s) & = & \{(a, \pi_i(t)) \mid s \xrightarrow{a} t \wedge a \in A_?\} \cup \{(a, \pi_{i+1}(t)) \mid s \xrightarrow{a} t \wedge a \in A_>\} \\ \pi_{i+1}(s) & = & h_{i+1}(\pi_i(s), sig_{i+1}(s)) \end{array}$$

Note that $sig_{i+1}(s)$ is defined inductively in terms of any π_i-values, and only π_{i+1} values of states that are smaller in $>$. To show how the definition works and how the choice of the partition influences performance, we continue with an example.

Example 9 *Consider the following LTS:*



If we take $A_> := \{a\}$, and set $\pi_0(s) := 0$ for all states, we get the following run:

$$\begin{array}{lcll}
sig_1(5) & := & \{(b,0)\} & \pi_1(5) = 1 \\
sig_1(4) & := & \{(b,0),(a,1)\} & \pi_1(4) = 2 \\
sig_1(3) & := & \{(a,2)\} & \pi_1(3) = 3 \\
sig_1(2) & := & \{(b,0),(a,3)\} & \pi_1(2) = 4 \\
sig_1(1) & := & \{(b,0),(a,4)\} & \pi_1(1) = 5 \\
sig_1(0) & := & \{(a,5)\} & \pi_1(0) = 6
\end{array}$$

Note that every state got a different signature, so in this case we reach the final partition in one round. Also note that the order of computation was completely fixed, because the label a imposes a total order on the states.

Next, consider the same example, but let $A_> = \{b\}$. Note that this is also terminating. Again, we take $\pi_0(s) = 0$ for any state s.

$$\begin{array}{lcll|lcll}
sig_1(0) & := & \{(a,0)\} & \pi_1(0) = 1 & sig_1(3) & := & \{(a,0)\} & \pi_1(3) = 1 \\
sig_1(1) & := & \{(a,0),(b,1)\} & \pi_1(1) = 2 & sig_1(4) & := & \{(a,0),(b,1)\} & \pi_1(4) = 2 \\
sig_1(2) & := & \{(a,0),(b,2)\} & \pi_1(2) = 3 & sig_1(5) & := & \{(b,2)\} & \pi_1(5) = 4 \\
\hline
sig_2(0) & := & \{(a,2)\} & \pi_2(0) = 5 & sig_2(3) & := & \{(a,2)\} & \pi_2(3) = 5 \\
sig_2(1) & := & \{(a,3),(b,5)\} & \pi_2(1) = 6 & sig_2(4) & := & \{(a,4),(b,5)\} & \pi_2(4) = 7 \\
sig_2(2) & := & \{(a,1),(b,6)\} & \pi_2(2) = 8 & sig_2(5) & := & \{(b,7)\} & \pi_2(5) = 9 \\
\hline
sig_3(0) & := & \{(a,6)\} & \pi_3(0) = 10 & sig_3(3) & := & \{(a,7)\} & \pi_3(3) = 11 \\
sig_3(1) & := & \{(a,8),(b,10)\} & \pi_3(1) = 12 & sig_3(4) & := & \{(a,9),(b,11)\} & \pi_3(4) = 13 \\
sig_3(2) & := & \{(a,5),(b,12)\} & \pi_3(2) = 14 & sig_3(5) & := & \{(b,13)\} & \pi_3(5) = 15
\end{array}$$

Note that this time we need three iterations, but there is some room for parallel computation, because the signature of 0 and 3 can be computed independently, because they have no b successors.

2.4 Inductive signatures for branching bisimulation

In the splitting procedure of the Groote-Vaandrager algorithm, whenever a state has one or more τ successors inside the block that is being split, the algorithm tests if the behavior of one of those τ successors includes all of the behavior of the state. If such a successor exists, then the state is put in the same block as that successor. Because of this splitting procedure the graph has to be τ-cycle free. A similar effect can be achieved by exploiting τ cycle freeness when we define the branching signature. Thus, we assume that $\tau \in A_>$ for all partitions $\langle A_?, A_> \rangle$.

The inductive branching signature is computed in two steps. First, the *pre*-signature is computed, which consists of all transitions to all successors, including τ-steps to possibly equivalent states. Second, we look for a τ-successor in the same block of the previous partition which contains all *pre* behavior except the τ step to that successor. If such a successor is found then the signature is the signature of that successor, otherwise the signature is the *pre*-signature:

Definition 10 (inductive branching bisimulation) *Given an LTS $(S, \rightarrow, s^0)$, a well founded partition $\langle A_?, A_> \rangle$ for it with $\tau \in A_>$ and an initial partitition function $\pi_0 : S \rightarrow \mathbb{N}$. Define*

$$\begin{array}{lcl}
pre_{i+1}(s) & = & \{(a, \pi_i(t)) \mid s \xrightarrow{a} t \wedge a \in A_?\} \cup \{(a, \pi_{i+1}(t)) \mid s \xrightarrow{a} t \wedge a \in A_>\} \\
sig_{i+1}(s) & = & \textit{if there exists a } t \textit{ with } s \xrightarrow{\tau} t,\ \pi_i(s) = \pi_i(t) \textit{ and } pre_{i+1}(s) \subseteq sig_{i+1}(t) \cup \{(\tau, \pi_{i+1}(t))\} \\
 & & \quad \textit{then } sig_{i+1}(t) \\
 & & \quad \textit{else } pre_{i+1}(s) \\
\pi_{i+1}(s) & = & h_{i+1}(\pi_i(s), sig_{i+1}(s))
\end{array}$$

It is not immediately obvious that this is well-defined: what if there exists more than one τ-successor that passes the test? The answer is: then they have the same signature. We prove this in lock step with the observation that if a signature σ contains a pair (a, n), then any state with signature σ has a path of silent τ steps to a state where an a step is possible to a final state in partition n.

