Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Partial τ-confluence for efficient state space generation

S.C.C. Blom

REPORT SEN-R0123 AUGUST 2001

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Partial τ-confluence for Efficient State Space Generation

Stefan Blom
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
We extend the theory of partial order methods with a new notion of τ-confluence. Based on this new notion we define a reduced transition system, which is branching bisimilar to the original state space. We show that the new notion is preserved under various transformations. We present an algorithm which efficiently computes the reduced transition system from a symbolic representation.

2000 Mathematics Subject Classification: 68Q60, 68Q85
Keywords and Phrases: τ-confluence, state space generation, partial order reduction, labeled transition system, branching bisimulation
Note: Research carried out in SEN2 with financial support of the "Systems Validation Center".

TABLE OF CONTENTS

1. INTRODUCTION

One approach to the verification of distributed systems is based on an exhaustive state space exploration. This approach suffers from the well-known state-explosion problem. Much research is devoted to algorithms that generate a reduced, but essentially equivalent, state space. In this paper, we present a new reduction algorithm. The focus is on τ-confluence, which provides the theoretical foundation of the reduction algorithm.

Following [Mil89], several notions of τ-confluence have been proposed. All of these notions have in common that they use diagram properties to identify sets of τ-steps between equivalent states. In [GS96], it is defined what it means for the set of all τ-transitions to be weakly confluent. The formulation of this property requires convergence of τ-transitions. That is, all τ-sequences must be finite. In [Yin00] weak confluence is redefined in such a way that convergence is no longer needed. To distinguish the two notions, we will refer to the notion in [GS96] as ultra weak confluence. In [GP00], the notion of ultra weak confluence is extended to subsets of τ-transitions. Since this notion does not refer to all τ-transitions, we refer to it as a notion of partial τ-confluence.

A property of weakly confluent τ-transitions is that if there is a weakly confluent transition from one state to another these states are branching bisimilar [vGW96]. Thus, weak confluence induces an equivalence relation on the set of states. A branching bisimilar state space can be obtained by taking the set of equivalence classes

as nodes and allowing a transition from one class to another if there exists a transition from a node in the first class to a node in the second.

The reduced transition system defined above would be difficult to compute, if one would have to compute a whole equivalence class. Fortunately, it can be proven that if one chooses any state in the equivalence class one only needs to consider states that are reachable form the chosen state to compute the transitions of the equivalence class. However, the set of states that must be considered can still be large. It would be more efficient if we would only have to consider a single representative of each equivalence class. For the existing notion of strong τ-confluence, it is correct to use a single representative. For weak τ-confluence it is not. Unfortunately, strong τ-confluence a very restrictive property. Hence, we introduce the notion of confluent set of τ-transitions. This new notion of τ-confluence allows the use of a single representative for each equivalence class and is much less restrictive than strong τ-confluence.

1.1 Related work

Several *partial order reduction* algorithms that preserve branching bisimilarity have been proposed in the literature [Pel97]. These approaches also allow the reduction to a representative subset of all states. In contrast with τ-confluence, these approaches involve some notion of *determinacy*. That is, these approaches need to partition the set of τ-steps and impose some restrictions on the partitions.

The 'Twophase' algorithm described in [NG01], requires that at most one transition from each partition is possible in any state and that the partitions are completely independent. Given a state to expand the algorithm starts by following transitions from the first partition eventually either no more transitions are possible or a cycle is found and the algorithm stops in the minimal state on that cycle. The algorithm then repeats this for the other partitions. Finally all transitions of the resulting state are computed. Our own algorithm can be seen as an extension of this algorithm that replaces the partitioning and uniqueness constraints with the τ-confluence property.

In [GP00], an algorithm that first finds the maximal set of strongly confluent τ-steps in a finite state space and then reduces the state space with respect to that set of confluent τ-steps was presented. This algorithm was also implemented and in some cases it yields remarkable reductions. The major drawback of this approach is that the comparatively huge intermediate state space must be generated.

In this paper, we develop a theory of τ-confluence and we describe an algorithm to compute reduced state spaces. In [Pol01] it is described how confluent subsets of τ-transitions can be found. The same paper describes a transformation based on τ-confluence, which speeds up the computation of representatives and which enables his detection algorithm to find more confluent transitions.

1.2 Overview

In section 3 we formally introduce the different notions of τ-confluence and compare them. In section 4 we define reduced transitions systems for a labeled transition system with confluent τ-steps and show that these are branching bisimilar. in section 5 we consider the union of sets of confluent τ-transitions and a transformation of labeled transition systems, which preserves confluence. In section 6 we explain an algorithm that will compute the reduced transition system "on-the-fly". In section 7 we present a few case studies.

2. PRELIMINARIES

In this section we fix our notation for equivalence classes and labeled transition systems. We also recall the definition of branching bisimulation [vGW96].

For equivalence classes, we use the following notation:

Definition 2.1 Given a set S and an equivalence relation $R \subset S \times S$, we define

$$\forall s \in S : s/_R = \{s' \in S \mid s\, R\, s'\} \text{ and } S/_R = \{s/_R \mid s \in S\} \quad .$$

Our labeled transition systems are labeled with actions from a given set Act. We assume that $\tau \in \mathsf{Act}$.

