

State Space Reduction by Proving Confluence

Stefan Blom and Jaco van de Pol

CWI, P.O.-box 94.079, 1090 GB Amsterdam, The Netherlands
Email: {sccb1om, vdpo1}@cwi.nl

Abstract We present a modular method for on-the-fly state space reduction. The theoretical foundation of the method is a new confluence notion for labeled transitions systems. The method works by adding confluence information to the symbolic representation of the state space. We present algorithms for on-the-fly exploration of the reduced state space, for generating confluence information and for a symbolic reduction, called prioritization. The latter two algorithms rely on an automated theorem prover to derive the necessary information. We also present some case studies in which tools that implement these algorithms were used.

Keywords: Labeled transition systems, state space reduction, confluence, on-the-fly reduction, theorem proving, symbolic transformation, branching bisimulation, μ CRL tool set

Category: A – regular paper
Title: State Space Reduction by Proving Confluence
Authors: Stefan Blom, CWI, Amsterdam
Jaco van de Pol, CWI, Amsterdam
Contact Author: Jaco van de Pol
Address: CWI, Department of Software Engineering
P.O.-Box 94.079
1090 GB Amsterdam
The Netherlands
Email: Jaco.van.de.Pol@cwi.nl
Phone: +31 - (0)20 592 4137
Fax : +31 - (0)20 592 4199

State Space Reduction by Proving Confluence

Stefan Blom and Jaco van de Pol

CWI, P.O.-box 94.079, 1090 GB Amsterdam, The Netherlands
Email: {sccb1om, vdpo1}@cwi.nl

Abstract We present a modular method for on-the-fly state space reduction. The theoretical foundation of the method is a new confluence notion for labeled transitions systems. The method works by adding confluence information to the symbolic representation of the state space. We present algorithms for on-the-fly exploration of the reduced state space, for generating confluence information and for a symbolic reduction, called prioritization. The latter two algorithms rely on an automated theorem prover to derive the necessary information. We also present some case studies in which tools that implement these algorithms were used.

1 Introduction

A popular approach to the verification of distributed systems is based on an exhaustive state space exploration. This approach suffers from the well-known *state space explosion* problem. Much research is devoted to algorithms that generate a reduced, but essentially equivalent, state space. Collectively, these methods are called partial-order reduction methods. In this paper we introduce a new method for generating a reduced state space that is branching bisimilar with the original one.

The method is based on a subtle variation on the *confluence*-notion for labeled transition systems (LTS). Invisible (τ) steps in the LTS may be confluent or not. All states in a subset connected by confluent steps are branching bisimilar. By virtue of our new confluence notion, this subset may be replaced by a particular *representative* state, and only transitions outgoing from this representative need to be explored. This is explained in Section 2.

In order to apply confluence for the immediate generation of the reduced state space, the confluent transitions must be detected before generating the LTS. This is solved in Section 3 by representing the system specification in an intermediate format, called *linear process*. A specification in this format consists of a finite number of symbolic transitions. The confluence property of each symbolic transition (or in fact a stronger approximation) can be expressed as a Boolean formula over the data types that occur in the specification. This formula is solved by a separate automated theorem prover. If the formula can be proved, the transition is marked as confluent, allowing for some reduction of the corresponding state space.

In some cases it is even possible to feed the information on confluent symbolic transitions back to the symbolic level. This results in a transformation on linear

processes, which we call *symbolic prioritization* (Section 4). In Section 5 we show a number of applications, to which we applied our techniques to reduce the state space. Some of these examples go beyond partial-order reduction methods based on super-determinism. Furthermore, confluence detection and symbolic prioritization can be applied to infinite state spaces as well.

Our ideas are implemented in the context of the μCRL tool set [2]. The basic functionality of this tool set is to generate a state space (LTS) out of a μCRL specification. To this end *linear processes* are used as an intermediate representation. This contributes to the modularity of the tool set. In particular, several optimizations are implemented as separate tools that transform a linear process, aiming at a reduction of the state space to be generated.