To avoid unnecessary case distinctions between $a \in A_?$ and $a \in A_>$, we introduce the notation

$$\hat{a} \stackrel{\text{def}}{=} \begin{cases} 0 & \text{, if } a \in A_? \\ 1 & \text{, if } a \in A_> \end{cases}$$

This allows us to abbreviate "$\pi_i(s)$ if $a \in A_?$ and $\pi_{i+1}(s)$ if $a \in A_>$" by $\pi_{i+\hat{a}}(s)$.

Proposition 11 *For all states s:*

1. *If there exist t_1, t_2 with $s \xrightarrow{\tau} t_1$, $s \xrightarrow{\tau} t_2$, $\pi_i(s) = \pi_i(t_1) = \pi_i(t_2)$, $pre_{i+1}(s) \subseteq sig_{i+1}(t_1) \cup \{(\tau, \pi_{i+1}(t_1))\}$ and $pre_{i+1}(s) \subseteq sig_{i+1}(t_2) \cup \{(\tau, \pi_{i+1}(t_2))\}$ then $sig_{i+1}(t_1) = sig_{i+1}(t_2)$.*
2. *If $(a, n) \in sig_{i+1}(s)$ then $\exists s_1, \cdots, s_m, t : s \xrightarrow{\tau} s_1 \cdots \xrightarrow{\tau} s_m \xrightarrow{a} t \wedge \pi_i(s) = \pi_i(s_j) \wedge n = \pi_{i+\hat{a}}(t)$.*

Proof. We prove both parts at once by induction on $\xrightarrow{\tau}^*$.
Given a state s, we prove part 1 by contradiction. Suppose that $sig_{i+1}(t_1) \neq sig_{i+1}(t_2)$. Then

$$\{(\tau, \pi_{i+1}(t_1)), (\tau, \pi_{i+1}(t_2))\} \subseteq pre_{i+1}(s)$$

and therefore:

$$(\tau, \pi_{i+1}(t_1)) \in sig_{i+1}(t_2) \text{ and } (\tau, \pi_{i+1}(t_2)) \in sig_{i+1}(t_1)$$

Let $s_1 = t_1$. Because $s \xrightarrow{\tau} t_1$, the induction hypothesis applies to t_1. Thus by applying part 2, there exists a state s_1', such that $s_1 \xrightarrow{\tau}^+ s_1'$ and $\pi_{i+1}(s_1') = pi_{i+1}(t_2)$. This implies that $sig_{i+1}(s_1') = sig_{i+1}(t_2)$. So we can find s_2, such that $s_1' \xrightarrow{\tau}^+ s_2$ and $\pi_{i+1}(s_2) = \pi_{i+1}(t_1)$. In other words we get an infinite sequence

$$s_1 \xrightarrow{\tau}^+ s_1 \xrightarrow{\tau}^+ s_2 \xrightarrow{\tau}^+ \cdots$$

In a finite state space this implies the existence of a τ cycle. Contradiction.
Part 2 is proven by case distinction. We have two cases:

$sig_{i+1}(s) = pre_{i+1}(s)$ If $(a, n) \in pre_{i+1}(s)$ then for some t: $s \xrightarrow{a} t$ and $n = \pi_{i+\hat{a}}(t)$.

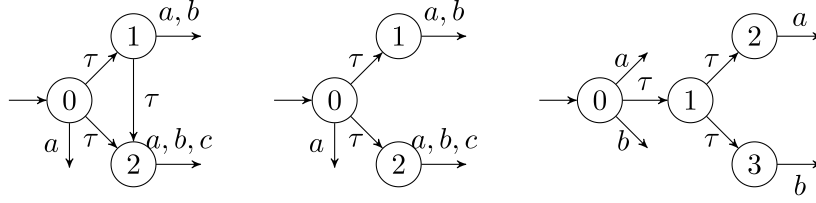
$s \xrightarrow{\tau} t \wedge \pi_i(s) = \pi_i(t) \wedge sig_{i+1}(s) = sig_{i+1}(t)$ By induction hypothesis, we have a sequence $t \xrightarrow{\tau} t_1 \xrightarrow{\tau} \cdots t_m \xrightarrow{a} t'$ satisfying the requirement for t. Which means that the requirement for s is satisfied by

$$s \xrightarrow{\tau} t \xrightarrow{\tau} t_1 \xrightarrow{\tau} \cdots t_m \xrightarrow{a} t'$$

□

We will show how the new definition works and is different from the approach of [7], by means of an example:

Example 12 *Consider the following three examples. We have only drawn the nodes of the graphs which are relevant. Let $\pi_0(s) = 0$ for all s and $\pi_i(s) = 0$ for all nodes s which have been omitted.*



Let $A_{>} = \{\tau\}$. Then for the left-most LTS on the left, we get:

$$\begin{array}{rcll}
pre_1(2) & := & \{(a,0),(b,0),(c,0)\} & \\
sig_1(2) & := & \{(a,0),(b,0),(c,0)\} & \pi_1(2) = 1 \\
pre_1(1) & := & \{(a,0),(b,0),(\tau,1)\} & Note: \{(a,0),(b,0),(\tau,1)\} \subseteq \{(a,0),(b,0),(c,0),(\tau,1)\} \\
sig_1(1) & := & \{(a,0),(b,0),(c,0)\} & \pi_1(1) = 1 \\
pre_1(0) & := & \{(a,0),(\tau,1)\} & Note: \{(a,0),(\tau,1)\} \subseteq \{(a,0),(b,0),(c,0),(\tau,1)\} \\
sig_1(0) & := & \{(a,0),(b,0),(c,0)\} & \pi_1(0) = 1
\end{array}$$