Definition 2.2 A labeled transition system 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 $(s,a,t) \in \rightarrow$. We also have the following arrow notations:

Definition 2.3

$$\begin{array}{rcl} \xrightarrow{a}\!\!\!\!\!\!\rightarrow & & \text{the transitive reflexive closure of } \xrightarrow{a} \\ \xleftrightarrow{a}\!\!\!\!\!\!\rightarrow & & \text{the transitive symmetric reflexive closure of } \xrightarrow{a} \\ \xrightarrow{a}{}^{\equiv} & & \text{the reflexive closure of } \xrightarrow{a} \\ \Longrightarrow^{a} & = & \begin{cases} \xrightarrow{\tau}{}^{\equiv} & \text{, if } a = \tau \\ \xrightarrow{a} & \text{, otherwise} \end{cases} \end{array}$$

In rewriting theory $\leftrightarrow\!\!\!\!\!\!\rightarrow$ is known as the convertibility relation and is often denoted with the $=$-sign. For use in diagrams $\leftrightarrow\!\!\!\!\!\!\rightarrow$ is much more convenient. Also, using $\leftrightarrow\!\!\!\!\!\!\rightarrow$ avoid overloading $=$ too much.

As notion of equivalence between labeled transition systems, we will use branching bisimulation:

Definition 2.4 Given two transition systems $S_1 \equiv (S_1, \xrightarrow[1]{}, s_0^{(1)})$ and $S_2 \equiv (S_2, \xrightarrow[2]{}, s_0^{(2)})$. A relation $R \subseteq S_1 \times S_2$ is a branching bisimulation on S_1 and S_2 if

$$\forall s_1 \in S_1, s_2 \in S_2 : s_1 \, R \, s_2 \implies \left\{ \begin{array}{l} \forall t_1 \in S_1 : s_1 \xrightarrow[1]{a} t_1 \implies \left\{ \begin{array}{c} a \equiv \tau \wedge t_1 \, R \, s_2 \\ \vee \\ \exists s_2', t_2 \in S_2 : s_2 \xrightarrow{\tau}\!\!\!\!\!\!\rightarrow s_2' \xrightarrow[2]{a} t_2 \wedge s_1 \, R \, s_2' \wedge t_1 \, R \, t_2 \end{array} \right. \\ \qquad\qquad\qquad\qquad \wedge \\ \forall t_2 \in S_2 : s_2 \xrightarrow[2]{a} t_2 \implies \left\{ \begin{array}{c} a \equiv \tau \wedge t_2 \, R \, s_1 \\ \vee \\ \exists s_1', t_1 \in S_1 : s_1 \xrightarrow{\tau}\!\!\!\!\!\!\rightarrow s_1' \xrightarrow[1]{a} t_1 \wedge s_1' \, R \, s_2 \wedge t_1 \, R \, t_2 \end{array} \right. \end{array} \right.$$

If a branching bisimulation R on S_1 and S_2 exists, such that $s_0^{(1)} \, R \, s_0^{(2)}$ then we say that S_1 and S_2 are branching bisimilar, denoted $S_1 \underline{\leftrightarrow}_b S_2$

3. Partial τ-confluence

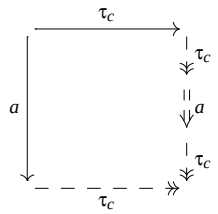
In this section we give the various τ-confluence definitions and compare them.

In [GS96] a notion of weak τ-confluence was defined, which was redefined in [Yin00]. We will refer to these notions as ultra weak τ-confluence and weak τ-confluence. The essential difference between ultra weak τ-confluence and weak τ-confluence is the same difference as the difference between weak confluence and confluence in term rewriting. Given convergence (termination) of the τ-steps the notions are equivalent, but in the presence of divergent (non-terminating) τ-steps they are not. To express this distinction, we use a different definition of weak τ-confluence than that in [Yin00]. We will show that the two definitions are equal. We will also show that, just as for term rewriting, weak τ-confluence implies ultra weak τ-confluence and ultra weak τ-confluence plus termination (convergence) of labeled τ-steps implies weak τ-confluence. In addition, we will define a new notion of τ-confluence, which is essentially stronger than weak confluence. This new notion is necessary, because weak τ-confluence is not sufficient to guarantee the correctness of our reduction algorithm. We will also use the notion of strong τ-confluence, defined in [GS96].

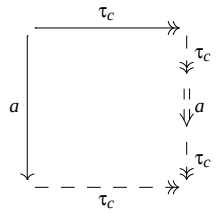
Given a labeled transition system $(S, \rightarrow, s_0)$ and a subset of τ-transitions $c \subset \xrightarrow{\tau}$, we write $s \xrightarrow{\tau_c} t$ for $(s,t) \in c$. For subsets of τ-transitions, we define the following properties:

Definition 3.1 Given a transition system $S \equiv (S, \rightarrow, s_0)$ and a subset of τ-transitions c, we say that c is *ultra*

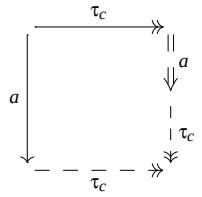
weakly confluent if the following diagram holds:



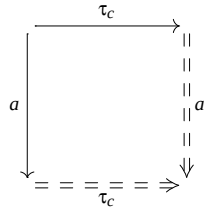
We say that c is *weakly confluent* if the following diagram holds:



We say that c is *confluent* if the following diagram holds:



We say that c is *strongly confluent* if the following diagrams holds:

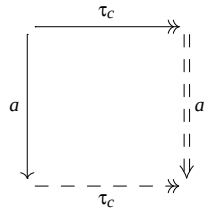


It was already mentioned in [GS96] and [Yin00] that the properties defined in those papers form a hierarchy. With the addition of τ-confluence, we still have a hierarchy. First, we will prove the implications. Afterwards, we will explain the differences and give counterexamples.