To this tool set, we added symbolic prioritization as yet another optimizer on linear processes. Moreover, the on-the-fly reduction algorithm has been integrated in the state space generator of the tool set.

With the approach in this paper we further contribute to modularity. In particular, we defined a notion of confluence, which is quite liberal, but nevertheless sufficient to ensure correctness of the on-the-fly reduction algorithm. Finding confluent transitions is a separate task. In fact, while the maximal set of confluent transitions is hard to detect, it is perfectly acceptable if actual confluence detection algorithms only find a subset.

We propose to use an automated theorem prover to find a reasonable subset of confluent transitions, but an alternative approach could be to prove confluence by hand, or with interactive theorem provers. In cases where the specification is automatically generated from source code, it is sometimes even possible to know a priori that certain transitions are confluent.

1.1 Related Work

Several *partial order reduction* algorithms that preserve branching bisimilarity have been proposed in the literature [18,14,13]. These approaches also allow the reduction to a representative subset of all states. Some of these approaches restrict attention to deterministic transitions. All these approaches involve some notion of *determinacy*.

For instance, criterion $\ddot{A}8$ [18] allows the selection of one outgoing transition from a state, provided it is an invisible *super-deterministic* transition. In our approach, such a transition need not be deterministic, but only confluent, which means that *eventually* the computation paths reach the same state. It can be proved that the set of super-deterministic transitions forms a confluent set of τ -transitions, but not vice versa.

Another criterion for obtaining a branching bisimilar state space is $\ddot{A}5$, which prevents that a transition is postponed forever. This is implemented in [13] by the algorithm Twophase. In phase one, a state is expanded by repeatedly applying *deterministic* invisible transitions, which is then fully expanded in phase two. Our algorithm can be seen as a modification: we take confluent transitions

until a terminal strongly connected component (SCC) is reached, instead of deterministic transitions only.

Confluence is a property of sets of invisible transitions in an LTS. Several confluence notions have been studied in the setting of branching bisimulation [11,19,8]. In [1] these notions are compared systematically. In summary, the notions in [11,19] only deal with *global confluence*, whereas we deal with *partial confluence*, i.e. it suffices to identify a *subset* of the τ -transitions which is confluent. This is essential for practical applications, in which it is seldom the case that all τ -transitions are confluent.

The confluence notion in [11] was adapted to partial confluence in [8]. In order to apply it for state space reduction, this notion required the absence of τ -loops. [8] introduced an algorithm to reduce concrete state spaces as follows. First, all strongly connected τ -components are collapsed, in order to get rid of τ -cycles. Then, the *maximal* set of strongly confluent τ -steps is computed, and strongly confluent τ -transitions are given priority over other transitions. We note that these steps can only be applied *after* generating the unreduced state space. Especially absence of τ -loops is a severe restriction for on-the-fly generation.

In our paper, we use theorem proving to find (not necessarily all) confluent transitions in the specification, which allows the immediate generation of the reduced state space. So we provide a really on-the-fly reduction method. Our new method even allows to perform some optimization at specification level already.

Finally, also [19] introduces a notion of confluence. That notion has the nice theoretical property that the τ -transition relation is confluent, if and only if it only connects states that are branching bisimilar. For reduction purposes that definition is not convenient, because it would require to consider all outgoing transitions in each terminal SCC. Our notion of confluence is slightly stronger, and as a consequence we only have to take all outgoing transitions from one member of each terminal SCC.

2 Confluence and Reduction On-the-fly

In this section we present the confluence property, a state space reduction method based on the confluence property and an algorithm that computes these reduced state spaces “on-the-fly”.

