State Space Reduction Using Partial τ-Confluence

Jan Friso Groote[1,2] and Jaco van de Pol[1]

[1] CWI, P.O.-box 94.079, 1090 GB Amsterdam, The Netherlands
[2] Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract. We present an efficient algorithm to determine the maximal class of confluent τ-transitions in a labelled transition system. Confluent τ-transitions are inert with respect to branching bisimulation. This allows to use τ-priorisation, which means that in a state with a confluent outgoing τ-transition all other transitions can be removed, maintaining branching bisimulation. In combination with the removal of τ-loops, and the compression of τ-sequences this yields an efficient algorithm to reduce the size of large state spaces.

1 Introduction

A currently common approach towards the automated analysis of distributed systems is the following. Specify an instance of the system with a limited number of parties and a small data domain. Subsequently, generate the state space of this system and reduce it using an appropriate equivalence, for which weak bisimulation [16] or branching bisimulation [6] generally serves quite well. The reduced state space can readily be manipulated, and virtually all questions about it can be answered with ease, using appropriate, available tools (see e.g. [5,4,8] for tools to generate and manipulate state spaces). By taking the number of involved parties and the data domains as large as possible, a good impression of the behaviour can be obtained and many of its problems are exposed, although total correctness cannot be verified in general.

A problem of the sketched route is that the state spaces that are generated are as large as possible, which, giving the growing memory capacities of contemporary computers is huge. So, as the complexity of reduction algorithms is generally more than linear, the time required to reduce these state spaces increases even more. Let n be the number of states and m be the number of transitions of a state space. The time complexity of computing the minimal branching bisimilar state space is $\mathcal{O}(nm)$ [11]; for weak bisimulation this is $\mathcal{O}(n^\alpha)$ where $\alpha \approx 2.376$ is the constant required for matrix multiplication [12].

We introduce a state space reduction algorithm of complexity $\mathcal{O}(m\, Fanout_\tau^3)$ where $Fanout_\tau$ is the maximal number of outgoing τ-transitions of a node in

[0] A technical report including full proofs appeared as [9].
E-mail: `JanFriso.Groote@cwi.nl`, `Jaco.van.de.Pol@cwi.nl`

M. Nielsen and B. Rovan (Eds.): MFCS 2000, LNCS 1893, pp. 383–393, 2000.
© Springer-Verlag Berlin Heidelberg 2000

the transition system. Assuming that for certain classes of transition systems $Fanout_\tau$ is constant, our procedure is linear in the size of the transition system.

The reduction procedure is based on the detection of *τ-confluence*. Roughly, we call a τ-transition from a state s confluent if it commutes with any other a-transition starting in s. When the maximal class of confluent τ-transitions has been determined, τ-priorisation is applied. This means that any outgoing confluent τ-transition may be given "priority". In some cases this reduces the size of the state space with an exponential factor. For convergent systems, this reduction preserves branching bisimulation, so it can serve as a preprocessing step to computing the branching bisimulation minimization.

Related Work. Confluence has always been recognized as an important feature of the behaviour of distributed communicating systems. In [16] a chapter is devoted to various notions of determinacy of processes, among which confluence, showing that certain operators preserve confluence, and showing how confluence can be used to verify certain processes. In [14,19] these notions have been extended to the π-calculus. In [10] an extensive investigation into various notions of global confluence for processes is given, where it is shown that by applying τ-priorisation, state spaces could be reduced substantially. In particular the use of confluence for symbolic verification purposes in the context of linear process operators was discussed. In [17] it is shown how using a typing system on processes it can be determined which actions are confluent, without generating the transition system. In [13] such typing schemes are extended to the π-calculus. Our method is also strongly related to partial order reductions [7,21], where an independence relation on actions and a property to be checked are assumed. The property is used to hide actions, and the independence relation is used for a partial order reduction similar to our τ-priorisation.