Proposition 3.2 Given a transition system $S \equiv (S, \rightarrow, s_0)$ and a subset of τ-transitions c, we have that

$$c \text{ is strongly confluent} \implies c \text{ is confluent} \implies c \text{ is weakly confluent} \implies c \text{ is ultra weakly confluent.}$$

Proof. It is easy to prove that the following diagram holds if c is strongly confluent:



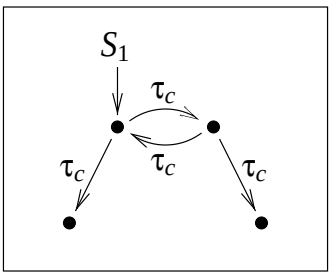


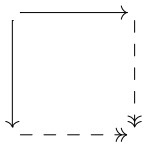
Figure 1: The LTS S_1

This diagram is a special case of the diagram for confluence, which is a special case of the diagram of weak confluence, which is a special case of the diagram of ultra weak confluence. □

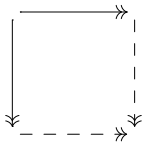
We now give some counterexamples to show that the implications are proper implications.

To compare ultra weak confluence to weak confluence, we need to borrow the notions weak confluence and confluence from term rewriting:

Definition 3.3 Given a set A and relation $\rightarrow \subset A \times A$. The relation $\rightarrow$ is weakly confluent if the following diagram holds:



The relation $\rightarrow$ is confluent if the following diagram holds:



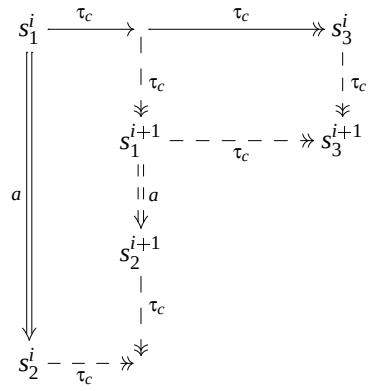
Ultra weak confluence implies that $\xrightarrow{\tau_c}$ is weakly confluent. Weak confluence implies that $\xrightarrow{\tau_c}$ is confluent. It is well-known that weak confluence does not imply confluence. LTS S_1 in Fig. 1 is an example of an LTS, which is ultra weakly confluent, but not weakly confluent. It was constructed from the classical counterexample, that shows that weak confluence does not imply confluence, by tagging one of the states in the transition system as the initial state.

The well-known "Newman's lemma" states that termination plus weak confluence implies confluence. A similar property holds here:

Proposition 3.4 Given a transition system $S \equiv (S, \rightarrow, s_0)$ and a subset of τ-transitions c, we have that if c is ultra weakly confluent and $\xrightarrow{\tau_c}$ is convergent (terminating) then c is weakly confluent.

Proof. Given $s_1, s_2, s_3 \in S$, such that $s_1 \xrightarrow{a} s_2 \wedge s_1 \xrightarrow{\tau_c} s_3$. We can now construct three finite sequences that satisfy the condition $s_1^i \stackrel{a}{\Longrightarrow} s_2^i \wedge s_1^i \xrightarrow{\tau_c} s_3^i$ as follows: Let $s_1^0 = s_1$, $s_2^0 = s_2$ and $s_3^0 = s_3$.

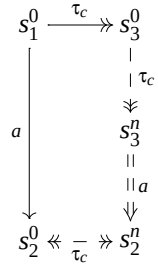
If $s_1^i = s_3^i$ then the sequence ends. Otherwise, we can find s_1^{i+1}, s_2^{i+1} and s_3^{i+1} such that



The left sub-diagram exists due to ultra weak confluence. The right sub-diagram exists due to ultra weak confluence and Newman's lemma. Termination of this construction follows from the fact that given an infinite sequence, we would have an infinite τ_c sequence:

$$s_1^0 \xrightarrow{\tau_c} \xrightarrow{\tau_c} s_1^1 \xrightarrow{\tau_c} \xrightarrow{\tau_c} s_1^2 \xrightarrow{\tau_c} \cdots$$

Such an infinite sequence contradicts the convergence assumption. Hence, the sequence is finite. Let n be the length of the sequences then we have



Weak confluence of c follows easily from this diagram. □

This proposition explains why ultra weak confluence works in a setting with convergent τ-steps and why weak confluence has to be used in the general case.

The key difference between weak confluence and confluence is this: if in a certain state an a-transition is possible then after a confluent τ-transition some a-transition is possible. After a weakly confluent τ-transition the a-transition need not be possible until some other weakly confluent τ-transition have been taken. This explains why the transition system S_2 in Fig. 2 is weakly confluent, but not confluent.

In Fig. 3 we have drawn a transition system S_3, which is confluent, but not strongly confluent.
This completes the counter examples. We continue with a few elementary properties.

