State Space Reduction using Partial \(\tau\)-Confluence

Jan Friso Groote
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Department of Mathematics and Computing Science, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
Email: JanFriso.Groote@cwi.nl

Jaco van de Pol
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Email: Jaco.van.de.Pol@cwi.nl

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

2000 Mathematics Subject Classification: 68Q20, 68Q50.
Keywords and Phrases: Confluence, State Space, Reduction, Branching Bisimulation, Labelled Transition System.

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 that are involved, and with a limited number of data that is exchanged. Subsequently, generate the state space of this system and reduce it using an appropriate equivalence, for which weak or branching bisimulation 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). Of course correctness of the full distributed system does not follow in this way. But by taking the number of involved parties as well as the data domains as large as possible, a good impression of the behaviour can be obtained and many of its problems are exposed.

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 super 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 complexity of computing the minimal branchingbisimilar state space is \(O(nm)\) [10] and computing the minimal weakly bisimilar one is approximately \(O(n^\alpha)\) where \(\alpha = 2.376\) is the typical constant required for matrix multiplication [11].

When the size of state spaces that can be handled grows, the time required to reduce these grows with an extra factor. Currently, state spaces that can be generated and handled on common computers contain in the order of \(10^6\) states. In the coming years we expect this number to grow to \(10^8\) and beyond. Assuming that the number of transitions are of the same order as the number of states and considering those cases where the complexity measure of the algorithm is tight, an increase in time of a factor \(10^5\) for branching bisimulation can be expected. This is not easily matched by an increase of the speed of processors.

We introduce a state space reduction algorithm of complexity \(O(m \text{ Fanout}_\tau^2)\) where \(\text{Fanout}_\tau\) is the maximal number of outgoing \(\tau\)-transitions of a node in the transition system. Assuming that for certain classes of transition systems \(\text{Fanout}_\tau\) is constant, our procedure is linear in the size of the transition system.
The reduction procedure is based on the detection of $\tau$-confluence. Roughly, we call a $\tau$-transition from a state $s$ confluent if for each $a$-transition starting in $s$, we can get again to the same state via an $a$ and a confluent $\tau$-transition, respectively. If the maximal class of confluent $\tau$-transitions has been determined in this way, $\tau$-priorisation is applied. This means that from each state with an outgoing confluent $\tau$, all other outgoing transitions may be removed, giving “priority” to the $\tau$-transition. In this way large parts of the transition system can become unreachable, which in some cases 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 both [15] and [16] a chapter is devoted to determinate and confluent processes, showing that certain operators preserve confluence, and showing how confluence can be used to verify certain processes. In [13] these notions have been extended to the $\pi$-calculus. In [9] an extensive investigation into various notions of global confluence for processes is given, where it is shown that by applying $\tau$-priorisation, state spaces could be reduced substantially. In particular the use of confluence for verification purposes in the context of linear process operators was discussed. Linear process operators can be viewed as a symbolic representation of state spaces, into which huge or infinite state spaces can easily be encoded. The advantage of this approach is that confluence and $\tau$-priorisation can be employed before the generation of state spaces, avoiding the generation of state spaces that are unnecessarily large. 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. The goal of this is also to enable $\tau$-priorisation before the generation of state spaces. In [12] such typing schemes are extended to the $\pi$-calculus.

Partial order reductions [7, 20] on the one hand, and $\tau$-confluence and $\tau$-priorisation on the other are strongly related. The idea behind partial order reductions is that a priori some property is established that must be checked on a transition system, together with an independence relation between certain actions. Based on the property it is established which actions can be viewed as internal, or $\tau$. The independence relation on actions then determines a subset of $\tau$-transitions for which the system is $\tau$-confluent. Now the property can be checked by carrying out $\tau$-priorisation, which in other words can be phrased as a reduction based on the partial ordering of actions.

Our primary contribution consists of providing an algorithm that determines the maximal set of confluent $\tau$-transitions for a given transition system. This differs from the work in [9] which is only applicable if all $\tau$-transitions are confluent, which is generally not the case. It also differs from approaches that use type systems or independence relations, in order to determine a subset of the confluent $\tau$-transitions, given a high level description of the transition system. These methods are incapable of determining the maximal set of confluent $\tau$-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 $\tau$-loop elimination, and $\tau$-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. This happens when the number of classes of equivalent states in a transition system is small (typically up to 100 classes) and there are many hidden $\tau$-transitions in the transition system. Under these circumstances the branching bisimulation algorithm performs well, whereas our algorithm is relatively weak.