Our primary contribution consists of providing an algorithm that determines the maximal set of confluent τ-transitions for a given transition system. This differs from the work in [10] which is only applicable if all τ-transitions are confluent, which is often not the case. It also differs from approaches that use type systems or independence relations, in order to determine a subset of the confluent τ-transitions. These methods are incapable of determining the maximal set of confluent τ-transitions in general.

In order to assess the effectiveness of our state space reduction strategy, we implemented it and compared it to the best implementation of the branching bisimulation reduction algorithm that we know [3]. In combination with τ-loop elimination, and τ-compression, we found that in the worst case the time that our algorithm required was in the same order as the time for the branching bisimulation algorithm. Under the favourable conditions that there are many equivalence classes and many visible transitions, our algorithm appears to be a substantial improvement over the branching bisimulation reduction algorithm.

Acknowledgements. We thank Holger Hermanns for making available for comparison purposes a new implementation of the branching bisimulation algorithm devised by him and others [3].

2 Preliminaries

In this section we define elementary notions such as labelled transition systems, confluence, branching bisimulation and T-convergence.

Definition 1. *A labelled transition system is a triple $A = (S, Act, \longrightarrow)$ where S is a set of states; Act is a set of actions; and $\longrightarrow \subseteq S \times Act \times S$ is the transition relation. We assume that $\tau \in Act$, a special constant denoting internal action.*

We write $\xrightarrow{a}$ for the binary relation $\{\langle s,t\rangle \mid \langle s,a,t\rangle \in\longrightarrow\}$. We write $s \xrightarrow{\tau^*} t$ iff there is a sequence $s_0, \ldots, s_n \in S$ with $n \geq 0$, $s_0 = s$, $s_n = t$ and $s_i \xrightarrow{\tau} s_{i+1}$. We write $t \rightleftarrows s$ iff $t \xrightarrow{\tau^*} s$ and $s \xrightarrow{\tau^*} t$, i.e. s and t lie on a τ-loop. Finally, we write $s \xrightarrow{\overline{a}} s'$ if either $s \xrightarrow{a} s'$, or $s = s'$ and $a = \tau$.

A set $T \subseteq \xrightarrow{\tau}$ is called a *silent transition set of* A. We write $s \xrightarrow{\tau}_T t$ iff $\langle s,t\rangle \in T$. With $s \xrightarrow{\overline{\tau}}_T t$ we denote $s = t$ or $s \xrightarrow{\tau}_T t$. We define the set $Fanout_\tau(s)$ for a state s by: $Fanout_\tau(s) = \{s \xrightarrow{\tau} s' | s' \in S\}$. A is *finite* if S and Act have a finite number of elements. In this case, n denotes the number of states of A, m is the number of transitions in A and m_τ denotes the number of τ-transitions. Furthermore, we write $Fanout_\tau$ for the maximal size of the set $Fanout_\tau(s)$.

Definition 2. *Let $A = (S, Act, \longrightarrow)$ be a labelled transition system and T be a silent transition set of A. We call A T-confluent iff for each transition $s \xrightarrow{\tau}_T s'$ and for all $s \xrightarrow{a} s''$ ($a \in Act$) there exists a state $s''' \in S$ such that $s' \xrightarrow{\overline{a}} s'''$, $s'' \xrightarrow{\overline{\tau}}_T s'''$. We call A confluent iff A is $\xrightarrow{\tau}$-confluent.*

Definition 3. *Let $A = (S_A, Act, \longrightarrow_A)$ and $B = (S_B, Act, \longrightarrow_B)$ be labelled transition systems. A relation $R \subseteq S_A \times S_B$ is called a branching bisimulation relation on A and B iff for every $s \in S_A$ and $t \in S_B$ such that sRt it holds that*

1. *If $s \xrightarrow{a}_A s'$ then for some t' and t'', $t \xrightarrow{\tau^*}_B t' \xrightarrow{\overline{a}}_B t''$ and sRt' and $s'Rt''$.*
2. *If $t \xrightarrow{a}_B t'$, then for some s' and s'', $s \xrightarrow{\tau^*}_A s' \xrightarrow{\overline{a}}_A s''$ and $s'Rt$ and $s''Rt'$.*