We use labeled transition systems (LTS) to model states spaces. Confluence is a property of sets of invisible transitions in an LTS. Any set of transitions induces an equivalence relation on the set of states, which identifies states in the same component. If a set of invisible transitions is confluent then the induced equivalence relation is a branching bisimulation. Moreover, each finite equivalence class has a representative, whose transitions are the same as those of the whole equivalence class. Because of these representatives, we can give an algorithm that computes the reduced state space without computing the whole original state space.

Because the reduction preserves branching bisimilarity, the reduced state space can be used to check all properties of the original system that are expressible in action based CTL*-X (computation tree logic without next-time) or HML (Hennessy-Milner logic).

2.1 Confluence

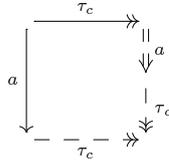
The labels of our LTSs will be taken from a given set Act . We assume that Act contains a special element τ , representing an invisible action.

Definition 1 (LTS). A labeled transition system is a triple (S, \rightarrow, s_0) , consisting of a set of states S , transitions $\rightarrow \subseteq S \times \text{Act} \times S$ and an initial state $s_0 \in S$.

We write $s \xrightarrow{a} t$ for $(s, a, t) \in \rightarrow$. Moreover, \xrightarrow{a} denotes the transitive reflexive closure of \xrightarrow{a} , and we write $s \xRightarrow{a} t$ if either $s \xrightarrow{a} t$, or $a = \tau$ and $s = t$. Given a subset of τ -transitions $c \subseteq \xrightarrow{\tau}$, we write $s \xrightarrow{c} t$ for $(s, t) \in c$. Finally, we write $\mathcal{S}_1 \xleftrightarrow{b} \mathcal{S}_2$ to denote that LTSs \mathcal{S}_1 and \mathcal{S}_2 are branching bisimilar [5].

The idea is that a subset c of the set of invisible transitions is confluent if the steps in c cannot make real choices. This is formalized with two conditions. First, if in a certain state two different sequences of c steps are possible then these sequences can be extended with more c steps to sequences that end in the same state. Second, if in a state both a step from c is possible and an a step is possible then after doing the step from c another a step is possible and the results of the two a steps are in the same c equivalence class. These two requirements can be neatly expressed in a single diagram, as is done in the following definition:

Definition 2 (confluence). Let c be a subset of $\xrightarrow{\tau}$. Then c is confluent iff the following diagram holds:



2.2 Reduction

As mentioned before, the equivalence relation induced by a confluent set is a branching bisimulation. By taking the original state space modulo this equivalence one can reduce the state space. An effective way of computing the transitions of the reduced state space is to find a representative of each equivalence class, whose transitions are precisely the transitions of the equivalence class. These representatives are guaranteed to exist if the equivalence class (seen as a directed graph) has a Terminal Strongly Connected Component (TSCC). In that case every element of the TSCC can be chosen as a representative. Because of confluence there is at most one TSCC. The notion of *representation map* is based on this idea. The first condition forces every element in an equivalence class to have the same representative. The second condition forces this representative to be in the TSCC.

Definition 3 (representation map). Given an LTS $\mathcal{S} \equiv (S, \rightarrow, s_0)$ with a confluent subset of τ -steps labeled c , a map $\phi : S \rightarrow S$ is called a representation map if $\forall s, t \in S : s \xrightarrow{\tau_c} t \Rightarrow \phi(s) = \phi(t)$ and $\forall s \in S : s \xrightarrow{\tau_c} \phi(s)$.

A representation map exists if and only if every equivalence class has a TSCC. Because we usually work with finite LTSs the existence of TSCCs and hence representation maps is guaranteed. Based on the notion of representation map we can define a reduced LTS. The set of states of the reduced LTS is the set of representatives. For every transition from a representative to a destination state in the original LTS, we include a transition from that representative to the representative of that destination in the reduced LTS. Finally, the new initial state is the representative of the old initial state. In [1] it is proven that the reduced LTS is branching bisimilar to the original LTS.