But under more favourable conditions, with many equivalence classes, and many visible transitions, our algorithm reduced state spaces with factors varying from 10 to 1000 within seconds, being a substantial improvement over the time branching bisimulation requires.

In section 2 we introduce elementary notions, including the definition of confluence. In section 3 we define the removal of $\tau$-loops as a necessary preprocessing step of our algorithm. In section 4 we explain our algorithm to determine the maximal class of $\tau$-confluent transitions, and prove its correctness and complexity. In section 5 we explain $\tau$-priorisation and $\tau$-compression and in section 6 we combine all parts into a full algorithm and present some benchmark results.

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 A labelled transition system is a three tuple $A = (S, Act, \rightarrow)$.
where

- $S$ is a set of states;
- $\text{Act}$ is a set of actions. We assume there is a special internal action $\tau$ that is always included in $\text{Act}$;
- $\rightarrow \subseteq S \times \text{Act} \times S$ is a transition relation.

We write $s \xrightarrow{a} t$ for the binary relation $\{(s,t) \mid (s,a,t) \in \rightarrow\}$. 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 $t \xleftarrow{a} s$ iff $t \xrightarrow{a} s$ and $s \xrightarrow{\tau} t$, i.e. $s$ and $t$ lie on a $\tau$-loop. Finally, we write $s \xrightarrow{a} s'$ if either $s \xrightarrow{a} s'$, or $s = s'$ and $a = \tau$.

A set $T \subseteq S$ is called a silent transition set of $A$. We write $s \xrightarrow{\tau_T} t$ iff $(s,t) \in T$. With $s \xrightarrow{\tau_T} t$ we denote $s = t$ or $s \xrightarrow{\tau_T} t$. We define the set $\text{Fanout}_\tau(s)$ for a state $s$ by: $\text{Fanout}_\tau(s) = \{ s \xrightarrow{a} s'| s|s' \in S \}$.

We say that $A$ is finite if $S$ and $\text{Act}$ 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 $m_\tau$ denotes the number of $\tau$-transitions. Furthermore, we write $\text{Fanout}_\tau$ for the maximal size of the set $\text{Fanout}_\tau(s)$.

Definition Let $A = (S, \text{Act}, \rightarrow)$ 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 $s \xrightarrow{a} s''$ ($a \in \text{Act}$) there exists a state $s''' \in S$ such that $s' \xrightarrow{a} s'''$, $s'' \xrightarrow{\tau_T} s'''$. We call $A$ confluent iff $A$ is $\tau$-confluent.

Definition Let $A = (S_A, \text{Act}, \rightarrow_A)$ and $B = (S_B, \text{Act}, \rightarrow_B)$ 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 there exist $t'$ and $t''$, such that $t \xrightarrow{\tau_B} t'$, $\xrightarrow{a} \rightarrow_B t''$ and $sRt'$ and $s'Rt''$.
2. If $t \xrightarrow{a} B t'$, then there exist $s'$ and $s''$, such that $s \xrightarrow{a} A s'$, $s'' \xrightarrow{\tau_A} s''$ and $sRt'$ and $s''Rt''$.

For states $s \in S_A$ and $t \in S_B$ we write $s \equiv_B t$, 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, $\equiv_B$ itself is the maximal branching bisimulation, and it is an equivalence relation. In the special case that $A$ and $B$ are the same transition systems, we say that $R$ is a branching auto-bisimulation on $A$. For $s, s' \in S_A$ we write $s \equiv_B s'$ iff there is a branching auto-bisimulation $R$ on $A$ such that $sRs'$.

Definition Let $A = (S, \text{Act}, \rightarrow)$ be a labelled transition system. A transition $s \xrightarrow{\tau} s'$ is called inert if $s \equiv_B A s'$.