For states $s \in S_A$ and $t \in S_B$ we write $s \leftrightarrow_b t$ on $A \times B$, and say s and t are *branching bisimilar*, iff there is a *branching bisimulation relation* R on A and B such that sRt. In this case, $\leftrightarrow_b$ itself is the maximal branching bisimulation, and it is an equivalence relation. A transition $s \xrightarrow{\tau} s'$ is called *inert* iff $s \leftrightarrow_b s'$.

Theorem 1. *Let $A = (S, Act, \longrightarrow)$ be a labelled transition system and let T be a silent transition set of A. If A is T-confluent, every $s \xrightarrow{\tau}_T s'$ is inert.*

Proof. (sketch) It can be shown that the relation $R = \xrightarrow{\overline{\tau}}_T$ is a branching bisimulation relation. □

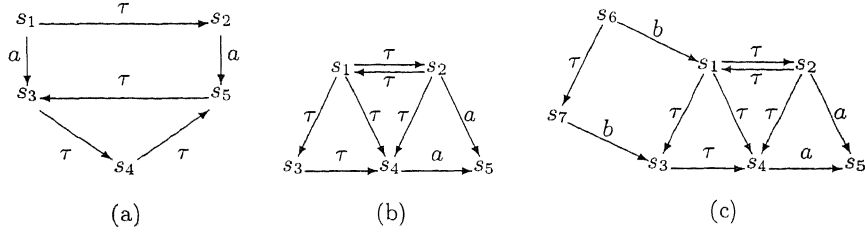


Fig. 1. Counterexamples to preservation of confluence

Lemma 1. *Let A be a labelled transition system. There exists a largest silent transition set T_{conf} of A, such that A is T_{conf}-confluent.*

Proof. (sketch) Consider the set $\mathcal{T}$, being the set of all silent transition sets T such that A is T-confluent. Define $T_{conf} = \bigcup \mathcal{T}$. □

Definition 4. *Let $A = (S, Act, \longrightarrow)$ be a labelled transition system. We call A convergent iff there is no infinite sequence $s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \cdots$.*

3 Elimination of τ-Cycles

In this section we define the removal of τ-loops from a transition system. The idea is to collapse each loop to a single state. This can be done, because $\rightleftarrows$ is an equivalence relation on states.

Definition 5. *Let $A = (S, Act, \longrightarrow)$ be a labelled transition system. Define $[s]_A = \{t \in S \mid t \rightleftarrows s\}$. Define the relation $[\longrightarrow]_A$, such that $S[\xrightarrow{a}]_A S'$ iff there exist $s \in S$, $s' \in S'$ such that $s \xrightarrow{a} s'$ but not $S = S'$ and $a = \tau$. Write $[S]_A$ for $\{[s]_A | s \in S\}$ and $[T]_A$ for the relation $\{[s]_A[\xrightarrow{a}]_A[t]_A \mid s \xrightarrow{a} t \in T\}$.*

Definition 6. *Let $A = (S, Act, \longrightarrow)$ be a labelled transition system. The τ-cycle reduction of A is the labelled transition system $A_\otimes = ([S]_A, Act, [\longrightarrow]_A)$.*

Using the algorithm to detect strongly connected components [1] it is possible to construct the τ-cycle reduction of a labelled transition system A in linear time.