Definition 4 (LTS modulo ϕ). Given a representation map ϕ , we define $\mathcal{S}/\phi = (\phi(S), \xrightarrow{\phi}, \phi(s_0))$, where $s \xrightarrow{\phi} t$ if $a \neq \tau_c$ and $\exists t' : s \xrightarrow{a} t'$ and $\phi(t') = t$. As usual, $\phi(S) = \{\phi(s) \mid s \in S\}$.

Theorem 5. Given a transition system \mathcal{S} with a confluent subset of τ -steps labeled c and a representation map ϕ , we have that $\mathcal{S} \xleftrightarrow{b} \mathcal{S}/\phi$.

2.3 Algorithm for Reduction On-the-fly

The essential functions in an “on-the-fly” interface for an LTS are a function that yields the initial state and a function that computes outgoing transitions of a given state. Given an on-the-fly representation of an LTS and the label for confluent transitions, the key to providing an on-the-fly representation of the reduced LTS is a function that computes a representation map. (See Figure 1.)

Such a function must return a representative from the terminal strongly connected component of the τ_c graph. Moreover, this representative must be unique for all elements of an equivalence class. We implemented the latter requirement simply by maintaining a table of the representatives that we have computed so far. To compute a representative if it is not in the table, we use a variation on Tarjan’s algorithm for finding strongly connected components [17]. More precisely, we perform a depth first search of the graph of confluent transitions until we find a state with a known representative or until we backtrack from a node where we entered a strongly connected component. The first thus encountered component is the TSCC. In the latter case this node is chosen as the representative, and stored in the table.

The table consumes a significant amount of memory. If this is unacceptable and a total order on the set of states is available then one can choose the least state in the TSCC as the representative and recompute the TSCC each time rather than storing it.

3 Confluence Detection by Theorem Proving

In the previous section we discussed a state space generation algorithm, which applies reduction on-the-fly, based on some information on confluent transitions.

<pre> ReducedInit() return Representative(Init()) ReducedNext(state) return { (a, Representative(s)) a ≠ τ_c, (a,s) ∈ Next(state) } </pre>

Figure 1.

In this section we show how this information is obtained. In particular, by exploiting a special format for the specification, we show that (stronger approximations of) confluence properties can be expressed by quantifier-free first order formulae. These formulae can be solved by a separate automated theorem prover.

3.1 Symbolic specifications in μCRL

We implemented our ideas in the setting of the μCRL formalism. A μCRL specification [10,2] consists of an *algebraic specification*, defining the data sorts, and a *process specification*, defining the system under scrutiny as the parallel composition of several components, each specified as a process algebra expression. We utilize the result in [9], that such specifications can be transformed to a *linear process*, without a considerable blow-up in size.

A linear process consists of a vector of global state variables (each with a particular sort from the algebraic data specification), an initial state vector (terms whose sorts match the state variables), and a set of program rules, traditionally called summands. These summands define the process behaviour in a condition/action/effect style, similar to I/O-automata or UNITY programs. These summands have the following form¹:

$$\left\{ \sum_{e_i} [b_i(d, e_i)] \Rightarrow a_i(d, e_i); d := g_i(d, e_i) \right\}_{i \in I}$$

We assume that the data algebra contains the special sorts Bool for booleans and Act for external actions. In the summands above, we have that :

- e_i is a vector of local variables, used for instance to model arbitrary input of this summand.
- $b_i(d, e_i)$ is a term of type Bool, with variables among d and e_i , denoting the condition or guard of the summand.
- $a_i(d, e_i)$ is a term of type Act, with variables among d and e_i , denoting the action executed by this summand.
- $g_i(d, e_i)$ is a vector of terms, whose sorts match the vector of global state variables.

¹ We focus on the essential ingredients, rather than concrete μCRL syntax.