Theorem Let $A = (S, \text{Act}, \rightarrow)$ 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 We show that the relation $R = \xrightarrow{\tau_T}$ is a branching bisimulation relation. Note that the theorem is a direct consequence of this result. So, let $sRs'$. The case where $s = s'$ is trivial. Consider the case where $sRb'$ because $s \xrightarrow{\tau_T} s'$. We prove the two cases of branching bisimulation:

Assume $s \xrightarrow{a} t$. So, as $A$ is $T$-confluent, there exists some $t' \in S$ such that $t \xrightarrow{a} t'$ and $sRt'$ and $t'Rt'$. Clearly, $s' \xrightarrow{a} s'$ and $s' \xrightarrow{\tau} t'$ and $sRs'$ and $t'Rt'$.

Now assume $s' \xrightarrow{a} t'$. Then $s \xrightarrow{a} s' \xrightarrow{\tau} t'$ and $sRs'$ and $t'Rt'$.

Lemma Let $A$ be a labelled transition system. There exists a largest silent transition set $T_{\text{conf}}$ of $A$, such that $A$ is $T_{\text{conf}}$-confluent. Proof Consider the set $T$, being the set of all silent transition sets $T$ such that $A$ is $T$-confluent. Define $T_{\text{conf}} = \bigcup T$. We first show that $A$ is $T_{\text{conf}}$-confluent. So, let $s \xrightarrow{a} T_{\text{conf}} s'$ and $s \xrightarrow{a} s''$, then by definition of $T_{\text{conf}}$, for some $T \in T$, $s \xrightarrow{\tau_T} s'$. By confluence of $T$, we find $s''$, such that $s' \xrightarrow{\tau} s''$ and $s'' \xrightarrow{\tau_T} s''$. By definition of $T_{\text{conf}}$, $s'' \xrightarrow{\tau_T} T_{\text{conf}} s''$. Now consider any $T$ such that $A$ is $T$-confluent. By definition of $T_{\text{conf}}$, $T \subseteq T_{\text{conf}}$. So $T_{\text{conf}}$ is indeed the largest.

Definition Let $A = (S, \text{Act}, \rightarrow)$ be a labelled transition system. We call $A$ ($\tau$-)convergent if there does not exist an infinite sequence of $\tau$ steps, i.e. there does not exist $s_1, s_2 \ldots \in S$ such that $s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \ldots$. We will employ well-founded recursion and induction over $\xrightarrow{\tau}$ for convergent systems.

3. Elimination of $\tau$-cycles
In this section we define the removal of $\tau$-loops from a transition system. The idea is to collapse each loop to a single state. This can be done, because $\xrightarrow{\tau}$ is an equivalence relation on states. Definition
Let \( A = (S, Act, \rightarrow) \) be a labelled transition system. We write for any state \( s \in S_A \):

\[
[s]_A = \{ t \in S \mid t \xrightarrow{\tau} s \}
\]

Moreover, we define the relation \([\rightarrow]_A\) by:

\[
S[A \rightarrow S'] \iff \exists s \in S, s' \in S' \text{ such that } s \xrightarrow{a} s', \text{ and } s \neq s' \text{ or } a \neq \tau.
\]

We write \([S]_A\) for \(\{[s]_A \mid s \in S\}\) and \([T]_A\) for \(\{[s]_A [\xrightarrow{a}]_A[t]_A \mid s \xrightarrow{a} t \in T\}\). **Definition** Let \( A = (S, Act, \rightarrow) \) be a labelled transition system. The \(\tau\)-cycle reduction of \( A \) is the labelled transition system \( A_{\odot} = ([S]_A, Act, [\rightarrow]_A) \). Using the algorithm to detect strongly connected components [1] it is possible to construct the \(\tau\)-cycle reduction of a labelled transition system \( A \) in linear time. **Lemma** Let \( A = (S, Act, \rightarrow) \) be a labelled transition system and let \( A_{\odot} \) be its \(\tau\)-cycle reduction. Then for every state \( s \in S, s \in A \) is branching bisimilar to \([s]_A\) in \( A_{\odot} \). We show below that taking the \(\tau\)-cycle reduction can change the confluence structure of a process. The first example shows that it can happen that some \(\tau\)-transitions not being confluent, can become confluent after \(\tau\)-cycle reduction.