Lemma 2. *Let $A = (S, Act, \longrightarrow)$ be a labelled transition system and let $A_\otimes$ be its τ-cycle reduction. Then for every state $s \in S$, $s \leftrightarrow_b [s]_A$ on $A \times A_\otimes$.*

We now show that taking the τ-cycle reduction can change the confluence structure of a process. Figure 1.a shows that a non-confluent τ-transition may become confluent after τ-cycle reduction: $s_1 \xrightarrow{\tau} s_2$ is not confluent before τ-cycle reduction, but it is afterwards. Conversely, Figure 1.b shows that a confluent τ-transition may become non-confluent after τ-cycle reduction. Observe that all τ-transitions are confluent. After τ-cycle reduction is applied, s_1 and s_2 are taken together, and we see that $\{s_1, s_2\} \xrightarrow{\tau}_\otimes \{s_3\}$ and $\{s_1, s_2\} \xrightarrow{a}_\otimes \{s_5\}$; but

there is no way to complete the diagram. We can extend the example slightly (Figure 1.c) showing that states that have outgoing confluent transitions before τ-cycle reduction, do not have these afterwards. Nevertheless, τ-cycle reduction is unavoidable in view of Example 1 (Section 5).

4 Algorithm to Calculate T_{conf}

We now present an algorithm to calculate T_{conf} for a given labelled transition system $A = (S, Act, \longrightarrow)$. First the required data structures and their initialization are described: Each transition $s \xrightarrow{\tau} s'$ is equipped with a boolean `candidate` that is initially set to true, indicating whether this transition is still a candidate to be put in T_{conf}. Furthermore, for every state s we store a list of all incoming transitions, as well as a list with outgoing τ-transitions. Also, each transition $s \xrightarrow{a} s'$ is stored in a hash table, such that given s, a and s', it can be found in constant time, if it exists. Finally, there is a stack on which transitions are stored, in such a way that membership can be tested in constant time. Transitions on the stack must still be checked for confluence; initially, all transitions are put on the stack.

The algorithm now works as follows. As long as the stack is not empty, remove a transition $s \xrightarrow{a} s'$ from it. Check that $s \xrightarrow{a} s'$ is still confluent with respect to all τ-transitions outgoing from state s that have the variable `candidate` set. Checking confluence means that for each candidate transition $s \xrightarrow{\tau} s''$ it must be verified that either

- for some $s''' \in S$, $s'' \xrightarrow{a} s'''$ and $s' \xrightarrow{\tau} s'''$, which is still `candidate`;
- or $s'' \xrightarrow{a} s'$;
- or $a = \tau$ and $s' \xrightarrow{\tau} s''$ with the variable `candidate` set;
- or finally, $a = \tau$ and $s' = s''$.

For all transitions $s \xrightarrow{\tau} s''$ for which the confluence check with respect to some $s \xrightarrow{a} s'$ fails, the boolean `candidate` is set to false. If there is at least one transition $s \xrightarrow{\tau} s''$ for which the check fails, then all transitions $t \xrightarrow{a} s$ that are not on the stack must be put back on it. This can be done conveniently, using the list of incoming transitions of node s.

After the algorithm has terminated, i.e. when the stack is empty, the set T_{alg} is formed by all τ-transitions for which the variable `candidate` is still true. Termination of the algorithm follows directly from the following observation: either, the size of the stack decreases, while the number of candidate transitions remains constant; or the number of candidate transitions decreases, although in this case the stack may grow. Correctness of the algorithm follows from the theorem below, showing that $T_{alg} = T_{conf}$.

Lemma 3. *A is T_{alg}-confluent.*

Proof. Consider transitions $s \xrightarrow{a} s'$ and $s \xrightarrow{\tau}_T s''$. Consider the last step, with index say n, in the algorithm where $s \xrightarrow{a} s'$ is removed from the stack. The variable `candidate` of $s \xrightarrow{\tau} s''$ was never set to false, hence either:

- $a = \tau$ and $s' = s''$, or
- $s'' \xrightarrow{a} s'$, or
- $a = \tau$ and $s' \xrightarrow{\tau} s''$ with the variable `candidate` set (at step n), or
- for some $s''' \in S$, $s' \xrightarrow{\tau} s'''$ was a candidate at step n, and $s'' \xrightarrow{a} s'''$.