Note that $|dom(sig_1)| = |dom(sig_0)| = 1$, so sig_1 is stable, and all τ-steps are silent.
For the middle LTS, we obtain:

$$\begin{array}{rcll}
pre_1(2) & := & \{(a,0),(b,0),(c,0)\} & \\
sig_1(2) & := & \{(a,0),(b,0),(c,0)\} & \pi_1(2) = 1 \\
pre_1(1) & := & \{(a,0),(b,0)\} & \\
sig_1(1) & := & \{(a,0),(b,0)\} & \pi_1(1) = 2 \\
pre_1(0) & := & \{(a,0),(\tau,1),(\tau,2)\} & Note: \{(a,0),(\tau,1),(\tau,2)\} \not\subseteq \{(a,0),(b,0),(c,0),(\tau,1)\}, \\
 & & & \qquad \{(a,0),(\tau,1),(\tau,2)\} \not\subseteq \{(a,0),(b,0),(\tau,2)\} \\
sig_1(0) & := & \{(a,0),(\tau,1),(\tau,2)\} & \pi_1(0) = 3
\end{array}$$

Note that $|dom(sig_1)| = 3$, which cannot increase, so again sig_1 is stable. In this case, none of the τ-steps is silent.
For the LTS on the right, we get

$$\begin{array}{rcllcrcll}
sig_1(2) & := & \{(a,0)\} & \pi_1(2) = 1 & , & sig_1(3) & := & \{(b,0)\} & \pi_1(3) = 2 \\
sig_1(1) & := & \{(\tau,1),(\tau,2)\} & \pi_1(1) = 3 & , & sig_1(0) & := & \{(a,0),(b,0),(\tau,3)\} & \pi_1(0) = 4
\end{array}$$

Already after one iteration it is detected that none of the τ-steps is silent. In the original definition in [7], this would be detected later, as the following example shows.

$$\begin{array}{llcll}
sigb_1(2) := \{(a,0)\} & \pi_1(2) = 1 & , & sigb_1(3) := \{(b,0)\} & \pi_1(3) = 2 \\
sigb_1(1) := \{(a,0),(b,0)\} & \pi_1(1) = 3 & , & sigb_1(0) := \{(a,0),(b,0)\} & \pi_1(0) = 3 \\
\hline
sigb_2(2) := \{(a,0)\} & \pi_2(2) = 1 & , & sigb_2(3) := \{(b,0)\} & \pi_2(3) = 2 \\
sigb_2(1) := \{(\tau,1),(\tau,2)\} & \pi_2(1) = 4 & , & sigb_2(0) := \{(a,0),(b,0),(\tau,1),(\tau,2)\} & \pi_2(0) = 5
\end{array}$$

Note the two differences between inductive and classic signatures. First, the fact that $0 \xrightarrow{\tau} 1$ is not silent is detected in the first iteration by inductive and the second by classic signatures. Second, in the inductive case the size of the signature is limited by the number of outgoing transitions in the classic case it is not.

2.5 Correctness

We use the same proof technique as in previous work. That is, we prove that bisimilar states are always in the same block and that if a π_i partition is stable (π_i and π_{i+1} denote the same relation) then π_i is a bisimulation. Thus because $\underline{\leftrightarrow}$ is the coarsest bisimulation, we must have that π_i coincides with $\underline{\leftrightarrow}$.

In this section we work on a given LTS $(S, \rightarrow, s^0)$ and well-founded partition $(A_?, A_>)$, with $\tau \in A_>$. We consider inductive branching bisimulation and we let $s \underline{\leftrightarrow}_i t$ denote $\pi_i(s) = \pi_i(t)$.

One of the properties of a τ-cycle free LTS is that given a state one can always follow τ steps to bisimilar states, until a state is found that has no such step. These states are called canonical:

Definition 13 *A state s is* canonical *(denoted $s\downarrow$) if $\neg\exists s' :\ s \xrightarrow{\tau} s' \wedge s \underline{\leftrightarrow} s'$.*

Canonical states have the important property that all visible behavior is present as an immediate step rather than as a sequence of one or more invisible steps followed by a visible step.

Lemma 14 *If $\underline{\leftrightarrow} \subseteq \underline{\leftrightarrow}_i$ then for all states s, t we have $(s \underline{\leftrightarrow} t \wedge t\downarrow) \Rightarrow s \underline{\leftrightarrow}_{i+1} t$*

To prove this, we need two properties.

Proposition 15 *For all states s, t, we have*

1. *$pre_{i+1}(s) \subseteq sig_{i+1}(s) \cup \{(\tau, \pi_{i+1}(s))\}$.*
2. *$pre_{i+1}(s) \subseteq sig_{i+1}(s) \cup \{(\tau, \pi_{i+1}(s))\}$.*

1. *$pre_{i+1}(s) \subseteq sig_{i+1}(s) \cup \{(\tau, \pi_{i+1}(s))\}$.*
2. *if $pre_{i+1}(s) = pre_{i+1}(t)$ then $sig_{i+1}(s) = sig_{i+1}(t)$.*

Proof. Both parts are proven by case distinction.