![Diagram](image)

The transition \( s_1 \xrightarrow{\tau} s_2 \) is not confluent before \(\tau\)-cycle reduction, but it is afterwards. Of course, as the number of \(\tau\)-transitions in a \(\tau\)-cycle reduction of \( A \) can be much smaller than the number of \(\tau\)-transitions in \( A \), the absolute number of confluent \(\tau\)'s can be strongly reduced. But the following examples show that the situation is actually worse. Some \(\tau\)-transitions that are confluent may have lost that property after \(\tau\)-cycle reduction. Consider for instance

![Diagram](image)

Observe that all \(\tau\)-transitions are confluent. If \(\tau\)-cycle reduction is applied, then states \( s_1 \) and \( s_2 \) are taken together, and we see that transitions \( \{s_1, s_2\} \xrightarrow{\tau} \{s_3\} \) and \( \{s_1, s_2\} \xrightarrow{\tau} \{s_5\} \) exist, but there is no way to complete the diagram. So, in the \(\tau\)-cycle reduction, \( \{s_1, s_2\} \xrightarrow{\tau} \{s_3\} \) is not confluent.

We can extend the example slightly, showing that states that have outgoing confluent \(\tau\)-transitions before \(\tau\)-cycle reduction, do not have these afterwards:

![Diagram](image)

So, before \(\tau\)-cycle reductions all \(\tau\)-transitions are confluent. In particular state \( s_6 \) has an outgoing \(\tau\)-transition. After \(\tau\)-cycle reduction, \( \{s_1, s_2\} \xrightarrow{\tau} \{s_3\} \) is not confluent anymore, and consequently, \( \{s_6\} \xrightarrow{\tau} \{s_7\} \) is not, too. So, state \( s_6 \) has lost the property of having an outgoing confluent \(\tau\)-transition. Nevertheless, \(\tau\)-cycle reduction is unavoidable in view of Example 5.
4. Algorithm to Calculate $T_{\text{conf}}$

We now present an algorithm to calculate $T_{\text{conf}}$ for a given labelled transition system $A = (S, Act, \rightarrow)$. First the required data structures and their initialization are described:

1. Each transition $s \xrightarrow{a} s'$ is equipped with a boolean candidate that is initially set to true, and which indicates whether this transition is still a candidate to be put in $T_{\text{conf}}$.

2. For every state $s$ we store a list of all incoming transitions, as well as a list with outgoing $\tau$-transitions.

3. Each transition $s \xrightarrow{a} s'$ is stored in a hashtable, such that given $s$, $a$ and $s'$, it can be found in constant time, if it exists.

4. Finally, there is a stack on which transitions are stored, in such a way that membership can be tested in constant time. A transition $s \xrightarrow{\tau} s'$ is on the stack if confluence of $s \xrightarrow{a} s'$ must be checked with regard to $\tau$-transitions $s \xrightarrow{\tau} s''$. Initially, all transitions are put on the stack.

The algorithm 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 $\tau$-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

- there exist an $s''' \in S$ such that $s' \xrightarrow{\tau} s'''$ with the variable candidate set to true, and there is a transition $s'' \xrightarrow{a} s'''$. Note that as there is a list of outgoing $\tau$-transitions from $s'$ and all transitions $s'' \xrightarrow{a} s'''$ are stored in the hashtable, carrying out this check takes time proportional to $\text{Fanout}_{\tau}$. Or,
- $s'' \xrightarrow{a} s'$. This can be checked in constant time using the hash table. 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 $s \xrightarrow{a} s'$ 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 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 $T_{\text{alg}}$ is formed by all $\tau$-transitions for which the variable candidate is still true. Termination of the algorithm follows directly from the following observations:

- 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 $T_{\text{alg}} = T_{\text{conf}}$.

Lemma A is $T_{\text{alg}}$-confluent. Proof Consider transitions $s \xrightarrow{a} s'$ and $s \xrightarrow{\tau} s''$. Consider the last step, with index say $n$, in the algorithm where $s \xrightarrow{a} s'$ is removed from the stack. As the variable candidate of $s \xrightarrow{\tau} s''$ was never set to false, it was established either that

- $a = \tau$ 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
- there exists an $s''' \in S$ such that $s' \xrightarrow{\tau} s'''$ that 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 $T_{\text{alg}}$-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 $T_{\text{alg}}$. 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 $s \xrightarrow{\tau} s'$ is put on the stack to be removed at some step $n'' > n'$, respectively, has been reset. But this means
that $n$ was not the last step of the algorithm where $s \xrightarrow{a} s'$ was removed from the stack, contradicting the assumption.