In the first two cases it is obvious that $s \xrightarrow{a} s'$ and $s \xrightarrow{\tau} s''$ are T_{alg}-confluent w.r.t. each other. In the last two cases confluence is straightforward, if respectively $s' \xrightarrow{\tau} s''$ or $s' \xrightarrow{\tau} s'''$ are still candidate transitions when the algorithm terminates. This means that these transitions are put in T_{alg}. If, however, this is not the case, then there is a step $n' > n$ in the algorithm where the `candidate` variable of $s' \xrightarrow{\tau} s''$ or $s' \xrightarrow{\tau} s'''$, respectively, has been reset. In this case each transition ending in s' is put back on the stack. In particular $s \xrightarrow{a} s'$ is put on the stack to be removed at some step $n'' > n' > n$, contradicting the fact that n was the last such step. □

Theorem 2. $T_{conf} = T_{alg}$

Proof. From the previous lemma, it follows that $T_{alg} \subseteq T_{conf}$. We now prove the reverse. Assume towards a contradiction that $s \xrightarrow{\tau} s'$ in T_{conf} is the first transition in the algorithm, whose variable `candidate` is erroneously marked false. This only happens when confluence w.r.t. some $s \xrightarrow{a} s''$ fails. By T_{conf}-confluence, for some s''', $s'' \xrightarrow{\overline{\tau}}_{T_{conf}} s'''$ and $s' \xrightarrow{\overline{a}} s'''$. As $s \xrightarrow{\tau} s'$ is marked false, it must be the case that $s'' \xrightarrow{\tau} s'''$, and its `candidate` bit has been reset earlier in the algorithm. But this contradicts the fact that we are considering the first instance where a boolean `candidate` was erroneously set to false. □

Lemma 4. *The algorithm terminates in $\mathcal{O}(m\, Fanout_\tau^3)$ steps.*

Proof. Checking that a transition $s \xrightarrow{a} s'$ is confluent, requires $\mathcal{O}(Fanout_\tau^2)$ steps: for each $\xrightarrow{\tau}$-successor s'' of s we have to try all $\xrightarrow{\tau}$-successors of s''. This can be conveniently done using the list of outgoing τ-transitions from s' and s''. The check whether $s'' \xrightarrow{a} s'''$ is a single hash table lookup.

Every transition $s \xrightarrow{a} s'$ is put at most $Fanout_\tau + 1$ times on the stack: initially and each time the variable `candidate` of a τ-successor of s' is reset. For m transitions this leads to the upper bound: $\mathcal{O}(m\, Fanout_\tau^3)$. □

Note that it requires $\mathcal{O}(m\, Fanout_\tau^2)$ to check whether a labelled transition system is τ-confluent with respect to all its τ-transitions. As determining the set T_{conf} seems more difficult than determining global τ-confluence, and we only require a factor $Fanout_\tau$ to do so, we expect that the complexity of our algorithm cannot be improved. We have looked into establishing other forms of partial τ-confluence (cf. [10]), especially forms where, given $s \xrightarrow{a} s'$ and $s \xrightarrow{\tau} s''$, it suffices to find some state s''' such that $s' \xrightarrow{\tau^*} s'''$ and $s'' \xrightarrow{\tau^* a \tau^*} s'''$. However, doing this requires the dynamic maintenance of the transitive τ-closure relation, which we could not perform in a sufficiently efficient manner to turn it into an effective preprocessing step for branching bisimulation.