$\exists s' : s \xrightarrow{\tau} s' \wedge pre_{i+1}(s) \subseteq sig_{i+1}(s') \cup \{(\tau, \pi_{i+1}(s'))$ By definition we have $sig_{i+1}(s) = sig_{i+1}(s')$. Thus, part 1 is trivial.
It also means that $(\tau, \pi_{i+1}(s')) \in pre_{i+1}(s)$. So $(\tau, \pi_{i+1}(s') \in pre_{i+1}(t)$. So for some t', we have $t \xrightarrow{\tau} t'$ and $\pi_{i+1}(t') = pi_{i+1}(s')$. This implies that $sig_{i+1}(t') = sig_{i+1}(s')$. Finally, it follows that $sig_{i+1}(t) = sig_{i+1}(s)$.

otherwise By definition we have $sig_{i+1}(s) = pre_{i+1}(s)$. Thus, part 1 is trivial.
Due to symmetry we have have that $sig_{i+1}(t) = pre_{i+1}(t)$ as well.

□

Proof of Lemma 14. By induction on the order $\geq$ on pairs of states, defined as $(s, t) \geq (s', t')$ iff $s \geq s' \wedge t \geq t'$.

First, we prove that

$$pre_{i+1}(s) \subseteq pre_{i+1}(t) \cup \{(\tau, \pi_{i+1}(t))\} \tag{1}$$

The elements of $pre_{i+1}(s)$ fit one of two cases:

- $(a, \pi_i(s'))$, for $s \xrightarrow{a} s' \wedge a \in A_?$: Because $s \underline{\leftrightarrow} t$ and $a \neq \tau$, we have $t \xrightarrow{\tau}{}^{*} t'' \xrightarrow{a} t'$ with $s \underline{\leftrightarrow} t'' \wedge s' \underline{\leftrightarrow} t'$. Because $t\downarrow$, we have $t'' = t$. By assumption $s' \underline{\leftrightarrow}_i t'$ and thus $(a, \pi_i(s')) \in pre_{i+1}(t)$.

- $(a, \pi_{i+1}(s'))$, for $s \xrightarrow{a} s' \wedge a \in A_>$: We have three sub-cases:
  - $t \xrightarrow{a} t'$ with $s' \underline{\leftrightarrow} t'$: By induction hypothesis $s' \underline{\leftrightarrow}_{i+1} t'$ and thus $(a, h(sig_{i+1}(s')) \in pre_{i+1}(t)$.
  - $t \xrightarrow{\tau}^+ t'' \xrightarrow{a} t'$ with $s \underline{\leftrightarrow} t'' \wedge s' \underline{\leftrightarrow} t'$: Impossible due to $t{\downarrow}$.
  - $a = \tau$ and $s' \underline{\leftrightarrow} t$: By induction hypothesis $s' \underline{\leftrightarrow}_{i+1} t$, so $(a, \pi_{i+1}(s')) = (\tau, \pi_{i+1}(t))$.

This completes the proof of (1). Now, we distinguish on whether s is canonical or not.

- $s{\downarrow}$: In this case we claim $pre_{i+1}(s) = pre_{i+1}(t)$. Each of the inclusion is proven similar to the proof of (1) above. This implies $sig_{i+1}(s) = sig_{i+1}(t)$ and thus $s \underline{\leftrightarrow}_{i+1} t$.

- $s \xrightarrow{\tau} s' \wedge s \underline{\leftrightarrow} s'$: By induction hypothesis $sig_{i+1}(s') = sig_{i+1}(t)$. Thus

  $$pre_{i+1}(s) \subseteq pre_{i+1}(t) \cup \{(\tau, \pi_{i+1}(t))\} \subseteq sig_{i+1}(t) \cup \{(\tau, \pi_{i+1}(t))\} = sig_{i+1}(s') \cup \{(\tau, \pi_{i+1}(s'))\}$$

  Thus $sig_{i+1}(s) = sig_{i+1}(s')$.

□

Lemma 16 *If for all s, t: $s \underline{\leftrightarrow}_i t \Leftrightarrow s \underline{\leftrightarrow}_{i+1} t$ then $\underline{\leftrightarrow}_i$ is a branching bisimulation.*

Proof. Suppose that $s \underline{\leftrightarrow}_i t$ and $s \xrightarrow{a} s'$.
We distinguish two cases:

- $a = \tau \wedge s \underline{\leftrightarrow}_i s'$: This implies $s' \underline{\leftrightarrow}_i t$.

- $a \neq \tau \vee s \not\underline{\leftrightarrow}_i s'$: This implies $(a, \pi_{i+\hat{a}}(s')) \in sig_{i+1}(s)$. So $(a, \pi_{i+\hat{a}}(s')) \in sig_{i+1}(t)$. If $(a, \pi_{i+\hat{a}}(s')) \in pre_{i+1}(t)$ then $t \xrightarrow{a} t'$ with $s' \underline{\leftrightarrow}_i t'$. Otherwise, there must be a t°, such that $t \xrightarrow{a} t^\circ$ and $sig_{i+1}(t) = sig_{i+1}(t^\circ)$. By repeating the case distinction we can construct $t \xrightarrow{\tau}^* t'' \xrightarrow{a} t'$ with $s \underline{\leftrightarrow}_i t'' \wedge s' \underline{\leftrightarrow}_i t'$.

□

3 Distributed Algorithm

In this section, we present a distributed algorithm for computing the branching bisimulation equivalence relation.

The input to the algorithm is an LTS $(S, \rightarrow, s^0)$, a well founded partition $\langle A_?, A_> \rangle$, and a function $owner : S \rightarrow \{1, \cdots, W\}$ where W is the number of workers. The owner function is a given distribution of states among the workers.

The given isomorphisms of the theory are replaced by global hash tables in the implementation. Each worker stores an equal part of this global hash table. A second owner function $owner : ID \times Sig \rightarrow \{1, \cdots, W\}$ stores the worker where the (new) ID of the pair (oldID,signature) is stored.

In the actual implementation states and edges are numbered entities. Since the theory assumes that edges are triples, we need to introduce some new notation. Moreover, we have to distinguish which worker owns which state and which edge, so we need some notation for that as well.