Each summand specifies a structural transition \xrightarrow{i} as follows:

$$d \xrightarrow{i} d' \text{ iff } \exists e_i. b_i(d, e_i) \wedge d' = g_i(d, e_i) \wedge \alpha = a_i(d, e_i)$$

Together, a linear process specifies a *structural* labeled transition system $(\Sigma, (\xrightarrow{i})_{i \in I}, s_0)$, from which the ordinary LTS can be obtained by taking the union of all structural transitions. Here a state in Σ is a vector of data values for the global variables; s_0 is the initial state vector; I is the (index) set of the summands; \xrightarrow{i} is the structural transition generated by summand i , which is a subset of the transitions of the whole LTS.

Note that a structural transition defined in this way is partial (due to the enabling condition b) and non-deterministic (due to choice involved in $\exists e$). Also note that one summand may generate transitions with various labels. A similar decoupling of action labels and structural transitions occurs in [18].

3.2 Generation of Confluence Formulae

Owing to the format of linear processes, commutation formulae can be generated. In order to facilitate automated theorem proving, we try to avoid quantifiers. The generated formulae will be Boolean expressions over the user defined abstract data types, with implicit universal quantification.

To get the formulae in this form, we only consider a special case, which occurs frequently in practice. So in fact we detect a stronger approximation of confluence. Consider two divergent steps of summands i and j :

$$\sum_{e_i} [b_i(d, e_i)] \Rightarrow a_i(d, e_i); d := g_i(d, e_i)$$

$$\sum_{e_j} [b_j(d, e_j)] \Rightarrow \tau; d := g_j(d, e_j)$$

The first simplification is that we only consider a closing of this diagram *in one step* (strong confluence). Furthermore, we assume that the diagram is closed by using the same summands j and i again, and moreover we only try the same instance of summand i and j . This situation is depicted in the following picture (we left out the enabling conditions).

$$\begin{array}{ccc} d & \xrightarrow{\tau} & g_j(d, e_j) \\ \downarrow a_i(d, e_i) & & \downarrow a_i(g_j(d, e_j), e_i) \\ & & g_i(g_j(d, e_j), e_i) \\ & & = \\ & & g_i(d, e_i) \xrightarrow{\tau} g_j(g_i(d, e_i), e_j) \end{array}$$

Commutation of τ -summand j with summand i can be expressed by the following Boolean expression over the algebraic data theory:

$$b_i(d, e_i) \wedge b_j(d, e_j) \rightarrow \left(\begin{array}{l} b_i(g_j(d, e_j), e_i) \\ \wedge b_j(g_i(d, e_i), e_j) \\ \wedge a_i(d, e_i) = a_i(g_j(d, e_j), e_i) \\ \wedge g_i(g_j(d, e_j), e_i) = g_j(g_i(d, e_i), e_j) \end{array} \right)$$

If τ -summand j commutes with all summands i (including j), it can be safely marked as a confluent τ -summand. In fact, in case also $a_i(d, e_i) = \tau$, we also have to allow that $g_i(d, e_i) = g_j(d, e_j)$. As strong confluence implies confluence, the transitions generated by τ -summand j will form a confluent subset in the sense of Definition 2. Because the union of two confluent subsets constitutes a confluent subset, it is safe to label multiple summands in the same linear process.

3.3 Automated Theorem Prover

In order to prove formulae of the above kind, we have built a theorem prover for Boolean combinations over a user-defined algebraic data type. In [16] we show how an extension of binary decision diagrams (BDD) enhanced with term rewriting can be applied to these formulae. This is along the lines of the BDDs extended with equality developed in [7]. Given a formula, the prover returns an equivalent but ordered BDD. If this BDD equals TRUE, the pair (i,j) commutes.

If the resulting BDD doesn't equal TRUE, then the formula could not be proved, and τ -summand i cannot be marked as confluent. Note that this may be due to the fact that it is not confluent, or due to the fact that the prover is inherently incomplete (simple equalities over an abstract data type are undecidable already, let alone arbitrary Boolean expressions). In this case, the prover provides some diagnostics, on the basis of which user interaction is possible.