**Theorem 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 wrt. some $s \xrightarrow{a} s''$ fails. By $T_{conf}$-confluence, for some $s'''$, $s'' \xrightarrow{\tau} T_{conf} s'''$ and $s' \xrightarrow{\tau} s'''$. 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** The algorithm terminates in $O(m \text{ Fanout}_T^3)$ steps. Proof Checking that a single transition $s \xrightarrow{a} s'$ is confluent, requires $O(\text{Fanout}_T^2)$ steps (for each $\xrightarrow{\tau}$-successor $s''$ of $s$ we have to try the $\xrightarrow{\tau}$-successors of $s''$).

Every transition $s \xrightarrow{a} s'$ is put at most Fanout, + 1 times on the stack: initially and each time a the variable candidate of a $\tau$-successor $s''$ of $s'$ is reset. As there are $m$ transitions, this leads to a total amount of time of: $O(m \text{ Fanout}_T^3)$. Note that it requires $O(m \text{ Fanout}_T^2)$ to check whether a labelled transition system is $\tau$-confluent with respect to all its $\tau$-transitions. As determining the set $T_{conf}$ is more expensive than determining global $\tau$-confluence, and we only require a factor $\text{Fanout}_T$ to do so, we expect that the complexity of our algorithm cannot be improved.

We have looked into establishing other forms of partial $\tau$-confluence (cf. [9]), especially forms where if there are transitions $s \xrightarrow{a} s'$ and $s \xrightarrow{\tau} s''$, there exists some state $s'''$ such that $s' \xrightarrow{\tau} s'''$ and $s'' \xrightarrow{\tau} s'''$. However, doing this requires the dynamic maintenance of the transitive $\tau$-closure relation, which we could not perform in a sufficiently efficient manner to turn it into an effective preprocessing step for branching bisimulation.

5. $\tau$-priorisation and $\tau$-compression

After the set $T_{conf}$ for a labelled transition system $A$ has been determined we can “harvest” by applying $\tau$-priorisation and calculating a form of $\tau$-compression. Both operations can be applied in linear time, and moreover, reduce the state space.

The $\tau$-priorisation operation allows to give precedence to silent steps, provided they are confluent. This is defined as follows: **Definition** Let $A = (S, Act, \rightarrow_A)$ be a labelled transition system and let $T$ be a set of $\tau$-transitions of $A$. We say that a transition system $B = (S, Act, \rightarrow_B)$ is a $\tau$-priorisation of $A$ with respect to $T$ if 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 either $s \xrightarrow{a} B s'$, or for some $s'' \in S$ it holds that $s \xrightarrow{\tau} B s'' \in T$.

The following theorem implies that $\tau$-priorisation maintains branching bisimulation. **Lemma** Let $A = (S, Act, \rightarrow_A)$ be a convergent, labelled transition system, which is $T$-confluent for some silent transition set $T$. Let $B = (S, Act, \rightarrow_B)$ be a $\tau$-priorisation of $A$ wrt. $T$. Let $s \xrightarrow{A} t$. If $s \xrightarrow{\tau}_A s' \xrightarrow{a} A s''$ and $s \in \text{conf} A s'$, then for some $t' \xrightarrow{\tau} B t' \xrightarrow{a} A t''$ and $t' \xrightarrow{A} t''$. Proof Well-founded induction on $\xrightarrow{A}$. Assume $s \in \text{conf} A t$ and $s \xrightarrow{\tau}_A s' \xrightarrow{a} A s''$ and $s \in \text{conf} A s'$. By transitivity, $s \xrightarrow{A} t$. Hence for some $u' \xrightarrow{A} t$ and $u'' \xrightarrow{A} s''$, we have $t \xrightarrow{\tau}_A u' \xrightarrow{A} t$. We get two cases according to $t \xrightarrow{\tau}_A u'$:

- $t = u'$. As $B$ is a $\tau$-priorisation of $A$, we have two new cases:
  - either $u' \xrightarrow{\pi} B u''$. Then the theorem holds with $t' = t$ and $t'' = u''$.
  - or $u' \xrightarrow{\tau}_B u''$. By Theorem 2, $u' \xrightarrow{A} u''$, so $s \xrightarrow{A} u''$. As $t \xrightarrow{\tau}_A u''$, the induction hypothesis applies, so for some $t' \xrightarrow{A} u''$ and $t'' \xrightarrow{A} s''$, $t = u' \xrightarrow{\tau}_B u'' \xrightarrow{\tau}_B t' \xrightarrow{\pi} B t''$.