Table 1: Pseudo code for worker w (inductive branching bisimulation reduction)

```
1  set sig[S_w], dest_sig[E_w^τ], old_queue, sig_queue, new_queue
2  int old_id[S_w], current_id[S_w], dst_old[E_w^? ∪ E_w^τ], dst_new[E_w^>]
3  proc reduce()
4     int old_count:=0 , new_count:=1
5     for t ∈ S_w do current_id[t]:=0 end
6     while old_count≠new_count do
7        old_count:=new_count; indexed_set_clear()
8        for t ∈ S_w do old_id[t]:=current_id[t]; current_id[t]:=⊥ end
9        for e in E_w^? do dst_old[e]:=⊥ end ; for e in E_w^> do dst_new[e]:=⊥ end
10       for e in E_w^τ do dst_sig[e]:=⊥ ; dst_old[e]:=⊥ end
11       old_queue := S_w; sig_queue:= {s ∈ S_w | ¬∃a,t : s -a-> t} ; new_queue := ∅
12       do
13       :: take s from old_queue =>
14             for e in pred(s) with lbl(e) ∈ Act_? ∪ {τ} do
15                send set_old(e,old_id[s]) to owner(src(e)) end
16       :: recv set_old(e,id) => dst_old[e]:=id; check_ready(src(e))
17       :: take s from sig_queue =>
18             sig := compute_sig(s);
19             for e in pred(s) with lbl(e) = τ do
20                send set_sig(e,sig) to owner(src(e)) end
21             send get_global(s,old_id[s],sig) to owner(old_id[s],sig)
22       :: recv set_sig(e,e_sig) => dest_sig[e] := e_sig; check_ready(src(e))
23       :: recv get_global(s,id_old,sig) =>
24             send set_global(s,indexed_set_put(id_old,sig)) to owner(s)
25       :: recv set_global(s,id) => current_id[s]:=id; add s to new_queue
26       :: take s from new_queue =>
27             for e in pred(s) with lbl(e) ∈ Act_> do
28                send set_new(e,current_id[s]) to owner(src(e)) end
29       :: recv(set_new(e,id)) => dst_new[e]:=id; check_ready(src(e))
30       until ∀s ∈ S : current_id[s] ≠ ⊥
31       new_count:=distributed_sum(index_count)
32    end
33 end
```

The functions src, dst and lbl provide access to the source state, destination state and label of an edge, respectively:

$$\forall e \equiv (s,a,t) \in \rightarrow : src(e) = s,\ lbl(e) = a \text{ and } dst(e) = t\ .$$

Each worker owns a set of states and needs to know the outgoing τ edges, $A_?$ edges and $A_>$ edges:

$$S_w = \{s \in S \mid owner(s) = w\} \qquad E_w^\tau = \{e \in \rightarrow \mid src(e) \in S_w \wedge lbl(e) = \tau\}$$
$$E_w^? = \{e \in \rightarrow \mid src(e) \in S_w \wedge lbl(e) \in A_?\} \qquad E_w^> = \{e \in \rightarrow \mid src(e) \in S_w \wedge lbl(e) \in A_>\}$$

Finally, we need the definitions of successor and predecessor edges of a state:

$$succ(s) = \{e \mid src(e) = s\} \qquad pred(s) = \{e \mid dst(e) = s\}$$

Each worker stores both ingoing and outgoing edges of the states it owns in a way that allows it to quickly enumerate the successors and predecessors of every state.

Next, we will explain our algorithm for distributed computation of inductive signatures. Pseudo

code of the main loop can be found in Table 1. It leaves out the details of the signature computation and global hash table. These details can be found in table 2. The algorithm works in a few steps:

1. Put the initial partitition (every state is equivalent) in the current partition and start the first iteration. (See table 1, lines 4-5.)
2. Initialize the data structure needed in each iteration. That is, set the values of the successor partition IDs and signatures to undefined, clear the global hash table, clear the signature and new ID queues and put all states in the old ID queue. (See table 1, lines 7-11.)
3. If a state is in the old ID queue it means that the ID with respect to the previous partition has to be forwarded to the predecessors. This is done by sending a message for every incoming $A_?$ or τ edge. (See table 1, lines 13-15.) If such a message is received then the old ID is stored and if necessary the state is put in the signature queue. (See table 1, line 16.).
4. If a state is in the signature queue then all information needed to compute the signature is present. Once the signature has been computed it is sent to all τ predecessors and a request is sent to the global hash table to resolve the ID of the (oldID, signature) pair. (See table 1, lines 17-21.) If a signature set request is received then the signature is set and if necessary the state is put in the signature queue. (See table 1, line 22.) If a hash table request is received then the lookup is made and the reply is sent immediately. (See table 1, lines 23-24.) Upon receiving the reply, the state is put in the new ID queue. (See table 1, line 25.)
5. If a state is in the new ID queue then the ID in the current partition is ready to be sent to all $A_>$ predecessors. (See table 1, lines 26-28.) Receiving such a message leads to storing the result and possibly inserting the state in the signature queue. (See table 1, line 29.)
6. As soon as the new partition ID of every state is known everywhere, the message loop can exit. Note that this requires a simple form of distributed termination detection.
7. By adding up the share of every partition ID hash table, we compute the number of partitions and we repeat the loop if necessary.

As described above, messages from the old queue, signature queue and new queue are dealt with in parallel until finished. The actual implementation deals with these messages in waves: first the entire old queue is dealt with then the signature queue and new queue are emptied globally in sub iterations.

Before we discuss the experiments with our prototype implementation, we first discuss the time, memory and message complexity. For this analysis we assume that the fan out of every state is bounded. We assume an LTS with N states and M transitions.

