Coloured Petri Nets and State Space Generation with the Symmetry Method

Louise Lorentsen

Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, DENMARK.
louisel@daimi.au.dk

Abstract. This paper discusses state space generation with the symmetry method in the context of Coloured Petri Nets (CP-nets). The paper presents the development of the Design/CPN OPS tool which, together with the Design/CPN OE/OS tool, provides fully automatic generation of symmetry reduced state spaces for CP-nets with consistent symmetry specifications. Practical experiments show that the practical applicability of the symmetry method is highly depended on efficient algorithms for determining whether two states are symmetric. We present two techniques to obtain an efficient symmetry check between markings of CP-nets: a technique that improves the generation time and a technique that reduces the memory required to handle the symmetries during calculation. The presented algorithms are implemented in the Design/CPN OPS tool and their applicability is evaluated based on practical experiments.

1 Introduction

The state space of a system is a directed graph with a node for each reachable state of the system and an arc for each state change. From the state space it is possible to verify whether the system possesses a set of desired properties, e.g., the absence of deadlocks, the possibility to always reenter the system's initial state, etc. However, the practical use of state spaces for formal analysis and verification of systems is often limited by the *state explosion problem* [17]: even small systems may have a large (or infinite) number of states, thus making it impossible to construct the full state space of the system. Several reduction techniques have been suggested to alleviate the state explosion problem. An example of such a reduction technique is the *symmetry method* [3, 9, 4, 6]. The symmetry method is not restricted to a specific modelling language. In this paper we work with the symmetry method for Coloured Petri Nets (CP-nets or CPN) [8, 9]. The basic observation behind the symmetry method is that many concurrent and distributed systems posses a degree of symmetry which is also reflected in the state space. The idea behind the symmetry method is to factor out this symmetry and obtain a *condensed state space* which typically is much smaller than the full state space, but from which the same properties can be verified without unfolding to the full state space. The symmetries in such systems can be described by algebraic groups of permutations. For CP-nets the symmetries used for the reduction are induced by algebraic groups of permutations on the atomic colour sets of the CP-net. Hence, we will also use the term *state spaces with permutation symmetries* (SSPSs) to denote the condensed state spaces obtained by using the symmetry method.

In the context of CP-nets the theory of the symmetry method is well developed [9, 8] and a computer tool (the Design/CPN OE/OS tool [11, 10]) that supports state space generation with the symmetry method has been developed. However, the symmetry method in the context of CP-nets has only few applications in practice, e.g. [14, 5]. One of the drawbacks of the Design/CPN OE/OS tool is that it requires the user to implement two predicates determining whether two states/actions are symmetric or not. This requires both programming skills and a deep knowledge of the symmetry method. This is especially the case if the predicates are required to be efficient. However, when constructing SSPSs for CP-nets, it can be observed that the predicates can be automatically deduced [8] provided the algebraic groups of permutations used for the reduction

have been specified[1]. The above observation has motivated the construction of a tool (the Design/CPN OPS tool [13]) which, given an assignment of algebraic groups of permutations to the atomic colour sets of the CPN model, generates the predicates for the Design/CPN OE/OS tool expressing whether two states/actions are symmetric.

The problem of determining whether two states/actions are symmetric is in literature also referred to as *the orbit problem* and an efficient solution to this problem is a central issue for the applicability of the symmetry method. The computational complexity of the orbit problem has been investigated in [3] showing that in general it is at least as hard as *the graph isomorphism problem* for which no polynomial time algorithm is known. However, in the context of CP-nets symmetry can be determined efficiently in a number of special cases, e.g., when the CP-net only contains atomic colour sets [1, 8]. This is, however, not true in general; the problem of determining symmetry between states/actions is complicated by the fact that colour sets can contain arbitrary structural dependencies.

During development of the Design/CPN OPS tool a number of practical experiments have been performed with different strategies for the implementation of the predicates. The practical experiments show that the chosen strategy for the implementation of the predicates greatly influences whether the symmetry method for CP-nets is applicable in practice. The algebraic groups of permutations used for the reduction potentially becomes very large as the system parameters grow. The number of symmetries used for the reduction is potentially $\prod_{A \in \Sigma_A} |A|!$ where Σ_A denotes the atomic colour sets of the CPN model. Hence, efficient handling of the symmetries used for the reduction becomes an important aspect when developing algorithms for the predicates used in the symmetry method.

In this paper we present techniques and algorithms which implements an efficient solution to the orbit problem in SSPS generation for CP-nets. The algorithms presented in this paper are based on general techniques which can be applied independently of model specific details. Hence, the predicates can be automatically constructed by the tool.

The paper is structured as follows. Section 2 presents the symmetry method for CP-nets by means of an example. Section 3 presents the basic generation algorithm for SSPSs. Section 4 introduces a basic solution to the orbit problem for CP-nets that will be used for reference purposes. Section 5 presents algorithms that improve the run-time of the basic algorithm. Section 6 presents algorithms that ensure an compact representation of the symmetries throughout calculation of the SSPSs. Finally, Sect. 7 contains the conclusions.

2 The Symmetry Method for CP-nets

In this section we introduce the symmetry method for CP-nets by means of an example. We will use the example of a distributed database from Sect. 1.3 in [7]. Section 2.1 presents the CPN model of the distributed database and show how the symmetry method can be used to reduce the size of the state space. Section 2.2 explains how the symmetries used in the symmetry method are specified as permutations of atomic colours. Finally, Sect. 2.3 presents a data structure which can be used to represent sets of symmetries in a CP-net.

2.1 Example: Distributed Database

The CP-net for the distributed database is shown in Fig. 1. The CP-net models a simple distributed database with n different sites. Each site contains a copy of all data and this copy is handled by a database manager. Each database manager can change its own copy of the database and send a message to all other database managers requesting them to update their copy of the database.

[1] There are two main approaches in the literature: either the permutations can be automatically deduced from the model, e.g., [2, 16], or explicitly specified by the modeller [8, 9]. The latter approach is based on the belief that the modeller, who constructs the model is familiar with the system modelled and has an intuitive idea of the symmetries present in the model [8].