Our definition of weak τ-confluence is not identical to the definition in [Yin00]. In terms of diagrams the difference is that we use multiple steps in the top of the diagram and a single step, whereas Ying uses multiple steps on the left and a single step at the top. The following proposition states that our definition is equivalent to that of Ying:

Proposition 3.5 Given a transition system $S \equiv (S, \rightarrow, s_0)$ and a subset of τ-transitions c, we have that c is

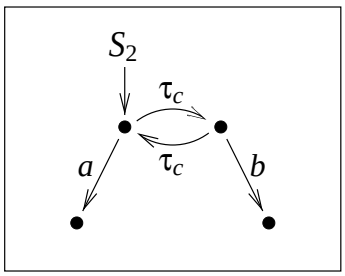


Figure 2: The LTS S_2

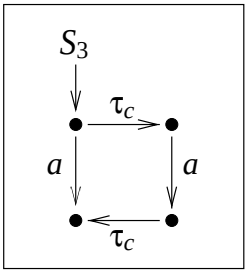
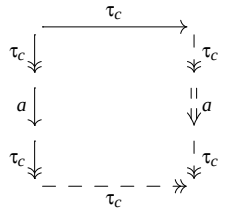


Figure 3: The LTS S_3

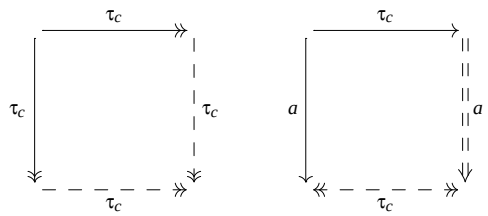
weakly confluent iff the following diagram holds:



Proof. Simple inductive tiling argument. □

A subset of τ-steps is confluent if it forms a confluent abstract reduction system and if in every state it holds that if an action is possible before a confluent τ-step it is still possible after that step and the results of the actions before and after convertible using confluent τ-steps:

Proposition 3.6 Given a transition system $S \equiv (S, \rightarrow, s_0)$ and a subset of τ-transitions c, we have that c is confluent iff the following two diagrams hold:



Proof. Trivial. □

4. State space reduction

In this section, we define several notions of reduced state space and show that under the right conditions these reduced state state spaces are branching bisimilar to their originals.

Definition 4.1 Given a transition system $S \equiv (S, \rightarrow, s_0)$ and an equivalence relation R on S, we define the transition system $S/_R$ as $(S/_R, \xrightarrow[R]{}, s_0/_R)$, where $C_1 \xrightarrow[R]{a} C_2$ if $\exists s_1 \in C_1, s_2 \in C_2 : s_1 \xrightarrow{a} s_2$.

The conversion $\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow$ is an equivalence relation and given that the subset is weakly confluent we have that the transition system modulo $\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow$ is branching bisimilar to the original transition system.

Theorem 4.2 Given a transition system $S \equiv (S, \rightarrow, s_0)$ with a weakly confluent subset of τ-steps labeled c, we have that

$$S \leftrightarrow_b S/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow} \ .$$

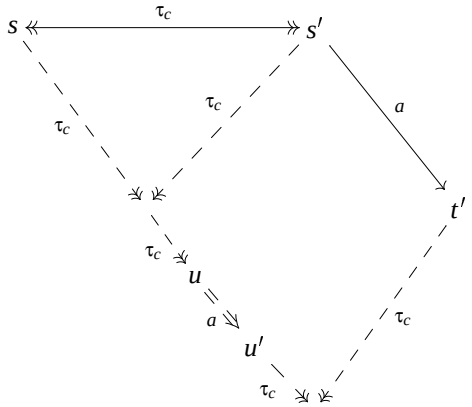
Proof. First, we show that the relation $R = \{(s, s/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}) \mid s \in S\}$ is a branching bisimulation for S and $S/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}$.

Given $s \, R \, s/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}$, we have to consider the following cases:

[$s \xrightarrow{\tau_c} s'$] We have that $s' \, R \, s'/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow} \equiv s/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}$.

[$s \xrightarrow{a} s'$ for $a \neq \tau_c$] We have that $s/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow} \xrightarrow{a} s'/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}$ and that $s' \, R \, s'/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}$.

[$s/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow} \xrightarrow{a} C$] For some $s' \in s/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}$ we have that $s' \xrightarrow{a} t' \in C$. By definition we have that $s \twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow s'$. Because of weak confluence we have the following diagram:



We must now distinguish two cases for $u \stackrel{a}{\Rightarrow} u'$:

$a \equiv \tau$ and $u \equiv u'$ Because we have that $s \twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow t'$, we also have that $t' \, R \, t'/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow} \equiv s/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}$.

$u \xrightarrow{a} u'$ We have that $s \twoheadrightarrow\!\!\!\!\!\xrightarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow u \xrightarrow{a} u'$, $u \, R \, s/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}$ and $u' \, R \, t'/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}$.

Thus, we have that R is a branching bisimulation. Because $s_0 \, R \, s_0/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}$ this implies that

$$S \leftrightarrow_b S/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow} \ .$$

□

Our next method for state space reduction is reduction modulo a representation map. A representation map chooses a representative in every $\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow$ equivalence class. The idea is that the transitions of the representative are precisely the transitions of the whole class in the transition system modulo $\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow$. To make this idea work the representation map cannot yield any element. Given an equivalence class $s/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}$ is must yield an element of the terminal strongly connected component of $(s/_{\twoheadleftarrow\!\!\!\!\!\xleftarrow{\tau_c}\!\!\!\!\!\twoheadrightarrow}, \xrightarrow{\tau_c})$.