The user can add equations to the data specification, or provide an invariant. It is possible to add new equations, provided they hold in the initial model. Proving correctness of the new equations requires induction, which is beyond our theorem prover. The new equations could be proved either manually, or using a separate interactive theorem prover.

In some cases, the formula is not valid in the initial model, but it would hold for reachable states d . In this case, the user may supply an invariant Inv and the confluence formulae are proved under the assumption $Inv(d)$. Of course such an invariant must be checked separately. This can be done with the same prover, by checking that Inv holds initially, and is preserved by all summands i :

$$b_i(d, e_i) \wedge Inv(d) \rightarrow Inv(g_i(d, e_i))$$

4 Optimization by Symbolic Prioritization

Combining the previous sections, we can now mark certain transitions as being confluent by using an automated theorem prover, and subsequently generate a

reduced state space by the on-the-fly reduction algorithm. However, the confluence marks can also be used to apply an optimization to the specification, i.e. on the symbolic level. Let transition j be a deterministic (i.e. without local variables) transition which is marked as confluent:

$$[b_j(d)] \Rightarrow \tau_c; d := g_j(d)$$

Now because the j summand is confluent, it may be given priority to other summands, as long as loops are avoided. To avoid loops, we will only give summand j priority just after a non-marked (visible or non-visible) transition. So let another summand i be given, which is not marked as confluent:

$$\sum_{e_i} [b_i(d, e_i)] \Rightarrow a_i(d, e_i); d := g_i(d, e_i)$$

Now, if we can prove that j is always enabled after the i -transition, we can combine the i and j summand in one step. Enabledness of j can be represented by the formula $b_i(d, e_i) \rightarrow b_j(g_i(d, e_i))$. This formula is sent to the prover, and upon success, we modify summand i to become:

$$\sum_{e_i} [b_i(d, e_i)] \Rightarrow a_i(d, e_i); d := g_j(g_i(d, e_i))$$

We call this transformation *symbolic prioritization*. One advantage of this symbolic optimization is that the intermediate state $g_i(d, e_i)$ needs not be explored during state space generation. Another advantage is that this optimization often gives rise to a cascade of other possible optimizations, such as elimination of unused variables removal of dead code, based on reachability analysis. In many cases the confluent j summand becomes unreachable (in the sense that in all reachable states d , $b_j(d)$ is false), in which case it can be removed from the specification. This in turn can give rise to elimination of state variables that were only used in summand j .

A very interesting effect is that we can now possibly mark more transitions as confluent. Recall that we only mark “strong confluence”, where a diverging pair is closed in one step. After symbolic prioritization, we might detect confluence also when the diverging pair can be closed in two steps, as illustrated in Figure 2. Of course, this process can be iterated.

In the system on the left, automatic confluence marking will not detect the confluence of τ -summand (2), because the divergence with a -summand (1) cannot be closed in one step. However, typically τ -summand (3) will be detected to be confluent, because no other summands are enabled in its source state. The marking of summand (3) is denoted by the τ_c -label. Note that after a -summand (1) it is always possible to perform the marked τ_c -summand (3). Hence symbolic prioritization can be applied, and we obtain the system on the right. In the new situation summand (2) becomes strongly confluent, so it will be detected by a second application of automatic confluence detection. Due to the confluence of summand (2), the state space generation algorithm will now visit a single path through this graph.

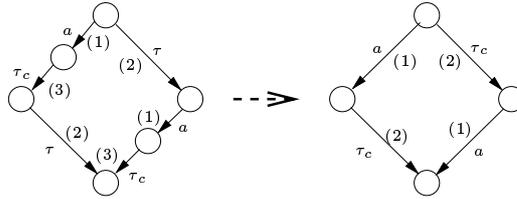


Figure 2. More confluence detection after symbolic prioritization

system	original state space		reduced state space		total costs	
	states	transitions	states	transitions	states	transitions
abp	97	122	29	54	97	98
brp	1952	2387	1420	1855	1952	2275
mutex	96	192	26	46	56	75
DKR(3)	67	124	2	1	20	19
DKR(5)	864	2687	2	1	32	31
DKR(7)	18254	77055	2	1	72	71
Firewire(10)	72020	389460	6171	22668	8443	23325
Firewire(12)	446648	2853960	27219	123888	40919	127016
Firewire(14)	2416632	17605592	105122	544483	167609	557419
Lift1	38000	112937	10448	30823	13326	33701
Lift2	214202	684419	60548	205527	76948	221927