The distributed database system uses the indexed colour set DBM to model the database managers, the enumeration colour set E to model whether the protocol is active, and the product colour set MES to model the messages. The content of the database and the messages are not modelled. Only header information (the sender and the receiver) is contained in a message.

The distributed database system possesses a degree of symmetry. The database managers are treated similarly, only their identities differ. This symmetry is also reflected in the state space of the distributed database system. The state space for the CPN model with three database managers is shown in the left-hand side of Fig. 2. The idea behind SSPSs is to factor out this symmetry and obtain a smaller state space from which the properties of the distributed database system can be verified without unfolding to the full state space. When constructing the SSPS for the database system we consider two markings/binding elements to be symmetric if they are equal except for a bijective renaming of the database managers. This kind of symmetry (based on bijective renamings) induces two equivalence relations; one on the set of markings and one on the set of binding elements [8]. The basic idea when constructing the SSPS is to lump together symmetric markings/binding elements into one node/arc, i.e., only store one representative from each equivalence class. The right-hand side of Fig. 2 shows the SSPS for the distributed database. The nodes in the full state space (in the left-hand side of the figure) are coloured such that nodes corresponding to symmetric markings have the same colour. The same colours are used in the SSPS (in the right hand side of the figure). From the figure it can be seen that the SSPS only contains one node per equivalence class of symmetric markings.

2.2 Symmetry Specification

The symmetries used for the reduction are obtained from permutations of the atomic colours in the CPN model. Let Σ_A denote the set of atomic colour sets of the CPN model. For each atomic colour set in the CPN model, $A \in \Sigma_A$, we define an algebraic group of permutations Φ_A, i.e., a subgroup of $[A \rightarrow A]$. A symmetry ϕ of the system is a set of permutations of the atomic colour sets of the model, i.e., $\phi = \{\phi_A \in \Phi_A\}_{A \in \Sigma_A}$. In the rest of the paper we will use the term *permutation symmetry* to denote a set of permutations of the atomic colour sets of a CPN model.

The symmetry considered in the distributed database system is a bijective renaming of the database managers. This is obtained by allowing all permutations of the atomic colour set DBM. Hence a permutation symmetry in the distributed database system is a set $\phi = \{\phi_E \in \Phi_E, \phi_{DBM} \in \Phi_{DBM}\}$, where $\Phi_{DBM} = [\text{DBM} \rightarrow \text{DBM}]$ and $\Phi_E = \{\phi_{id}\}$ (where ϕ_{id} is the identity permutation, i.e., $\phi_{id}(\mathsf{e})=\mathsf{e}$). From the permutation symmetries of the CPN model we derive permutations of the structured colour sets, multi-sets, markings and binding elements as described in [8].

A *symmetry specification* of a CP-net is an assignment of algebraic groups of permutations to each of the atomic colour sets of the CP-net and hence determines a group of permutation

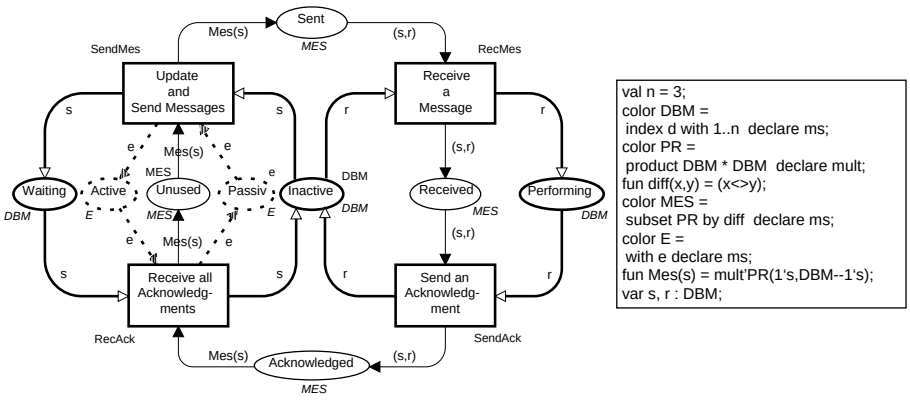

Fig. 1. CP-net for the Distributed Database example.

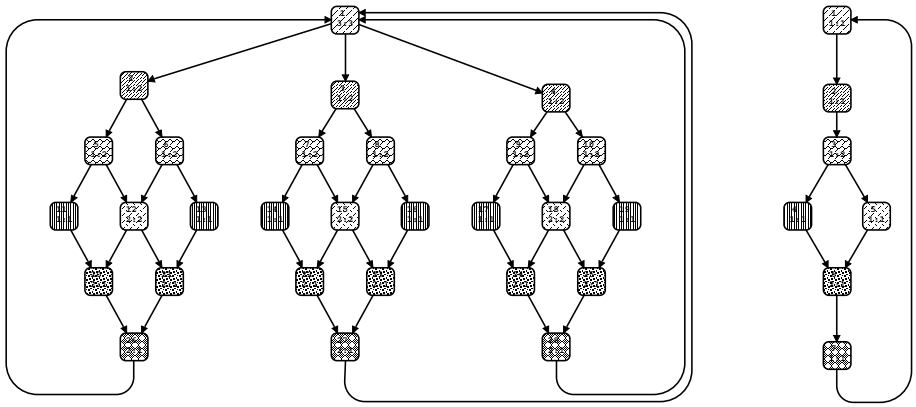

Fig. 2. The full state space (the left-hand side) and SSPS (the right-hand side) of the CP-net for the distributed database example with 3 symmetric database managers($n = 3$).

symmetries. The symmetry specification is required to be consistent [8] which means that it is required to only express symmetries that are actually present in the system. We will use Φ_{SG} to denote the group of permutation symmetries given by a consistent symmetry specification SG. In the rest of the paper we assume that a CP-net with places $P = \{p_1, p_2, ..., p_n\}$ is given together with a consistent symmetry specification SG which determines a group Φ_{SG} of permutation symmetries.