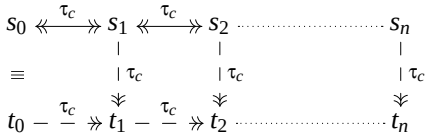


Figure 4: Construction in the proof of Prop. 4.4

Definition 4.3 Given a transition system $S \equiv (S, \rightarrow, s_0)$ and a subset of τ-steps c, a map $\phi : (S/_{\ll \tau_c \gg}) \rightarrow S$ is called a representation map if

$$\forall s \in S : s \xrightarrow{\tau_c} \!\!\!\!\rightarrow \phi(s/_{\ll \tau_c \gg}) \quad .$$

We define the transition system modulo ϕ as $S/_\phi = (S/_{\ll \tau_c \gg}, \xrightarrow[\phi]{}, s_0/_{\ll \tau_c \gg})$, where $C_1 \xrightarrow[\phi]{a} C_2$ if $a \neq \tau_c$ and $\exists s_2 \in C_2 : \phi(C_1) \xrightarrow{a} s_2$.

To make this reduction possible, a representation map must exist. If the LTS is infinite then an infinite sequence of confluent τ steps may prohibit the existence of a representation map. However, for the interesting cases of finite transition systems the existence is guaranteed:

Proposition 4.4 Given a transition system $S \equiv (S, \rightarrow, s_0)$ and a subset of τ-steps c. If $\xrightarrow{\tau_c}$ is confluent and S is finite then a representation map exists.

Proof. Define

$$TSCC(s) = \cap_{s' \in s/_{\ll \tau_c \gg}} \{s'' \in S \mid s' \xrightarrow{\tau_c} \!\!\!\!\rightarrow s''\} \quad .$$

Any map $\phi : (S/_{\ll \tau_c \gg}) \rightarrow S$, such that for all $s \in S$ $\phi(s/_{\ll \tau_c \gg}) \in TSCC(s)$, is a representation map. If for all $s \in S$ $TSCC(s)$ is not empty then such a map exists.

Given $s \in S$, we have that $s/_{\ll \tau_c \gg}$ is finite, so for some n we may write $s/_{\ll \tau_c \gg} = \{s_0, s_1, \cdots, s_n\}$. Define $t_0 = s_0$. Because $\xrightarrow{\tau_c}$ is confluent there exist t_i such that $s_i \xrightarrow{\tau_c} \!\!\!\!\rightarrow t_i$ and $t_{i-1} \xrightarrow{\tau_c} \!\!\!\!\rightarrow t_i$. We know that $t_n \in TSCC(s)$, so $TSCC(s)$ is non-empty. (See Fig. 4.) □

Although this reduction is very similar to reduction modulo confluent τ-steps there are differences. Most importantly, we need confluence for the state space modulo a representation map to be branching bisimilar to the original state space. For example, in Figure 5 we have drawn the results of taking state space S_2 from Figure 2 modulo confluent τ-steps and modulo each of the two possible representation maps. The state space modulo confluent τ-steps is branching bisimilar to S_2, but the other two are not.

Even if the state space modulo a representation map is branching bisimilar to the original state space it need not be identical to the state space modulo confluent τ-steps. For example, in Figure 6 we have drawn a transition system and two of its reductions. In the transition system we have indicated the representatives as open circle and other states as filled circles. We have indicated the non-trivial equivalence class with a dashed loop. The first reduction is modulo the representation map. The second reduction is modulo confluence. Note how this reduction introduces a new loop at the top state. The two reductions differ only by that loop.

Also, the result depends on the choice of representation map. For example, in Figure 7 we have drawn the same graph with different representation maps.

Theorem 4.5 Given a transition system $S \equiv (S, \rightarrow, s_0)$ with a confluent subset of τ-steps labeled c and a representation map ϕ, we have that

$$S \underline{\leftrightarrow}_b S/_\phi \quad .$$

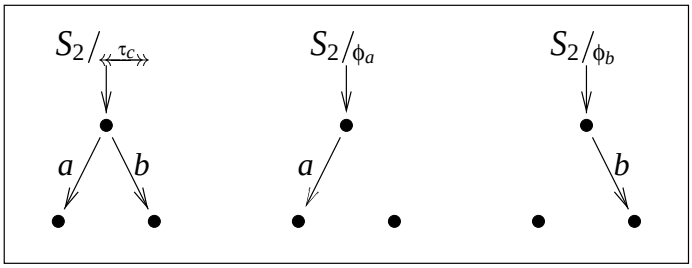


Figure 5: Three reductions of S_2.

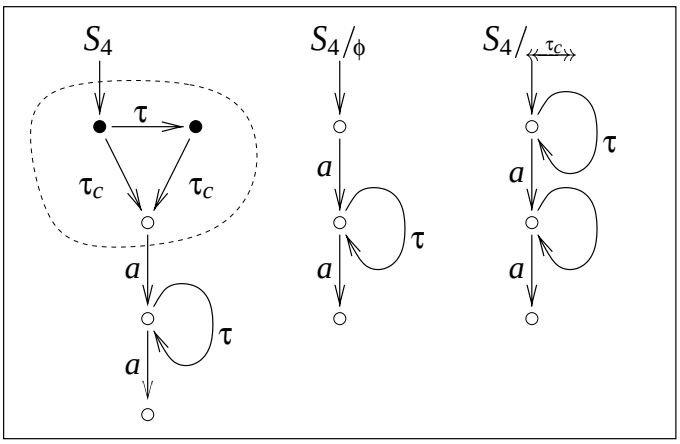


Figure 6: A comparison of transition systems modulo.