The time needed for the algorithm is the number of iterations times the cost of each iteration. The worst case number of iterations is the number of states N. (E.g. for the LTS $(\{0, \cdots, N-1\}, i \xrightarrow{a} i+1 \bmod N \cup 0 \xrightarrow{b} 0, 0)$.) In each iteration, for each state we must compute the signature and insert it in the global hash table. Due to the fact that the fan out is constant, this requires $\mathcal{O}(N)$ time and messages. For each edge, we may have to send the old ID, the new ID and the signature. This requires $\mathcal{O}(M)$ time and messages. Overall, the worst case time complexity is $\mathcal{O}(N \cdot N + M)$.

The number of times one cannot avoid waiting for a message in each iteration depends on the length of the longest $A_>$ path in the graph: computation has to start at the last node and work up to the first, incurring three message latencies at each step.

Table 2: Subroutines for inductive branching minimization.

```
1  proc check_ready(s)
2     for e in succ(s) do
3        if dest_id[e]=⊥ or lbl(e) = τ ∧ dest_sig[e]=⊥ then return end
4     end
5     add s to sig_queue
6  end
7  set compute_sig(s)
8      pre := ∅
9      for e in succ(s) ∩ E_w^? do pre := pre ∪ {(lbl(e), dst_old[e])} end
10     for e in succ(s) ∩ E_w^> do pre := pre ∪ {(lbl(e), dst_new[e])} end
11     for e in succ(s) with lbl(e) = τ and dest_id[s] = dst_old[e] do
12         if pre ⊆ dest_sig[e]∪{(τ, dst_new[e])} then return dest_sig[e] end
13     end
14     return pre
15 end
16 int  index_count:=0;  hashtable index_table:=∅
17 proc indexed_set_clear() index_count:=0; index_table:=∅ end
18 int indexed_set_put(pair)
19    if index_table[pair] = ⊥ then
20       index_table[pair]:=index_count∗workers+me; index_count++ end
21    return index_table[pair]
22 end
```

The memory needed by the algorithm to store the LTS and the signatures is linear in the number of states and transitions: $\mathcal{O}(N + M)$. (This is a difference to the old algorithm where even if the fan out was bounded, the size of many signatures could be in the order of the number of edges.) Provided that the owner functions work well, the memory use is evenly distributed across all workers. The memory needed for message buffering can be kept constant, because each step that involves sending more than one message is a step where a state has to be taken from a queue. Blocking these steps if the number of messages in the system is above a threshold limits the number of messages to that threshold. Overall, the worst case memory complexity of the algorithm is $\mathcal{O}(N + M)$.

The worst case memory is also the expected memory complexity, since we expect to keep the LTS in memory. The expected time complexity is much lower than the worst case: The expected number of iterations and the expected length of the longest $A_>$ path are orders of magnitude less than the number of states.

4 Experimental Evaluation

To study the performance of the implementation of the new algorithm, we use four models. We perform two tests on these models. First, we compare with existing branching bisimulation reduction tools. Second, we test how well the new implementation scales in the number of computes nodes and cores used per node. In addition, we briefly mention work in progress on inductive strong bisimulation.

Table 3: Problem sizes

	original		cycle free		branching		iterations				
	states	trans.	states	trans.	states	trans.	c.b.	i.b.	c.s.	i.s.	p
lift6	33,949,609	165,318,222	33,946,699	165,312,102	12,463	71,466	16	8	91	7	78
swp6	56,793,060	271,366,320	13,606,212	56,996,856	8,191	16,380	13	13	20	13	51
1394fin	88,221,818	152,948,696	86,692,394	148,537,294	26,264	79,002	7	5	91	6	75
fr53	84,381,157	401,681,445	81,115,587	385,379,715	2	1	2	2	-	-	196

The models that we use in our experiments are:

lift6 A distributed lift system [13]. This model describes a system that can lift large vehicles by using one leg for each wheel of the vehicle. These legs are connected in a ring topology. The instance we used has 6 legs.

swp6 A version of the sliding window protocol [1]. It has 2 data elements, the channels can contain at most one element and the window size is 6.

fr53 A model of Franklin's leader election protocol for anonymous processes along a bidirectional ring of asynchronous channels, which terminates with probability one [2, 11]. We chose an instance with 5 nodes and 3 identities.

1394fin Model of the physical layer service of the 1394 or firewire protocol and also the link layer protocol entities [16, 19]. We use an instance with 3 links and 1 data element.

The sizes of these models, in their original, cycle eliminated and branching reduced forms are shown in Table 3. This table also show the number of iterations needed by classic branching (c.b.), inductive branching (i.b.), classic strong (s.c.), inductive strong (i.s.) and the length of the longest τ path (p). Note that in two cases (lift6 and 1394fin) the number of iterations needed by the inductive branching algorithm is less than the number needed by the classical algorithm. Also note that the number of iterations needed for inductive strong bisimulation is always a lot less. It will be interesting to see, if we get similar results if we use real input graphs and $A_>$, instead of τ-cycle reduced graphs and $A_> = \{\tau\}$.

In Table 4, we show the results of the comparison. The tools in the comparison are

bcg_min The reduction tool from the CADP toolset [12]. Version 1.7 from the 2007q beta release, 64 bit installation. This implements the algorithm from [14], for which first the τ-cycles must be eliminated (ce).

ltsmin sequential The reduction tool which is released as part of the μCRL toolset [6]. We additionally implemented a sequential version of the inductive branching bisimulation algorithm in this tool.

ltsmin distributed A distributed implementation, which contains the classic distributed branching bisimulation reduction algorithm from [7], and the newly implemented inductive branching bisimulation reduction algorithm.