2.3 Restriction Sets

A consistent symmetry specification SG determines a group of permutation symmetries Φ_{SG}. During generation of the SSPS we need some kind of representation of Φ_{SG}. One possibility is to list the permutation symmetries. Since the symmetry groups used for the reduction can be very large, this is not a feasible solution.

A set of permutations of an atomic colour set can instead be represented as a *restriction set*. Restriction sets are introduced in [8] and formally defined in [1]. Here we will introduce restriction sets by means of an example. Below we use a restriction set to represent a subset of [DBM $\rightarrow$ DBM]. The set of permutations mapping `d(1)` to `d(2)` and the set {`d(2)`,`d(3)`} to the set {`d(1)`,`d(3)`} can be represented by the following restriction set:

d(1)	d(2)
d(2) d(3)	d(1) d(3)

Each row in the restriction set introduces a requirement for the set of permutations represented by the restriction set. The individual restrictions (rows) express that the colours on the left-hand side must be mapped into the colours of the right-hand side. In [1] it is proven that restriction sets can be efficiently intersected (while maintaining the compact representation) and that an arbitrary set of permutations can be represented by a set of restriction sets. Hence, restriction sets provide a potentially compact representation of sets of permutations. In the rest of the paper we will use restriction sets to represent sets of permutations of atomic colour sets. Hence, a symmetry specification can be represented by a set of restriction sets for each atomic colour set in the CP-net.

3 Condensed State Space Generation

In this section we give an introduction to the standard algorithm GENERATE-SSPS for construction of the SSPS of a CP-net [8]. *Nodes* and *Arcs* are sets of states (markings) and actions (binding elements), respectively, and it contains the states and actions that are included in the SSPS. *Unprocessed* is a set of states, and contains the states for which we have not yet calculated the successor

states. M_0 denotes the initial state. NEXT(M) is a function calculating the set of possible next moves (an action and the resulting state) from the state M. NODE(M) is a function that checks whether a node symmetric to M is already included in the SSPS. If not, M is added to *Nodes* and *Unprocessed*. Similarly, ARC(M_1,b,M_2) is a function that checks whether a symmetric arc is already included in the SSPS, i.e., an arc consisting of a binding element symmetric to b from a marking symmetric to M_1 to a marking symmetric to M_2. If not, (M_1,b,M_2) is added to *Arcs*.

Algorithm: GENERATESSPS () =

```
 1: Nodes ← {M0}
 2: Arcs ← ∅
 3: Unprocessed ← {M0}
 4: repeat
 5:    select M1 ∈ Unprocessed
 6:    for all (b,M2) ∈ Next(M1) do
 7:       NODE(M2)
 8:       ARC(M1,b,M2)
 9:    end for
10:    Unprocessed := Unprocessed \ {M1}
11: until Unprocessed = ∅
```

The algorithm proceeds in a number of iterations. In each iteration a state (M_1) is selected from *Unprocessed* and the successor states (and actions) are calculated using the NEXT function. For each of the successor states, M_2, it is checked whether *Nodes* already contains a state symmetric to M_2. If not M_2 is added to both *Nodes* and *Unprocessed*. Similar checks are made for the actions. The check for symmetric states and symmetric actions are instances of the *orbit problem*. From the basic generation algorithm it can be seen that efficient generation of the SSPS is highly dependent on the efficiency of the algorithms for determining the following two problems: 1) When reaching a new marking M during generation of the SSPS, is there a marking symmetric to M already included in the SSPS? And 2) When reaching a new arc (M_1, b, M_2) during generation of the SSPS, is there a symmetric arc already included in the SSPS?

GENERATESSPS is implemented in the Design/CPN OE/OS tool and used when calculating SSPSs for CP-nets. In Design/CPN a hash function is used when storing the markings during generation of the SSPS. When reaching a new marking during generation of the SSPS each marking stored with the same hash value is checked to see if it is symmetric to the newly reached marking, i.e., symmetry checks are performed locally between markings in the collision lists. The user of the tool is free to use his own hash function. The only requirement is that the hash function used is symmetry respecting, i.e., symmetric states are mapped to the same hash value. This is the case for the default hash function used in Design/CPN. Hence, when using the Design/CPN OE/OS tool for the generation of SSPSs efficient generation is dependent on the efficiency of the two predicates, P_M and P_{BE}, determining symmetry between markings and binding elements, respectively.

P_M: Given $M_1, M_2 \in \mathbb{M}$ determine whether $\exists \phi \in \Phi$ s.t. $\phi(M_1) = M_2$.
P_{BE}: Given $(t_1, b_1), (t_2, b_2) \in$ BE determine whether $\exists \phi \in \Phi$ s.t. $\phi(t_1, b_1) = (t_2, b_2)$.

The Design/CPN OE/OS tool requires P_M and P_{BE} to be implemented by the user. Implementing such predicates is error-prone for large CPN models and requires both programming skills and a detailed knowledge of the symmetry method. This is especially the case if the predicates are required to be efficient. The required user implementation of P_M and P_{BE} in the Design/CPN OE/OS tool has motivated the development of the Design/CPN OPS tool which given a CP-net and a consistent symmetry specification automatically generates the two predicates, P_M and P_{BE}, needed by the Design/CPN OE/OS tool. In the rest of the paper we will present techniques and algorithms to obtain implementations of P_M and P_{BE} in the Design/CPN OPS tool which are

efficient in practice. The algorithms are independent of the specific CP-net. Hence, the predicates can be automatically generated.

In the following discussions we will concentrate on the markings since the symmetry check between binding elements can be viewed as a special case of symmetry checks between markings: Given a transition t with variables $v_1, v_2, \ldots, v_m$, a binding b of t can be viewed as a vector of singleton multi-sets $(1`b(v_1), 1`b(v_2), \ldots, 1`b(v_m))$, where $b(v)$ denotes the value assigned to v in the binding b. Since transitions cannot be permuted by permutation symmetries in CP-nets determining symmetry between binding elements is the same as determining symmetry between markings. Hence, in the rest of the paper we will present techniques and algorithms to obtain an efficient implementation of P_M, i.e., given $M_1, M_2 \in \mathbb{M}$ determine whether $\exists \phi \in \Phi$ such that $\phi_{\mathbb{M}}(M_2) = M_1$.