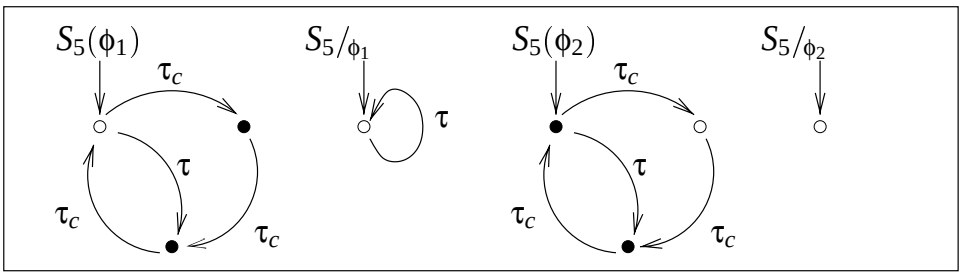
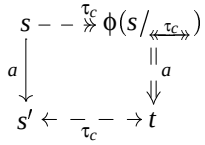


Figure 7: Differences due to choice of representatives.

Proof. We will show that the relation $R = \{(x, x/_{\ll \tau_c \gg}) \mid x \in S\}$, is a branching bisimulation for S and $S/_{\phi}$.
By definition $s_0 \, R \, s_0/_{\ll \tau_c \gg}$. Given $s \, R \, s/_{\ll \tau_c \gg}$, we have to consider the following cases:

[$s \xrightarrow{\tau_c} s'$] By definition we have that $s' \, R \, s'/_{\ll \tau_c \gg} \equiv s/_{\ll \tau_c \gg}$.

[$s \xrightarrow{a} s'$ for $a \neq \tau_c$] We have the following diagram:



We can now distinguish two cases:

- The step from $\phi(s/_{\ll \tau_c \gg})$ to t is empty. In this case we have that $s' \, R \, s'/_{\ll \tau_c \gg} \equiv s/_{\ll \tau_c \gg}$.
- The step from $\phi(s/_{\ll \tau_c \gg})$ to t is non-empty. In this cases we have that $s/_{\ll \tau_c \gg} \xrightarrow{a} t/_{\ll \tau_c \gg} \equiv s'/_{\ll \tau_c \gg}$ and $s' \, R \, s'/_{\ll \tau_c \gg}$.

[$s/_{\ll \tau_c \gg} \xrightarrow{a} C$] We have that $a \xrightarrow{\tau_c} \phi(s/_{\ll \tau_c \gg})$, $\phi(s/_{\ll \tau_c \gg}) \, R \, s/_{\ll \tau_c \gg}$, $\phi(s/_{\ll \tau_c \gg}) \xrightarrow{a} t$ and $t \, R \, C$,

Thus, we have that R is a branching bisimulation. □

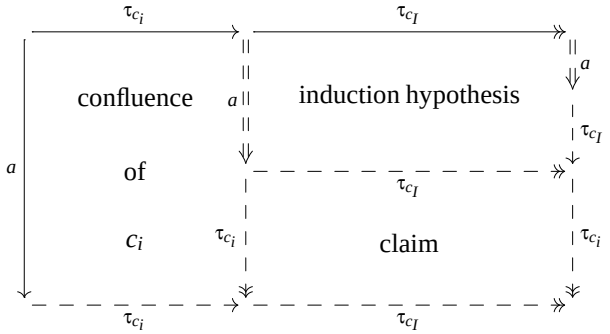
5. Properties of sets of confluent τ-transitions

In this section, we prove some theorems that are useful for proving the correctness of the `confelm` and `confcheck` tools for the μCRL toolset.

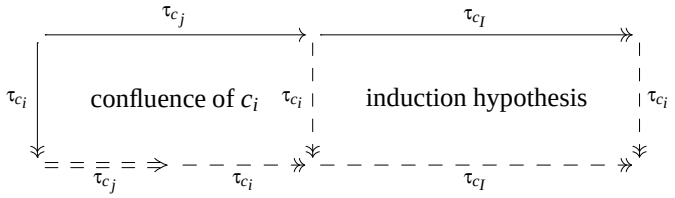
The `confcheck` tool described in [Pol01] is used to label confluent summands. It does so by checking each τ-summand separately. In other words, given a partitioned set of τ-transitions the tool checks which partitions are confluent and returns the union of the confluent partitions as the set of confluent transitions. To prove the correctness of this tool it is useful to know that the union of confluent sets of τ-steps is a confluent set of τ-steps:

Theorem 5.1 Given a transition system $S \equiv (S, \rightarrow, s_0)$ and subsets of τ-transitions c_i, $i \in I$. If for each $i \in I$ the set c_i is confluent then $\cup_{i \in I} c_i$ is confluent.

Proof. Let $c_I = \cup_{i \in I} c_i$. By induction on the number of steps at the top of the confluence diagram we can prove that c_I is confluent:



The claim is also proven using induction on the number of steps at the top of the diagram:



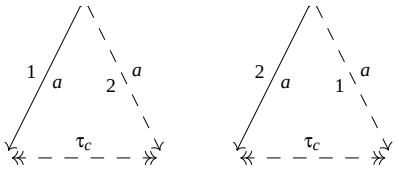
□

The `confelm` tool is a prototype, which symbolically appends some confluent τ-steps to other steps. That is, a system with a trace $s_0 \xrightarrow{a} s_1 \xrightarrow{\tau_c} s_2$ might be transformed into a system where we have the trace $s_0 \xrightarrow{a} s_2$. This transformation is quite useful. First, there are cases where `confelm` has enabled `confcheck` to find more confluent transitions. Second, there are cases where the τ-confluence reduction could be computed symbolically with `confelm`.

The transformation performed by `confelm` is a special case of a transformation where the destination of every arrow, which is not a confluent τ-transition, can be changed to a τ_c-convertible state. The following theorem states that the latter transformation preserves τ-confluence and branching bisimilarity.