5 τ-Priorisation and τ-Compression

After the set T_{conf} for a labelled transition system A has been determined we can "harvest" by applying τ-priorisation and calculating a form of τ-compression. Both operations can be applied in linear time, and moreover, reduce the state space. The τ-priorisation operation allows to give precedence to silent steps, provided they are confluent. This is defined as follows:

Definition 7. *Let $A = (S, Act, \longrightarrow_A)$ be a labelled transition system and let T be a set of τ-transitions of A. We say that a transition system $B = (S, Act, \longrightarrow_B)$ is a* τ-priorisation *of A with respect to T iff for all $s, s' \in S$ and $a \in Act$*

- *if $s \xrightarrow{a}_B s'$ then $s \xrightarrow{a}_A s'$, and*
- *if $s \xrightarrow{a}_A s'$ then $s \xrightarrow{a}_B s'$ or for some $s'' \in S$ it holds that $s \xrightarrow{\tau}_B s'' \in T$.*

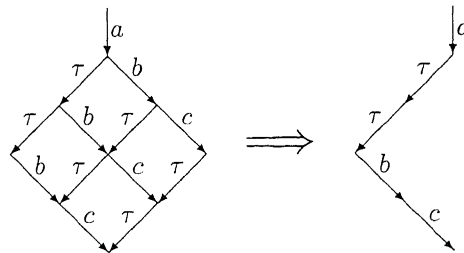
It holds that τ-priorisation maintains branching bisimulation.

Theorem 3. *Let $A = (S, Act, \longrightarrow_A)$ be a convergent, labelled transition system, which is T-confluent for some silent transition set T. Let $B = (S, Act, \longrightarrow_A)$ be a τ-priorisation of A w.r.t. T. Then for each state $s \in S$, $s \leftrightarrow_b s$ on $A \times B$.*

Proof. (sketch) The auto-bisimulation $\leftrightarrow_b^A$ on $A \times A$ is also a branching bisimulation relation on $A \times B$. This is proved using an auxiliary lemma, which ensures that if $s \leftrightarrow_b^A t$, $s \xrightarrow{\tau^*}_A s' \xrightarrow{a}_A s''$ and $s \leftrightarrow_b^A s'$, then for some t' and t'', $t \xrightarrow{\tau^*}_B t' \xrightarrow{a}_B t''$, $t \leftrightarrow_b^A t'$ and $s'' \leftrightarrow_b^A t''$. This is proved by induction on $\xrightarrow{\tau}$. □

Example 1. Convergence of a labelled transition system is a necessary precondition. Let a labelled transition system A be given with single state s and two transitions: $\{s \xrightarrow{\tau} s,\ s \xrightarrow{a} s\}$. It is clearly confluent, but not convergent. The τ-priorisation is a single τ-loop, which is not branching bisimilar with A.

The τ-priorisation w.r.t. a given set T of transitions can be computed in linear time, by traversing all states, and if there is an outgoing T-transition, removing all other outgoing transitions. Consider the labelled transition system below. All τ-transitions are confluent, and τ-priorisation removes more than half of the transitions.



A typical pattern in τ-prioritised transition systems are long sequences of τ-steps that can easily be removed. We call the operation doing so τ-compression.

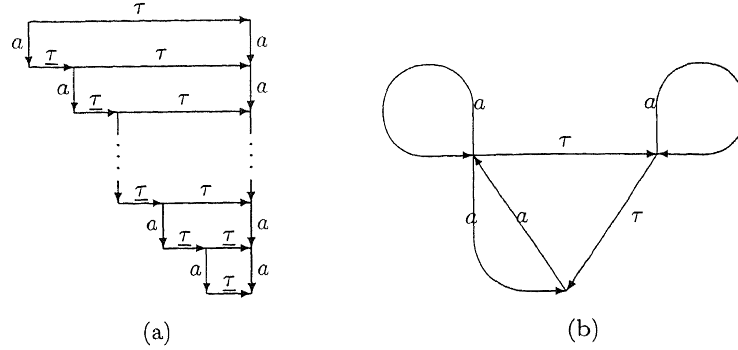


Fig. 2. The effect of repetition

Definition 8. *Let $A = (S, Act, \longrightarrow)$ be a convergent labelled transition system. For each state $s \in S$ we define with well-founded recursion on $\xrightarrow{\tau}$ the* τ-descendant *of s, notation $\tau^*(s)$, as follows: $\tau^*(s) = \tau^*(s')$ if for some s', $s \xrightarrow{\tau}_A s'$ is the only transition leaving s, and $\tau^*(s) = s$ otherwise. The τ-compression of A is the transition system $A_F = (S, Act, \longrightarrow_{A_F})$ where $\longrightarrow_{A_F} = \{\langle s, a, \tau^*(s')\rangle \mid s \xrightarrow{a}_A s'\}$.*