4 Basic Algorithm for P_M

In this section we will present a basic algorithm which implements the predicate P_M. Section 4.1 presents the algorithm. Section 4.2 presents experimental results obtained using the Design/CPN OPS tool where the basic algorithm presented in this section is used to determine symmetry between markings.

4.1 Presentation of the Algorithm

The algorithm is based on a simple approach where Φ_{SG}, i.e., the group of permutation symmetries allowed by the symmetry specification SG, is iterated to determine whether $\exists \phi \in \Phi_{SG}$ s.t. $\phi(M_1) = M_2$. The algorithm P_M^{Basic} is given below.

Algorithm: $P_M^{Basic}(M_1, M_2)$

```
1: for all φ ∈ Φ_SG do
2:    if φ(M_1) = M_2 then
3:       return true
4:    end if
5: end for
6: return false
```

The algorithm repeatedly selects a permutation symmetry ϕ from Φ_{SG} (line 1) and tests whether ϕ is a symmetry between the two markings, M_1 and M_2 given as input (lines 2-4). The iteration stops when a permutation symmetry ϕ for which $\phi(M_1) = M_2$ is found (line 3) or the entire Φ_{SG} has been iterated (line 6).

The algorithm P_M^{Basic} potentially tests fewer permutation symmetries than $|\Phi_{SG}|$. This is however not the case if M_1 and M_2 are not symmetric. In that case the algorithm checks the whole Φ_{SG}. Hence, P_M^{Basic} is only useful for CP-nets with few permutation symmetries. This is also supported by the experimental results presented below.

However, before we present the experimental results of the P_M^{Basic} algorithm we will briefly introduce how it is tested whether a permutation symmetry ϕ maps a marking M_1 to another marking M_2 (line 2 in P_M^{Basic}). In [1] it is shown how the set of permutation symmetries between two markings can be determined as the intersection of the sets of permutation symmetries between the markings of the individual places. Hence, to determine whether a permutation symmetry $\phi \in \Phi_{SG}$ is a symmetry between two markings M_1 and M_2, we in turn test the multi-sets constituting the markings of $p_i \in P$. Note that if a permutation symmetry is not a symmetry for the marking of a place $p_i \in P$, i.e., $\phi(M_1(p_i)) \neq M_2(p_i)$ the permutation symmetry ϕ cannot be a symmetry between M_1 and M_2 and therefore there is no need to test the remaining places in P. Using

the ideas presented in [1] we obtain an algorithm TestPermutationSymmetry which given a permutation symmetry ϕ and two markings, M_1 and M_2, tests whether $\phi(M_1) = M_2$.

Algorithm: TestPermutationSymmetry(ϕ,M_1,M_2) =

```
1: for all p_i ∈ P do
2:    if φ(M_1(p_i)) ≠ M_2(p_i) then
3:       return false
4:    end if
5: end for
6: return true
```

The algorithm repeatedly selects a place $p_i \in P$ of the CP-net (line 1) and tests whether ϕ is a symmetry between the markings of p_i in M_1 and M_2 (line 2-4). If not, ϕ is not a symmetry between M_1 and M_2, otherwise a new place is tested. The iteration proceeds until a place $p_i \in P$ for which ϕ is not a symmetry is found (line 3) or all places have been tested (line 6).

4.2 Experimental Results of the P_M^{Basic} Algorithm

This section presents experimental results obtained using the Design/CPN OPS tool. The following results are obtained using an implementation of P_M^{Basic} to determine whether two markings are symmetric. A similar approach is used for the implementation of P_{BE}.

The Design/CPN OPS tool represents Φ_{SG} as a restriction set. When checking symmetry between two markings using P_M^{Basic} Φ_{SG} is listed and the permutation symmetries from the list are removed and tested until a permutation symmetry ϕ is found for which $\phi(M_1) = M_2$ or the list is empty.

SSPSs have been generated for two different CP-nets in a number of configurations. The CP-nets used in the experiments are briefly described below. For a detailed description of the CP-nets we refer to [8, 12].

Commit [12]. A CP-net modelling a two-phase commit protocol with a coordinator and w symmetrical workers.

Distributed database [8]. The CP-net presented in Sect. 2 modelling the communication between n symmetrical database managers.

Table 1 shows the generation statistics for of the SSPS for different configurations of the two CP-nets using the P_M^{Basic} algorithm. The CP-net column gives the name (C stands for commit and D stands for distributed database) and configuration of the CP-net for which the SSPS is generated as well as the number of permutation symmetries given by the symmetry specification SG used for the reduction. The Count column gives two numbers: the total number of times the P_M predicate is called during calculation of the SSPS and the number of calls which evaluate to true, i.e., the number of those calls which determine that the two markings are symmetric. The Tests column presents statistics on the number of permutation symmetries applied to markings during generation of the SSPS: Total gives the total number of permutation symmetries applied to markings during generation of the SSPS, P_M^{Basic} gives the average number of permutation symmetries applied in each call of P_M^{Basic}, P_M^{Basic}=`true` gives the average number of permutation symmetries applied in each call of P_M^{Basic} which evaluates to true (the case where iteration of the entire Φ_{SG} is potentially avoided), and finally, % $|\Phi_{SG}|$ gives the average percentage of the permutation symmetries which are tested in a call of P_M^{Basic}. Finally, the Time column gives the number of seconds it took to generate the SSPS for the given CP-net. A '–' in an entry means that the SSPS could not be generated within 600 seconds. All experimental results presented in this paper are obtained on a 333MHz PentiumII PC running Linux. The machine is equipped with 128 Mb RAM.