Figure 3. Benchmarks for confluence detection and on-the-fly reduction

5 Applications

We applied our method to several distributed algorithms, protocols and industrial case studies. A number of experiments are described in detail in [16]. Figure 3 shows the reduction obtained by confluence detection and on-the-fly reduction. For each system, we list the size of the original and the reduced state space, and also – in order to allow fair comparisons – the total costs including the number of nodes and transitions that are visited during the TSCC-computation.

The first rows refer to the alternating bit protocol, the bounded retransmission protocol, and a mutual exclusion algorithm. Furthermore, $DKR(n)$ refers to the DKR leader election protocol with n parties; $Firewire(n)$ to the Firewire Tree Identify protocol for n components from the IEEE 1394 bus standard; and the lift entries refer to a case study with distributed lifts [6], used for lifting car trucks by several lift legs.

As a conclusion, we note that the contribution of confluence reduction to toy examples is rather modest. However, on the industrial case studies (Firewire, Lift) the reduction is notable. On the DKR protocol the reduction is even dra-

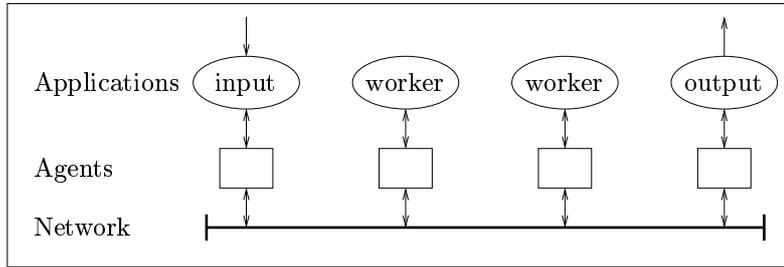


Figure 4. Splice architecture

matic (the number of visited states goes from exponential down to $n \log n$). We now discuss two experiments in more detail.

5.1 Leader election protocol

In the DKR (Dolev-Klawe-Rodeh) leader election protocol [4], n parties are connected in a ring by n channels, modeled as unbounded queues. These parties exchange messages, which are not visible for the outside world. After a finite number of messages, the party with the highest identification performs the action “I’m the leader”.

This algorithm allows for a remarkable state space reduction, viz. from exponential to linear in the number of parties. The theorem prover detects that all τ -summands are confluent, even when n is unknown. Given a concrete number of parties, the generation algorithm finds a completely deterministic path representing the whole state space. So the state space is immediately reduced to a single transition, labeled “I’m the leader”. We remark that also the traditional partial order reduction can be applied to this example.

5.2 Shared data space systems

We also studied distributed systems based on shared data space architectures, such as Splice [3]. A Splice system consists of a number of application processes, that coordinate through agents, which are coupled via some network. The agents locally maintain multi-sets of data items (the distributed data space), into which applications can write new items, and from which applications can read items. The agents distribute their items by asynchronously sending messages to each other over the network.