For `bcg_min`, we show the total time needed for reading the input, reducing and writing the output. For ltsmin sequential, we show both the total time and the time needed for reduction. For ltsmin distributed classic, we show the reduction time (wall clock time). For ltsmin distributed inductive, we show the time for sequential cycle elimination and the wall clock time of distributed

Table 4: Sequential tool comparison.

	bcg_min		ltsmin (sequential implementation)									ltsmin (distributed, 4 cores)			
	ce + GV [14]		classic			ce + classic			ce + inductive			classic		ce + inductive	
	time	mem	time	red	mem	time	red	mem	time	red	mem	red	mem	red	mem
lift6	1251	6493	261	225	2939	298	261	2203	191	154	2299	655	7116	64+246	5520
swp6	1298	10699	342	287	5464	264	209	3625	166	111	3573	621	12129	73+133	3587
1394	20906	8226	248	218	3473	231	201	2482	144	114	2724	730	8657	62+272	6315
fr53	204	15870	305	237	9744	1247	1180	5377	715	651	5462	188	16871	624+476	12991

reduction. In all cases we additionally show the total memory requirements in MB. The tests were performed on a dual quad core Xeon 3GHz machine with 48GB memory.

Several conclusions can be drawn from the results. By looking at the results for sequential ltsmin, we can conclude that inductive signatures are better than classic signatures. By looking at the times needed for fr53 it is obvious that this implementation of cycle elimination in ltsmin should be improved.

We can also conclude that on these cases, sequential ltsmin uses much less memory than bcg_min for branching bisimulation. With the exception of fr53, sequential ltsmin is also much faster than bcg_min. Part of the reason is that ltsmin is 64 bit optimized and bcg_min is not. (The performance of sequential ltsmin is identical when compiled 32 or 64 bit. The 64 bit version of bcg_min uses twice as much memory and 33% more time than the 32 bit version.)

It is also clear that the distributed tool is much more expensive in time and memory than the sequential tool. The extra cost in memory is easily explained. In ltsmin, signature ID's are stored per state only. In ltsmin they have to be stored per state and per transition. In ltsmin the LTS itself takes 4 bytes per state and 8 bytes per transition (label and state). In ltsmin it takes 8 bytes per state and 24 bytes per transition (label, owner and state for ingoing and outgoing edges). This mean that ltsmin has to work through roughly 3 times as much data in each iteration, which might take up to 3 times as much time. Frequent synchronization between the workers and having to send and receive information that in ltsmin can simply be accessed is expected to account for a lot of time.

To test how well the algorithms scale, we first eliminated the τ cycles from the four examples and then ran the inductive reduction on 1, 2, 4 and 8 nodes with 1, 2 ,4 and 8 cores per node. For these tests, we used a cluster with dual quad core Xeon 2GHz, 8GB memory machines connected with gigabit ethernet. The times needed for the reduction can be seen in Fig. 1.

The graphs have been ordered from the smallest to the largest problem. It is interesting to see that for the smallest problem (swp6), the first time that more workers leads to more rather than less time is using 2 nodes, 2 cores per node. For the next two (lift6,1394fin) this happens at 2 nodes, 4 cores per node and for the largest (franklin) at 4 nodes, 4 cores per node.

It is also clear that using 8 cores instead of 4 is problematic. For 1 and 2 nodes the performance increase is small and for 4 and 8 nodes, the performance actually gets worse. Taken together with the huge difference in performance between the sequential and the distributed tool this leads to the (unsurprising) conclusion that it would be better to change the implementation to be aware of which workers are local (allow shared memory) and which workers are remote (require message passing). We leave such a tuned heterogeneous cluster-of-multi-cores implementation for future work.