From Table 1 it can be seen that when system parameters increase the number of permutation symmetries tested increase significantly. This is caused by the increasing size of Φ_{SG}. From the

				P_M^{Basic}				
CP-net		Count		Tests				Time
Con.	$\|\Phi_{SG}\|$	P_M	P_M=`true`	Total	P_M^{Basic}	P_M^{Basic} =`true`	% $\|\Phi_{SG}\|$	Secs
C_2	2	11	7	19	1.73	1.57	78.5	0
C_3	6	26	19	90	3.46	2.53	42.0	0
C_4	24	53	41	488	9.21	4.88	20.3	0
C_5	120	95	76	3,242	34.1	12.7	10.5	0
C_6	720	157	127	27,297	174	44.86	6.2	23
C_7	5,040	–	–	–	–	–	–	–
D_2	2	4	2	7	1.75	1.50	75.0	0
D_3	6	14	6	61	4.36	2.17	36.0	0
D_4	24	35	15	533	15.2	3.53	14.7	0
D_5	120	71	31	5,037	70.9	7.64	6.37	0
D_6	720	126	56	51,693	410	23.1	3.20	16
D_7	5,040	–	–	–	–	–	–	–

Table 1. Generation statistics for SSPS generation using the P_M^{Basic} algorithm.

% $|\Phi_{SG}|$ column it can be seen that the average percentage of Φ_{SG} which are tested in P_M^{Basic} decreases when the system parameters increase. However, the increasing size of Φ_{SG} makes it impossible to generate the SSPS for the two CP-nets when system parameters, i.e., the number of concurrent readers or database managers, becomes greater than 6. This is also caused by the approach where Φ_{SG} is listed before the permutation symmetries are tested. For systems of increasing size $|\Phi_{SG}|$ imply that the entire Φ_{SG} cannot be represented in memory and, thus, generation of the SSPS is not possible. It should be noted that the results presented in Table 1 depends on the order in which the permutation symmetries are applied. The order used for the experiments is the same order in each call of P_M^{Basic} based on a recursive unfolding of the restriction set.

We conclude that the experiments performed using P_M^{Basic} in generation of SSPSs show that the run-time incurred by the iteration of Φ_{SG} becomes significant when system parameters grow. Hence, in order to make the calculation of SSPSs for CP-nets applicable in practice we need to carefully consider the number of permutation symmetries tested in the generation of the SSPSs. The next section presents techniques which improve P_M^{Basic} in this direction.

5 Approximation Techniques

In this section we will present an algorithm which presents an improved implementation of the predicate P_M^{Basic}. Section 5.1 presents the algorithm. Section 5.2 presents experimental results obtained using the Design/CPN OPS tool where the improved algorithm presented in this section is used to determine symmetry between markings.

5.1 Presentation of the Algorithm

The problem when using P_M^{Basic} for the symmetry check between markings is that in the worst case $|\Phi_{SG}|$ permutation symmetries will be checked. When determining symmetry between markings a selection of simple checks can in many cases determine that two markings are not symmetric or determine a smaller set of permutation symmetries that have to be checked.

In this section we will present a new algorithm for P_M which given two markings, M_1 and M_2, calculates a set Ψ_{M_1,M_2} such that $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\} \subseteq \Psi_{M_1,M_2} \subseteq \Phi_{SG}$. Hence, Ψ_{M_1,M_2} is a super-set of the set of permutation symmetries mapping M_1 to M_2. If $\Psi_{M_1,M_2} = \emptyset$ we can conclude that M_1 and M_2 are not symmetric. However, if Ψ_{M_1,M_2} is non-empty we have to test the individual permutation symmetries in Ψ_{M_1,M_2}. In worst case $|\Psi_{M_1,M_2}|$ permutation symmetries have to be checked. This is the case if M_1 and M_2 are not symmetric. If M_1 and M_2 are symmetric then in worst case $|\Psi_{M_1,M_2}| - |\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}| + 1$ permutation symmetries have to be checked. Hence, the goal of the approximation technique is to construct Ψ_{M_1,M_2} as close to $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}$ as possible.

In [1] it was shown that if a CP-net only contains atomic colour sets then the set $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}$ can be determined efficiently. This is, however, not the case if the CP-net contains structured colour sets. Nevertheless, we will use the technique to efficiently obtain an approximation Ψ_{M_1,M_2} of $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}$, thus reducing the number of permutation symmetries which have to be checked compared to the approach used in P_M^{Basic}. This is obtained at the cost of doing the approximation. In the following we will show how such an approximation can be obtained efficiently when Φ_{SG} is represented as a restriction set. The approximation technique is based on ideas from [8, 1].

The set of permutation symmetries mapping a marking M_1 to another marking M_2 can be found as the intersection of sets of permutation symmetries mapping $M_1(p_i)$ to $M_2(p_i)$ for all $p_i \in P$. Similarly, it is shown in [8] and proved in [1] how the set of permutation symmetries between such markings of places, i.e., multi-sets, can be determined as the intersection over sets of symmetries between sets with equal coefficient in the multi-sets, i.e., for a permutation symmetry to be a symmetry between ms_1 and ms_2 it must ensure that a colour appearing with coefficient c in ms_1 must be mapped into a colour appearing with the same coefficient in ms_2. We will illustrate using the CP-net of the Distributed Database (Fig. 1) as an example.

Let ms_1=`1‘d(2)+1‘d(3)` and ms_2=`1‘d(1)+1‘d(2)` be two markings of a the place `Inactive` with colour set `DBM`. In ms_1 two colours (`d(2)` and `d(3)`) appear with coefficient 1 and one colour (`d(1)`) appear with coefficient 0. We can express the multi-set of coefficients as `2‘1+1‘0`. In ms_2 it is also the case that two colours (`d(1)` and `d(2)`) appear with coefficient 1 and one colour (`d(3)`) appear with coefficient 0. Hence, ms_2 has the same multi-set of coefficients as ms_1 namely `2‘1+1‘0`. A permutation ϕ_{DBM} of the colour set `DBM` is a permutation mapping ms_1 to ms_2 if ϕ_{DBM} ensures that a colour appearing with coefficient 1 in ms_1 is mapped to a colour appearing with coefficient 1 in ms_2, and similar for the rest of the coefficients (here just 0). Hence, we can construct a restriction set representing the set of permutations between ms_1 and ms_2 by constructing a restriction for each of the coefficients appearing in ms_1 and ms_2.