Figure 4 depicts a simple Splice system, with a producer and a consumer. In between, several workers independently take items from their local storage, perform some computation, and write results back in the space. We want to prove transparency of the number of workers. See [12] for the full case study, which heavily relies on using our confluence reduction.

	original space	reduced state space		after prioritization	
		# states	cost	# states	cost
Ssplice(1,2)	85362	15	75	9	45
Ssplice(2,2)	18140058	69	644	9	65
Ssplice(3,2)	??	297	5151	9	101
Ssplice(1,4)	??	83	743	25	169
Ssplice(2,4)	??	1661	29936	25	249
Ssplice(3,4)	??	31001	1057187	25	393
Ssplice(1,6)	??	317	3657	56	425
Ssplice(2,6)	??	14387	326832	56	630
Ssplice(3,6)	??	??	??	56	999

Figure 5. Splice benchmarks with symbolic prioritization

This communication mechanism is a-synchronous, and leads to much non-determinism: messages from one agent are sent to the others in any order. Consequently, the agents receive messages in various orders, even when they originate from the same agent. By proving confluence, it is detected that all these different orders are equivalent. In fact, the on-the-fly reduction algorithm computes a reduced state space, as if there were only one global multi-set of data items.

Another reduction is possible within the workers. They read any message from their agent, and write some computed result back. Note that such transactions cannot be represented by super-deterministic transitions, because a worker can start with any message in the current set of its agent. Therefore, traditional partial-order reduction methods, which are based on super-determinism, fail on this example. However, 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.

For this example, we also needed symbolic prioritization: although the transactions commute, the corresponding diagrams can only be closed in multiple steps. This corresponds to the situation in Figure 2.

In Figure 5 we applied our reductions on the $\text{Splice}(m,n)$ benchmarks, having m workers and processing n input values. We show the number of generated states, as well as the total number of visited states, including those used in the TSCC search. The size of the original state space (first column) could only be computed in a few cases, and even here we used a parallel machine. After one application of confluence detection and on-the-fly reduction, in most cases the state space could be generated, but this approach doesn't scale well (middle two columns). After symbolic prioritization more transitions could be proven confluent, and running on-the-fly reduction again results in pretty small state spaces (last two columns).

As a final remark, we note that the size of the reduced space doesn't depend on the number of workers anymore. So this example has been solved nearly symbolically in the number of workers.

References

1. S.C.C. Blom. Partial τ -confluence for efficient state space generation. Technical Report SEN-R0123, CWI, Amsterdam, 2001.
2. S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μ CRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of the CAV 2001*, volume 2102 of *LNCS*, pages 250–254. Springer, July 2001.
3. M. Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, July 1993.
4. D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3):245–260, September 1982.
5. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.
6. J.F. Groote, J. Pang, and A.G. Wouters. A balancing act: Analyzing a distributed lift system. In S. Gnesi and U. Ultes-Nitsche, editors, *Proceeding of 6th International Workshop on Formal Methods for Industrial Critical Systems*, pages 1–12, Paris, France, 2001.
7. J.F. Groote and J.C. van de Pol. Equational binary decision diagrams. In M. Parigot and A. Voronkov, editors, *Proc. of LPAR 2000*, LNAI 1955, pages 161–178. Springer, 2000.
8. 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.
9. J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1-2):39–70, 2001.
10. J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 17. Elsevier, 2001.
11. J.F. Groote and M.P.A. Sellink. Confluence for process verification. *Theoretical Computer Science*, 170:47–81, 1996.
12. J.M.M. Hooman and J.C. van de Pol. Formal verification of replication on a distributed data space architecture. In *Proceedings of SAC 2002 (Madrid)*. ACM, 2002.
13. R. Nalumasu and G. Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, (to appear).
14. D. Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In Peled et al. [15], pages 233–258.
15. D.A. Peled, V.R. Pratt, and G.J. Holzmann, editors. *Partial Order Methods in Verification*, volume 29 of *DIMACS Series*. American Mathematical Society, July 1997.
16. J.C. van de Pol. A prover for the μ CRL toolset with applications – Version 0.1. Technical Report SEN-R0106, CWI, Amsterdam, 2001.
17. R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
18. A. Valmari. Stubborn set methods for process algebras. In Peled et al. [15], pages 213–232.
19. M. Ying. Weak confluence and τ -inertness. *Theoretical Computer Science*, 238:465–475, 2000.