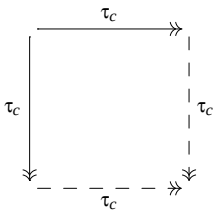
Theorem 5.2 Given transition systems $S_1 \equiv (S, \xrightarrow[1]{}, s_0)$ and $S_2 \equiv (S, \xrightarrow[2]{}, s_0)$ and a confluent subset of τ-transitions c of $\xrightarrow[1]{}$.

If c is also a subset of τ-transitions of $\xrightarrow[2]{}$ and the following diagrams hold



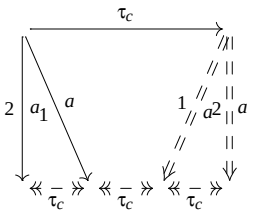
then c is a confluent subset of τ-transitions of $\xrightarrow[2]{}$ and $S_1 \underline{\leftrightarrow}_b S_2$.

Proof. First we prove that c is a confluent subset of τ-transitions of $\xrightarrow[2]{}$. By Prop. 3.6, we have that



We can omit the subscripts 1 and 2 because as far as c steps are concerned there is no difference.

We also have



Hence, by Prop. 3.6, we have that c is confluent for S_2.

From Thm. 4.2, we can conclude that $S_1 \underline{\leftrightarrow}_b S_1/_{\ll \tau_c \gg}$ and $S_2 \underline{\leftrightarrow}_b S_2/_{\ll \tau_c \gg}$. From the given diagrams it follows that $S_1/_{\ll \tau_c \gg} \equiv S_2/_{\ll \tau_c \gg}$. Because being branching bisimilar is a transitive property, the conclusion is that $S_1 \underline{\leftrightarrow}_b S_2$. □

The presentation of the theory ends here. We now continue with the application of the theory.

6. Application to state space generation

In this section we explain an algorithm for state space generation, which is capable of computing a representation map and the state space modulo that representation map "on-the-fly".

init	A constant which denotes the initial state.
transitions	A function which takes a state and returns the possible transitions from the given state in the form of a set of pairs of labels and states.

Table 1: Standard transition system API

outInit	A procedure that writes the initial state of the generated state space.
outTrans	A procedure that takes a transition in the form of a from state, a label and a to state, and outputs this transition.

Table 2: Output API

A simple method for implementing state space generation is to implement an API that allows on-the-fly generation of a transition system and then using that API for a simple traversal of all reachable states. (See Tables 1, 2 and 3.)

Often the generated transition system is reduced with respect to some notion of bisimulation immediately after its generation. Our new algorithm exploits τ-confluence to reduce the transition system on-the-fly. More precisely, our algorithm computes a representation map on-the-fly and generates the state space modulo that map rather than the full state space.

The input API and the top-level traversal of the generation algorithm are quite similar to the simple implementation. (See Tables 4 and 5.) The only difference is that for both the initial state and the target states of transitions we compute representatives. In Table 6, we give a detailed description of the algorithm that computes representatives. This description uses mutable partial functions (hash tables) and simple lists with constructors cons and nil and function head and tail.

The algorithm is derived from Tarjan's algorithm for finding strongly connected components (see [Tar72]). Like Tarjan's algorithm we perform a depth first traversal of the confluent transition graph. However, we terminate as soon as either a state with a known representative is found or the terminal strongly connected component (TSCC) is found. In the former case we return the known representative, in the latter case we choose the entry point to the terminal strongly connected component as the representative. The representative of all states we visited is set to the one we found.

To determine which states belong to the TSCC, we number each state as we enter it to explore it. We maintain a partial function 'low', which is initially set to the number and which on backtracking from a state will contain the least possible number for a state, reachable by following some transitions in the depth first search tree and at most one other transition. The first time we try to backtrack from a state whose number is equal to its 'low' value, we know that we try to backtrack from the entry state to the TSCC. Backtracking information is kept in

```
generate()
    visited := {init}
    explored := ∅
    outInit(init)
    while visited ≠ explored
        s := choose(visited \ explored)
        explored := explored∪{s}
        for (a,s') ∈ transitions(s)
            outTrans(s,a,s')
            visited := visited∪{s'}
```

Table 3: Standard state space generation algorithm

init	A constant which denotes the initial state.
conftau	A function which takes a state and returns the set of states that are reachable form that state in one confluent τ-step.
transitions	A function which takes a state and returns the possible transitions from the given state in the form of a set of pairs of labels and states, excluding the confluent τ-transitions.

Table 4: τ-confluence aware transition system API

```
generate()
    repr:=undefined
    visited := {representative(init)}
    explored := ∅
    outInit(representative(init))
    while visited ≠ explored
        s := choose(visited \ explored)
        explored := explored∪{s}
        for (a,s') ∈ transitions(s)
            s' := representative(s')
            outTrans(s,a,s')
            visited := visited∪{s'}
```

Table 5: τ-confluence aware state space generation algorithm

a partial function ‘back’. The stack of visited states that Tarjan’s algorithm maintains, is not necessary here, because the only information we need is membership of this stack and until the first SCC is found membership of the stack is equal to having been explored.

Note that the confluent transitions are being generated ”on-the-fly”. Also note that ‘repr’ will contain the representation map once the state space generation is finished.

7. Case studies

In this section we describe some case studies performed by Jaco van de Pol. These case studies are performed in the setting of the μCRL toolset. They use an implementation of the state space generation algorithm described in the previous section and the confluence labeling algorithm described in [Pol01].