Coefficient 0:	`d(1)`	`d(3)`
Coefficient 1:	`d(2) d(3)`	`d(1) d(2)`

In the above example the two multi-sets had the same multi-sets of coefficients. This is a necessary requirement for the two multi-sets to be symmetric [1]. If not, the left and right-hand sides of the constructed restrictions do not contain the same number of elements, and thus does not represent a valid set of permutations. Multi-sets of coefficients are formally defined in [1]. We define multi-sets of coefficients using the notation used in this paper below and present an algorithm which calculates the set of permutation symmetries between two multi-sets over an atomic colour set.

Definition:

For a multi-set ms over a colour set C we define $\text{Coefficients}_C(ms)$ as the set of coefficients appearing in ms:

$$\text{Coefficients}_C(ms) = \{i \in \mathbb{N} | \exists c \in C \text{ such that } ms(c) = i\}$$

Let ms be a multi-set over a colour C. For $i \in \text{Coefficients}_C(ms)$ we define the *i-coefficient-class* for ms as the set of colours in C appearing with coefficient i:

$$\mathsf{C}_i(ms) = \{c \in C | ms(c) = i\}$$

We define the *multi-set of coefficients* for ms by

$$\text{Cfms}(ms) = \{ms(i)‘i\}_{i \in \mathsf{Coefficients}_C(ms)}$$

Based on the above definitions we formulate an algorithm FINDPERMUTATIONS which given two multi-sets ms_1 and ms_2 over an atomic colour set $A \in \Sigma_A$ calculate the set $\{\phi_A \in \Phi_A \mid \phi_A(ms_1) = ms_2\}$.

Algorithm: FINDPERMUTATIONS$_{ms}(ms_1,ms2)$
1: **if** CFMS(ms_1) = CFMS(ms_2) **then**
2: **return** $\{(C_i(ms_1),C_i(ms_2)\}_{i \in \text{Coefficients}(ms_1)}$
3: **else**
4: **return** $\emptyset$
5: **end if**

The algorithm tests whether CFMS(ms_1) = CFMS(ms_2) (line 1), i.e., the multi-set of coefficients are equal. If not ms_1 and ms_2 are not symmetric [1], i.e., the empty set is returned (line 4), otherwise a restriction set is constructed containing a restriction $(C_i(ms_1),C_i(ms_2))$ for each of the coefficients i in COEFFICIENTS(ms_1) (line 2).

Given two markings, M_1 and M_2, the algorithm FINDPERMUTATIONSYMME-TRIES$_M$ calculates the a set of permutation symmetries Ψ_{M_1,M_2} as the intersection of Φ_{SG} and the sets of permutations between the markings of the individual places with atomic colour sets (calculated using FINDPERMUTATIONS$_{ms}$).

Algorithm: FINDPERMUTATIONSYMMETRIES$_M(M_1,M_2)$ =
1: $\Phi' \leftarrow \Phi_{SG}$
2: **for all** $p \in \{p' \in P \mid p'$ has an atomic colour set$\}$ **do**
3: $\Phi' \leftarrow \Phi' \cap$ FINDPERMUTATIONS$_{ms}(M_1(p),M_2(p))$
4: **end for**
5: **return** Φ'

If the CP-net only contains places with atomic colour sets the set Ψ_{M_1,M_2} of permutation symmetries calculated using FINDPERMUTATIONSYMMETRIES$_M$ is equal to the set $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}$. If the CP-net also contains places with structured colour sets then Ψ_{M_1,M_2} is a superset of $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}$, i.e, $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\} \subseteq \Psi_{M_1,M_2}$. We will use FINDPERMUTATIONSYMMETRIES$_M$ to improve the P_M^{Basic} algorithm presented in Sect. 4, i.e., to reduce the number of permutation symmetries which have to be checked. The new algorithm P_M^{Approx} is presented below.

Algorithm: $P_M^{Approx}(M_1, M_2)$
1: **for all** $\phi \in$ FINDPERMUTATIONSYMMETRIES$_M$ (M_1,M_2) **do**
2: **if** TESTPERMUTATIONSYMMETRY' (ϕ,M_1,M_2) **then**
3: **return** `true`
4: **end if**
5: **end for**
6: **return** `false`

The algorithm repeatedly selects a permutation symmetry ϕ from the set of permutation symmetries approximated using FINDPERMUTATIONSYMMETRIES$_M$ (line 1) and tests whether ϕ is a symmetry between the two markings (lines 2-4). The iteration stops when a permutation symmetry ϕ for which $\phi(M_1) = M_2$ is found (line 3) or the entire set has been iterated (line 6). P_M^{Approx} (M_1, M_2) uses TESTPERMUTATIONSYMMETRY' (ϕ,M_1,M_2), a modified version of the algorithm TESTPERMUTATIONSYMMETRY presented in Sect. 4, to determine whether $\phi(M_1) = M_2$. The difference is that given a permutation symmetry ϕ and two markings, M_1 and M_2, TESTPERMU-

TATIONSYMMETRY' only test ϕ on the places which have a structured colour set. The markings of the places with atomic colour sets are already accounted for in the approximation and do not have to be tested again.

The complexity of the calculation of FINDPERMUTATIONSYMMETRIES$_M$ is independent of $|\Phi_{SG}|$. This is a very attractive property, since the experimental results presented in Sect. 4 showed that iterating the group of permutation symmetries is not applicable in practice when the symmetry specification determines a large set of permutation symmetries.