- $t \xrightarrow{\tau}_A t'' \xrightarrow{A} u'$. As $B$ is a $\tau$-priorisation of $A$, we have again two cases:
  - Either $t \xrightarrow{\tau}_B t''$. By the stuttering lemma [6], we know $t \xrightarrow{B} t'''$.
  - Or $t \xrightarrow{\tau}_B u'''$ for some $u'''$. By Theorem 2, $t \xrightarrow{B} u'''$. 

Equational Binary Decision Diagrams
In both cases we have some \( v \) with \( s \triangleleft k^A v \) and \( t \xrightarrow{\tau} B v \), so the induction hypothesis applies, which yields \( t' \triangleleft k^A v \) and \( t'' \triangleleft k^A s'' \) such that \( t \xrightarrow{\tau} B v \xrightarrow{\tau} B t' \xrightarrow{\tau} B t'' \).

**Theorem** Let \( A = (S, Act, \rightarrow_A) \) be a convergent, labelled transition system, which is \( T \)-confluent for some silent transition set \( T \). Let \( B = (S, Act, \rightarrow_B) \) be a \( \tau \)-priorisation of \( A \) wrt. \( T \). Then the auto-bisimulation \( \triangleleft k^A \) is a branching bisimulation relation on \( A \) and \( B \). **Proof** Let \( s \triangleleft k^A t \). We have to prove the obligations of branching bisimulation between \( A \) and \( B \):

- If \( s \xrightarrow{a} A s' \), then for some \( t' \triangleleft k^A s \) and \( t'' \triangleleft k^A s' \), we have \( t \xrightarrow{\tau} B v \xrightarrow{\tau} B t' \xrightarrow{\tau} B t'' \); and
- If \( t \xrightarrow{a} B t' \), then for some \( s' \triangleleft k^A t \) and \( s'' \triangleleft k^A t' \), we have \( s \xrightarrow{\tau} A s' \xrightarrow{\tau} A s'' \).

The first one follows from the previous lemma, as \( s \xrightarrow{\tau} s \). For the second one, let \( t \xrightarrow{a} B t' \), then as \( B \) is a \( \tau \)-priorisation of \( A \) also \( t \xrightarrow{a} A t' \). Now (2) trivially follows from the fact that \( s \triangleleft k^A t \).

**Example** Convergence of a labelled transition system is a necessary precondition. Consider the following \( \{ s \xrightarrow{\tau} s \} \)-confluent transition system, which is clearly not convergent.

![Diagram](image_url)

The \( \tau \)-priorisation with respect to \( \{ s \xrightarrow{\tau} s \} \) consists of a single \( \tau \)-loop. Clearly, state \( s \) before and after \( \tau \)-priorisation are not branching bisimilar. There is a straightforward linear algorithm that applies \( \tau \)-priorisation wrt. a given set \( T \) of transitions to a labelled transition system. Just traverse all states. For each state check whether there is an outgoing \( \tau \)-transition in \( T \). If so, select one such \( \tau \)-transition and remove all other transitions that start in this state. **Example** Consider the labelled transition system below. All \( \tau \)-transitions are confluent, and \( \tau \)-priorisation removes more than half of the transitions.

![Diagram](image_url)

Note also that a typical pattern in \( \tau \)-prioritised transition systems are long sequences of \( \tau \)-steps that can easily be removed. We call the operation doing so \( \tau \)-compression.