Theorem 4. *Let $A = (S, Act, \longrightarrow)$ be a labelled transition system and A_F the τ-compression of A. Then for all $s \in S$, $s \leftrightarroweq_b s$ on $A \times A_F$.*

Proof. (sketch) It can be shown that $R = \{\langle s, s\rangle \mid s \in S\} \cup \{\langle s, \tau^*(s)\rangle \mid s \in S\}$ is a branching bisimulation. □

Note that the τ-compression can be calculated in linear time. During a depth first sweep $\tau^*(s)$ can be calculated for each state s. Then by traversing all transitions, each transition $s \xrightarrow{a} s'$ can be replaced by $s \xrightarrow{a} \tau^*(s')$.

6 The Full Algorithm and Benchmarks

In this section we summarize all operations in a complete algorithm and give some benchmarks to indicate its usefulness. Note that τ-compression can make new diagrams confluent (in fact we discovered this by experimenting with the implementation). Therefore, we present our algorithm as a fixed point calculation, starting with a transition system $A = (S, Act, \longrightarrow)$:

$B{:=}A_\otimes$;
repeat
  $n{:=}\#$states of B;
  calculate T_{conf} for B;
  apply τ-priorisation to B w.r.t. T_{conf};
  apply τ-compression to B;
while $n \neq \#$states of B;

The example in Figure 2.a shows that in the worst case the loop in the algorithm must be repeated $\Omega(n)$ times for a labelled transition system with n states. Only the underlined τ-transitions are confluent. Therefore, each subsequent iteration of the algorithm removes the bottom tile, connecting the two arrows in the one but last line.

An improvement, which we have not employed, might be to apply strong bisimulation reductions, in this algorithm. As shown by [18] strong bisimulation can be calculated in $\mathcal{O}((m+n)\log n)$, which although not being linear, is quite efficient. Unfortunately, Figure 2.b shows an example which is minimal with respect to all mentioned reductions and strong bisimulation, but not with respect to branching bisimulation.

In order to understand the effect of partial confluence checking we have applied our algorithm to a number of examples. We can conclude that if the number of internal steps in a transition system is relatively low, and the number of equivalence classes is high, our algorithm performs particularly well compared to the best implementation [3] of the standard algorithm for branching bisimulation [11]. Under less favourable circumstances, we see that the performance is comparable with the implementation in [3].

In Table 1 we summarize our experience with 5 examples. In the rows we list the number of states "n" and transitions "m" of each example. The column under "n (red1)" indicates the size of the state space after 1 iteration of the algorithm. "#iter" indicates the number of iterations of the algorithm to stabilize, and the number of states of the resulting transition system is listed under "n (red tot)". The time to run the algorithm for partial confluence checking is listed under "time conf". The time needed to carry out branching bisimulation reduction and the size of the resulting state space are listed under "time branch" and "n min", respectively. The local confluence reduction algorithm was run on a SGI Powerchallenge with a 300MHz R12000 processor. The branching bisimulation reduction algorithm was run on a 300MHz SUN Ultra 10.

The Firewire benchmark is the firewire or IEEE 1394 link protocol with 2 links and a bus as described in [15,20]. The SWP1.2 and SWP1.3 examples are sliding window protocols with window size 1, and with 2 and 3 data elements, respectively. The description of this protocol can be found in [2]. The processes PAR2.12 and PAR6.7 are specially constructed processes to see the effect of the relative number of τ-transitions and equivalence classes on the branching bisimulation and local confluence checking algorithms. They are defined as follows:

$$\text{PAR2.12} = \mathop{\Big\|}_{i=1}^{12} \tau\, a_i \qquad\qquad \text{PAR6.7} = \mathop{\Big\|}_{i=1}^{7} \tau\, a_i\, b_i\, c_i\, d_i\, e_i$$

Note that in these cases, partial confluence checking finds a minimal state space w.r.t. branching bisimulation.