If the CP-net contains places with atomic colour sets P_M^{Approx} potentially tests fewer permutation symmetries than P_M^{Basic}. In P_M^{Basic} at most $|\Phi_{SG}| - |\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}| + 1$ permutation symmetries are checked when determining whether two markings are symmetric, whereas at most $|$FINDPERMUTA-TIONSYMMETRIES$_M(M_1, M_2)| - |\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}| + 1$ permutation symmetries are tested using P_M^{Approx}. The experimental results presented later in this section show that for the two CP-nets used in the experiments the approximation is very close (or even equal) to the exact set of permutation symmetries mapping M_1 to M_2. Hence, the number of permutation symmetries which have to be tested is very low in practice. Furthermore, if the multi-sets of coefficients are different for markings no permutation symmetries have to be tested to determine that the markings are not symmetric. It should be noted that a necessary requirement for two markings M_1 and M_2 to be symmetric is that CFMS$(M_1(p_i))$ =CFMS$(M_2(p_i))$ for all places $p_i \in P$ (also for places with structured colour sets). Hence, an obvious way to improve P_M^{Approx} is to test the equality of multi-sets of coefficients for places with structured colour sets before checking any permutation symmetries. Places with atomic colour sets are already accounted for in the approximation.

5.2 Experimental Results of the P_M^{Approx} Algorithm

In this section we will present experimental results obtained using an implementation of P_M based on the P_M^{Approx} algorithm. The approximation operated directly on the restriction sets and the approximate set Ψ_{M_1,M_2} is also represented as a restriction set. Before checking the permutation symmetries in Ψ_{M_1,M_2} the approximated set is represented as a list. The permutation symmetries from the list are removed and checked until a permutation symmetry ϕ is found for which $\phi(M_1) = M_2$ or the list is empty. The experimental results are obtained using the two CP-nets presented in Sect. 4.

Table 2 presents generation statistics for SSPSs for different configurations of the two CP-nets using the P_M^{Approx} algorithm. The first four columns are the same as the first four columns in Table 1 presenting the generation statistics using the P_M^{Basic} algorithm. The CP-net column gives the name and configuration of the CP-net for which the SSPS is generated as well as the number of permutation symmetries given by the symmetry specification. The Count column gives two numbers: the total number of times the P_M predicate is called during calculation of the SSPS and the number of calls of P_M which evaluate to true, i.e., the number of calls which determines that the two markings are symmetric. The last six columns are specific to the P_M^{approx} algorithm. The Cfms column gives the number of calls of P_M^{Approx} for which the multi-sets of coefficients are different for the two markings, i.e., the number of calls of P_M^{Approx} where no permutation symmetries are tested. The Tests column presents statistics on the number of permutation symmetries applied to markings during generation of the SSPS: Total gives the total number of permutation symmetries applied to markings during generation of the SSPS, P_M^{Approx} gives the average number of permutation symmetries applied in each call of P_M^{Approx}, P_M^{Approx}=`true` gives the average number of permutation symmetries applied in each call of P_M^{Approx} which evaluates to true (the case where iteration of the entire Φ_{SG} potentially is avoided), and finally, $\%|\Phi_{SG}|$ gives the average percentage of the permutation symmetries which are tested in a call of P_M^{Approx} during generation of the entire SSPS. Finally, the Time column gives the number of seconds it took to generate the SSPS for the CP-net in the given configuration.

From Table 2 it can be seen that checking the multi-sets of coefficients before testing any permutation symmetries in P_M^{Approx} reduces the number of permutation symmetries tested compared

to P_M^{Basic}. It is worth noticing that in all calls of P_M^{Approx} which evaluated to false no permutation symmetries are tested, i.e., in all cases the multi-sets of coefficients differ. This is of course highly dependent on the CPN model and is a question of the amount of redundancy encoded in markings of places with structured colour sets. Furthermore, all calls of P_M^{Approx} which evaluated to true only in average requires one permutation symmetry to be tested. This is not a general fact of the technique. However, it is our experience from other experiments that in practice many CP-nets contains a degree of redundancy such that the approximation based on the atomic colour sets of the CP-net often is very close (or equal) to the exact set of permutation symmetries mapping one marking to another.

Even though the number of permutation symmetries tested after approximating the set of permutation symmetries is 1 for both CP-nets in all configurations it can be seen that SSPSs could not be generated for more than 7 database managers or workers. The reason is the memory required by P_M^{Approx}: in the implementation of P_M^{Approx} used for the practical experiments the approximated set of permutation symmetries is listed before testing the permutation symmetries. Hence, even though the approximation determines the exact set of permutation symmetries in worst case $|\Phi_{SG}|$ permutation symmetries are listed. Thus, in order to make the method applicable in practice we need to carefully consider the representation of the sets of permutation symmetries during generation of the SSPSs. This is the topic of the next section.

6 Lazy Listing

In the previous sections we have used sets of restriction sets to represent sets of permutation symmetries. The approximation technique presented in Sect. 5 operates directly on the restriction sets. However, in the implementations of both P_M^{Basic} and P_M^{Approx} the permutation symmetries are listed before they are checked. The major drawback of the approach presented in the previous section is that even though the approximation is exact or very close to $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}$, i.e., only few permutation symmetries have to be checked, the entire approximated set is listed. The experimental results presented in Sect. 5 also showed that this approach is not applicable in practice since the memory use becomes a serious bottleneck as system parameters grow. The main goal of this section is therefore to improve the P_M^{Approx} algorithm such that a compact representation of the approximated set of permutation symmetries is maintained during calculation of P_M^{Approx}.

In this section we will present an algorithm which is an improved implementation of the predicate P_M^{Approx}. Section 6.1 presents the algorithm. Section 6.2 presents experimental results obtained using the Design/CPN OPS tool where the improved algorithm presented in this section is used to determine symmetry between markings.

				P_M^{Approx}					
CP-net		Count		cfms	Tests				Time
Con.	$\|\Phi_{SG}\|$	P_M	P_M=true		Total	P_M^{Approx}	P_M^{Approx} =true	$\%\|\Phi_{SG}\|$	Secs
C_2	2	11	7	4	7	1	1	50.0	0
C_3	6	26	19	7	19	1	1	16.7	0
C_4	24	53	41	12	41	1	1	4.17	0
C_5	120	95	76	19	76	1	1	0.83	0
C_6	720	157	127	30	127	1	1	0.14	1
C_7	5,040	242	197	45	197	1	1	0.02	60
C_8	40,320	–	–	–	–	–	–	–	–
D_2	2	4	2	2	2	1	1	50	0
D_3	6	14	6	8	6	1	1	16.67	0
D_4	24	35	15	20	15	1	1	4.17	0
D_5	120	71	31	40	31	1	1	0.83	0
D_6	720	126	56	70	56	1	1	0.14	0
D_7	5,040	204	92	112	92	1	1	0.02	7
D_8	40,320	–	–	–	–	–	–	–	–

Table 2. Generation statistics for SSPS generation using the P_M^{Approx} algorithm.