**Definition** Let \( A = (S, Act, \rightarrow) \) be a convergent labelled transition system. For each state \( s \in S \) we define with well-founded recursion \( \rightarrow \) the \( \tau \)-descendant of \( s \), notation \( \tau^*(s) \), as follows:

\[
\tau^*(s) = \begin{cases} 
\tau^*(s') & \text{if } s \xrightarrow{\tau} A s' \text{ and for all } a \in Act, s'' \in S, \text{ if } s \xrightarrow{a} s'' \text{ then } a = \tau \text{ and } s' = s'', \\
\{s\} & \text{otherwise.}
\end{cases}
\]

The \( \tau \)-compression of \( A \) is the labelled transition system \( A_F = (S, Act, \rightarrow_{A_F}) \) where

\[
\rightarrow_{A_F} = \{ (s, a, \tau^*(s')) | s \xrightarrow{a} A s' \}.
\]

**Theorem** Let \( A = (S, Act, \rightarrow) \) be a labelled transition system and \( A_F \) the \( \tau \)-compression of \( A \). Then \( R = \{ (s, s) | s \in S \} \cup \{ (s, \tau^*(s)) | s \in S \} \) is a branching bisimulation relation on \( A \) and \( A_F \). **Proof** Let \( sRt \). We have two cases:

- If \( t = s \). First, assume \( s \xrightarrow{a} A s' \), then by definition, \( s \xrightarrow{a} A_F \tau^*(s') \) and \( s'R \tau^*(s') \). Conversely, if \( s \xrightarrow{a} A_F s' \), then by definition of \( A_F \), for some \( s'' \) we have \( s' = \tau^*(s'') \) and \( s \xrightarrow{a} A s'' \). This proves both obligations of branching bisimulation in this case.
• $t = \tau^*(s)$. First, assume $s \xrightarrow{a} A s'$. We again have two cases:
  
  - $a = \tau$ and this is the only $\tau$-step from $s$. Then $\tau^*(s) = \tau^*(s')$ by definition of $\tau^*$, hence we have $\tau^*(s) \xrightarrow{A^r} \tau^*(s')$, viz. in zero steps.
  
  - In the other case $\tau^*(s) = s$, so the case $t = s$ applies.

Conversely, assume $\tau^*(s) \xrightarrow{a} A s'$. Then by definition of $A_F$, for some $s''$ we have $\tau^*(s'') = s'$ and $\tau^*(s) \xrightarrow{A} s''$. Of course, $s \xrightarrow{\tau} A \tau^*(s)$. As $\tau^*(s) R \tau^*(s)$ and $s'' R s'$, we finished the proof.

Note that the $\tau$-compression can be calculated in linear time by strictly following the definition. 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 parts that have hitherto been described in a complete algorithm and give some benchmarks to indicate how useful the algorithm is. When we had implemented the algorithm, our first observation was that iterative application of the algorithm led to repeated reduction of the state space. The reason is of course that $\tau$-compression can make new diagrams confluent. Therefore, we present our algorithm as a fixed point calculation, starting with a transition system $A = (S, \text{Act}, \xrightarrow{-})$:

$$B := A \otimes;$$

repeat

$$n := \#\text{states of } B;$$

calculate $T_{\text{conf}}$ for $B$;

apply $\tau$-priorisation to $B$ wrt. $T_{\text{conf}}$;

apply $\tau$-compression to $B$;

while $n \neq \#\text{states of } B$;

The following example 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.

**Example** In the labelled transition system below the underlined $\tau$-transitions are the confluent ones. Therefore, only the left $a$-transition at the bottom will be removed by $\tau$-priorisation. An application of $\tau$-compression will connect the two $\tau$-transitions at the one but last horizontal arrows. This will yield a new confluent diagram that will be removed by the subsequent iteration of the algorithm.

![Diagram](image)

A further improvement, which we have not employed, might be to apply strong bisimulation reductions, in this algorithm. We have seen examples where this gives a substantial further reduction. As shown by [18] strong bisimulation can be calculated in $O((m + n) \log n)$, which although not being linear, is quite efficient. An interesting question that arises immediately, is whether the combination of partial confluence checking, $\tau$-priorisation, $\tau$-reduction and strong bisimulation would yield a minimal transition system with respect to branching bisimulation. Unfortunately, this is not the case, as the following example shows, which is not minimal with respect to branching bisimulation, but which cannot be reduced using any of the reductions mentioned above, including the application of strong 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 [10]. 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 250/300MHz R12000 processors. 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 [14, 19]. 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 relative speed of the branching bisimulation and local confluence checking algorithms. They are defined as follows:

\[
\text{PAR2.12} = \bigotimes_{i=1}^{12} \tau a_i \delta \\
\text{PAR6.7} = \bigotimes_{i=1}^{7} \tau a_i c_i d_i e_i \delta 
\]

where δ is a constant representing deadlock.

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

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