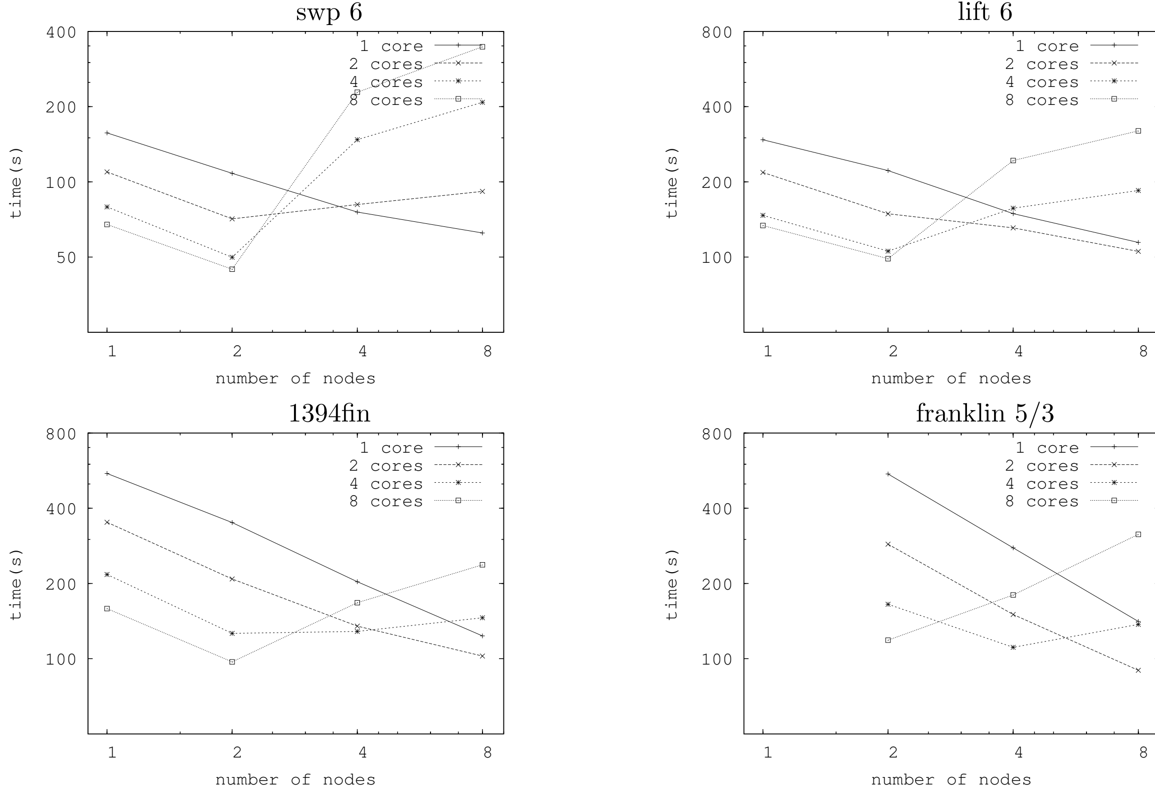


Figure 1: Distributed reduction times for inductive branching bisimulation

5 Conclusion

We have defined the notion of inductive branching signature and proven that it corresponds to branching bisimulation. We have given a distributed algorithm that computes the coarsest branching bisimulation using inductive signatures. In the experiments section, we have shown that it is possible to implement the algorithm in such a way that it scales for up to 8 workers with 1 or 2 cores.

The current prototype is good enough to show the merit of the concept of inductive signatures. However, it can be optimized in several ways. For example, the information about edges between two workers is currently stored by both the source worker and the destination worker. If both workers are on the same machine, then they could share a single instance of the data. Similarly, the algorithm uses a lot of small messages. For good performance, message combining is needed, which is currently done at the worker level, but could be done at the node level instead.

Because strong bisimulation is a special case of branching bisimulation, our algorithm can also be used for strong bisimulation. However, for branching bisimulation we can eliminate τ cycles to get a well-founded partition. For strong bisimulation, we will have to come up with a good heuristic to automatically find well-founded partitions.

As a final conclusion, we note that inductive signatures for branching bisimulation improve time and memory requirements compared to classical signatures, both in a sequential and a distributed implementation. Of course, distributed minimization can handle larger graphs that don't fit in

the memory of a single machine. Additionally, the distributed version using 8 cores on 2 nodes consistently beats the best sequential algorithm in time.

References

[1] Bahareh Badban, Wan Fokkink, Jan Friso Groote, Jun Pang, and Jaco van de Pol. Verification of a sliding window protocol in μCRL and PVS. *Formal Aspects of Computing*, 17(3):342–388, 2005.

[2] Rena Bakhshi, Wan Fokkink, Jun Pang, and Jaco van de Pol. Leader election in anonymous rings: Franklin goes probabilistic. In Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri, and C.-H. Luke Ong, editors, *IFIP TCS*, volume 273 of *IFIP*, pages 57–72. Springer, 2008.

[3] J. Barnat, J. Chaloupka, and J. Van De Pol. Distributed Algorithms for SCC Decomposition. *Journal of Logic and Computation*, 2009.

[4] Jiri Barnat, Lubos Brim, Ivana Cerná, Pavel Moravec 0002, Petr Rockai, and Pavel Simecek. Divine - a tool for distributed verification. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 278–281. Springer, 2006.

[5] Twan Basten. Branching bisimilarity is an equivalence indeed! *Inf. Process. Lett.*, 58(3):141–147, 1996.

[6] Stefan Blom, Wan Fokkink, Jan Friso Groote, Izak van Langevelde, Bert Lisser, and Jaco van de Pol. μcrl: A toolset for analysing algebraic specifications. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer, 2001.

[7] Stefan Blom and Simona Orzan. Distributed branching bisimulation reduction of state spaces. *Electr. Notes Theor. Comput. Sci.*, 89(1), 2003.

[8] Stefan Blom and Simona Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *STTT*, 7(1):74–86, 2005.

[9] Stefan Blom and Simona Orzan. Distributed state space minimization. *STTT*, 7(3):280–291, 2005.

[10] Stefan Blom and Jaco van de Pol. Distributed branching bisimulation minimization by inductive signatures, 2009. Accepted for PDMC 2009.

[11] Wm. Randolph Franklin. On an Improved Algorithm for Decentralized Extrema Finding in Circular Configurations of Processors. *Commun. ACM*, 25(5):336–337, 1982.

[12] Hubert Garavel, Radu Mateescu, Frédéric Lang, and Wendelin Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer, 2007.

[13] Jan F. Groote, Jun Pang, and Arno G. Wouters. A Balancing Act: Analyzing a Distributed Lift System. In S. Gnesi and U. Ultes-Nitsche, editors, *Proc. 6th Workshop on Formal Methods for Industrial Critical Systems*, pages 1–12, 2001.

[14] Jan Friso Groote and Frits W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In Mike Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer, 1990.

[15] William McLendon III, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901 – 910, 2005.

[16] S.P. Luttik. Description and formal specification of the link layer of P1394. Technical Report SEN-R9706, CWI, Amsterdam, The Netherlands, 1997.

[17] Simona Orzan. *On distributed verification and verified distribution.* PhD thesis, VU Amsterdam, The Netherlands, 2004.

[18] Simona Orzan and Jaco van de Pol. Detecting strongly connected components in large distributed state spaces. Technical Report SEN-E0501, CWI, Amsterdam, 2005.

[19] Mihaela Sighireanu and Radu Mateescu. Verification of the Link Layer Protocol of the IEEE-1394 Serial Bus (FireWire): An Experiment with E-LOTOS. *STTT*, 2(1):68–88, 1998.

[20] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

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