Table 1. Benchmarks showing the effect of partial τ-confluence checking

	n	m	n (red1)	#iter	n (red tot)	time conf	n min	time branch
Firewire	372k	642k	46k	4	23k	3.6s	2k	132s
SWP1.2	320k	1.9M	32k	6	960	13s	49	9s
SWP1.3	1.2M	7.5M	127k	6	3k	57s	169	136s
PAR2.12	531k	4.3M	4k	2	4k	98s	4k	64s
PAR6.7	824k	4.9M	280k	2	280k	55s	280k	369s

References

1. A.V. Aho, J.E. Hopcroft and J.D. Ullman. Data structures and algorithms. Addison-Wesley. 1983.
2. M.A. Bezem and J.F. Groote. A correctness proof of a one bit sliding window protocol in μCRL. The Computer Journal, 37(4): 289-307, 1994.
3. M. Cherif and H. Garavel and H. Hermanns. The bcg_min user manual, version 1.1. `http://www.inrialpes.fr/vasy/cadp/man/bcg_min.html`, 1999.
4. D. Dill, C.N. Ip and U. Stern. Murphi description language and verifier. `http://sprout.stanford.edu/dill/murphi.html`, 1992-2000.
5. H. Garavel and R. Mateescu. The Caesar/Aldebaran development package. `http://www.inrialpes.fr/vasy/cadp/`, 1996-2000.
6. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. In *Journal of the ACM*, 43(3):555–600, 1996.
7. P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305-326, 1994.
8. J.F. Groote and B. Lisser. The μCRL toolset. `http://www.cwi.nl/~mcrl`, 1999-2000.
9. J.F. Groote and J.C. van de Pol. *State space reduction using partial τ-confluence.* Technical Report CWI-SEN-R0008, March 2000. Available via `http://www.cwi.nl/~vdpol/papers/`.
10. J.F. Groote and M.P.A. Sellink. Confluence for process verification. In Theoretical Computer Science B, 170(1-2):47-81, 1996.
11. J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proc. 17th ICALP*, LNCS 443, 626–638. Springer-Verlag, 1990.
12. P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43-68, 1990.
13. N. Kobayashi, B.C. Pierce, and D.N. Turner. Linearity and the π-calculus. In: *Proceedings of the 23rd POPL*, pages 358-371. ACM press, January 1996.
14. X. Liu and D. Walker. Confluence of processes and systems of objects. In *Proceedings of TAPSOFT'95*, pages 217-231, LNCS 915, 1995.
15. S.P. Luttik. Description and formal specification of the link layer of P1394. Technical Report SEN-R9706, CWI, Amsterdam, 1997.
16. R. Milner. *Communication and Concurrency.* Prentice Hall International. 1989.
17. U. Nestmann and M. Steffen. Typing confluence. In: *Proceedings of FMICS'97*, pages 77-101. CNR Pisa, 1997.

18. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973-989, 1987.
19. A. Philippou and D. Walker. On confluence in the pi-calculus. *24th Int. Coll. on Automata, Languages and Programming*, LNCS 1256, Springer-Verlag, 1997.
20. M. Sighireanu and R. Mateescu. Verification of the link layer protocol of the IEEE-1394 serial bus (firewire): an experiment with E-LOTOS. In *Journal on Software Tools for Technology Transfer (STTT)*, 2(1):68–88, 1998.
21. A. Valmari. A stubborn attack on state explosion. In *Proc. of Computer Aided Verification*, LNCS 531, pages 25-42, Springer-Verlag, 1990.