In table 7, we present the results for a few instances of the firewire tree identify protocol and a program written in the Splice architecture. The table lists the sizes of the original and reduced state spaces. The latter are listed under ’generated’. The columns under ’visited’ are listed to give a fair comparison of the amount of work. The time needed to compute the original state space is linear in the size, the time needed to compute the reduced state space is linear in the ’visited’ column.

The results for the tree identify protocol shows that by using confluence, we gain an order of magnitude. The results for the Splice problem are even better.

The splice problem is described extensively in [Pol01], so we will just give a short description here.

Splice is a distributed shared data space architecture. A Splice network consists of a number of application processes, that coordinate through agents. The agents are coupled via a network. Fig. 8 depicts a simple Splice network. The agents maintain a set of items (the shared data space), which applications can write items into, or read items from. The agents distribute their items by asynchronously sending messages to each other over the network.

Due to the asynchronous communication mechanism, there is much non-determinism: messages from one agent are sent to other agents in any order. Consequently, different agents can receive two messages in different

```
representative(v)
    if repr(v) defined then return repr(v)
    visited := ∅
    number(v) := 0
    count:=0
    loop
        if number(v)=0 then
            count++
            number(v):=count
            low(v):=count
            next(v):=nil
            visited := visited ∪ conftau(v)
            for s ∈ conftau(v)
                if repr(s) defined then
                    v := repr(s)
                    exit loop
                next(v):=cons(s,next(v))
                number(s):=0
        if next(v)=nil then
            if number(v)=low(v) then exit loop
            low(back(v)):=min(low(back(v)),low(v))
            v:= back(v)
        else
            u:=head(next(v))
            next(v):=tail(next(v))
            if number(u)=0 then
                back(u):=v
                v:=u
            else
                if number(u) < number(v) then low(v):=min(low(v),number(u))
    end loop
    for s ∈ visited
        repr(s):= v
    return v
```

Table 6: Finding the representative

	original state space		reduced state space			
			visited		generated	
system	nodes	transitions	nodes	transitions	nodes	transitions
Splice(2,1)	85362	613482	105	106	9	10
Splice(2,2)	$> 15.10^6$	$>> 15.10^7$	157	158	9	10
Splice(4,2)	??	??	617	628	25	36
Splice(6,2)	??	??	1573	1606	56	89
Firewire(10)	72020	389460	8443	24940	6171	22668
Firewire(12)	446648	2853960	40919	137588	27219	123888
Firewire(14)	2416632	17605592	167609	606970	105122	544483

Table 7: Results

orders, even if they originate from the same agent. After proving confluence, the generation algorithm detects that all these different orders are equivalent, and reduces the state space as if there were only one global set.

Another possible reduction occurs when the workers read any message from their agent, and write some computed result back. Such transactions cannot be represented by super-deterministic transitions, because a worker can start with any message in the current set of its agent. Nevertheless, several such transactions commute, basically because $(A \cup \{a\}) \cup \{b\} = (A \cup \{b\}) \cup \{a\}$. Using confluence reduction, only a fixed transaction order is explored.

8. Conclusion

We have introduced a new notion of τ-confluence, which allows a reduced state space to be computed efficiently from a symbolic representation of a state space. The expirimental results show that the gain can be very useful.

As a result of adding τ-confluence based tools to the μCRL toolset, we are now capable of dealing with bigger problems. The next step will be the development of a distributed version of the state space generator, which is capable of computing the redution.

9. Acknowledgments

The author is grateful to Wan Fokkink, Jan Friso Groote, Izak van Langevelde, and Jaco van de Pol for comments on drafts of this paper.

References

[BP97] M.A. Bezem and A. Ponse. Two finite specifications of a queue. *Theoretical Computer Science*, 177(2):487–507, 1997.

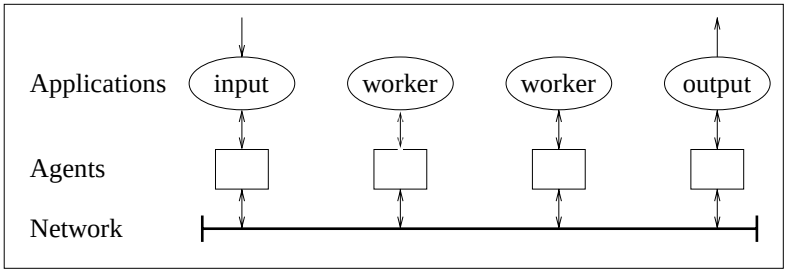


Figure 8: Splice architecture

[GP00] J.F. Groote and J.C. van de Pol. State space reduction using partial τ-confluence. In M. Nielsen and B. Rovan, editors, *Proc. of MFCS 2000*, LNCS 1893, pages 383–393. Springer, 2000.

[GS96] J.F. Groote and M.P.A. Sellink. Confluence for process verification. *Theoretical Computer Science*, 170:47–81, 1996.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[NG01] R. Nalumasu and G. Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, (to appear), 2001.

[Pel97] D.A. Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In D.A. Peled, V.R. Pratt, and G.J. Holzmann, editors, *Partial Order Methods in Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 233–258. American Mathematical Society, July 1997.

[Pol01] J.C. van de Pol. A prover for the μCRL toolset with applications – Version 0.1. Technical Report SEN-R0106, CWI, Amsterdam, 2001.

[Tar72] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[vGW96] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[Yin00] Mingsheng Ying. Weak confluence and τ-inertness. *Theoretical Computer Science*, 238:465–475, 2000.