6.1 Presentation of the Algorithm

One way of viewing a set of permutation symmetries (represented as a set of restriction sets) is as a tree. Each level in the tree corresponds to possible images of a given colour. Hence, leafs in the tree represent the permutation symmetries given by the permutation of the individual elements found by following the path from the root to the leaf. Figure 3 shows a tree representing Φ_{SG} for the CPN model of the distributed database with 3 database managers and a consistent symmetry specification SG which allow all possible permutations of the atomic colour set `DBM`. The leafs of the tree represent the 6 different permutations symmetries in Φ_{SG}. Below each leaf the permutation symmetry is represented by a restriction set.

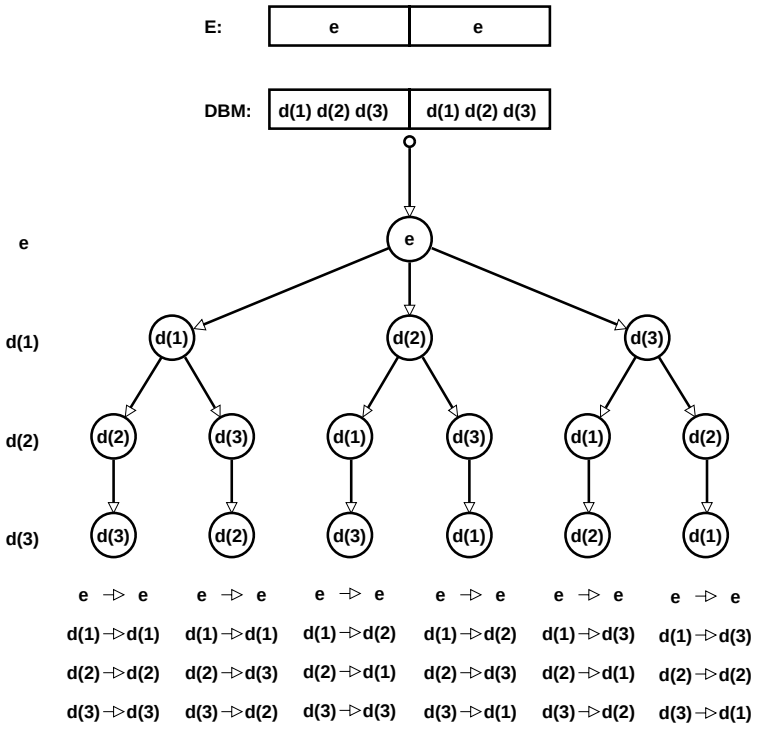

Fig. 3. All permutation symmetries in Φ_{SG} represented as a tree.

When testing a set of permutation symmetries on a marking in the P_M^{Basic} and P_M^{Approx} algorithms we first unfolded the restriction sets to a list of permutation symmetries and then applied the permutation symmetries from an end (until one was found or the entire set was checked). With the approach presented in this section we instead make recursive unfoldings of restriction sets based on a depth first generation of the 'tree view'; each node in the tree corresponds to a recursive call. Each time a leaf is reached the corresponding permutation symmetry is checked. If the permutation symmetry is a symmetry between the two markings checked we conclude that the markings are symmetric (and the iteration stops) otherwise the permutation symmetry is thrown away and the algorithm backtracks to generate the next permutation symmetry. In this way at most one permutation symmetry is contained in memory at a time. In a recursive call corresponding to the ith layer of the tree the algorithm only needs to keep track of the restriction set in the root as well as the images of the colours corresponding to the layers $1,..,(i-1)$. Hence, instead of listing potentially $\prod_{A \in \Sigma_A} |A|!$ permutation symmetries the algorithm needs to represent in the worst case images of at most $\Sigma_{A \in \Sigma_A} |A|$ colours plus the restriction set in the root. An algorithm

for such lazy listing of permutations symmetries represented by sets of restriction sets is shown below.

Algorithm: LazyList (i,Φ',M_1,M_2)

```
 1: if SinglePermutationSymmetry(Φ') then
 2:    φ ← GetPermutationSymmetry(Φ')
 3:    return TestPermutationSymmetry(φ, M1, M2)
 4: else
 5:    col ← GetColour(i)
 6:    images ← GetImages(Φ',col)
 7:    found ← false
 8:    repeat
 9:       select col' ∈ images
10:       images ← images\{col'}
11:       found ← LazyList(i+1,Split(Φ',col,col',M1,M2))
12:    until images = ∅ ∨ found = true
13: end if
14: return found
```

The algorithm takes four arguments: i is the depth of the call (corresponds to the level in the tree), Φ' is a set of restriction sets representing a set of permutation symmetries, and M_1 and M_2 are the two markings which are checked. First LazyList (i,Φ',M_1,M_2) tests whether the set of restriction sets Φ' given as input represents a single permutation symmetry (line 1). If this is the case a leaf in the tree is reached and the result of applying the permutation symmetry is returned (lines 2-3). If the set of restriction sets represents more than one permutation symmetry (line 4) we have reached an internal node in the tree and a number of depth-first recursive calls are made (lines 8-12). The algorithm uses a number of functions which we will briefly describe below.

SinglePermutationSymmetry(Φ') returns true if Φ' represents a set of a single permutation symmetry and false otherwise.

GetPermutationSymmetry(Φ') returns one of the permutation symmetries in the set represented by the set of restriction sets Φ'.

TestPermutationSymmetry(ϕ, M_1, M_2) tests whether $\phi(M_1) = M_2$.

GetColour(i) returns the colour associated to the i'th level in the tree.

GetImages(Φ',col) returns the possible images of *col*, i.e., the right-hand side of the restriction in Φ in which *col* is contained in the left-hand side.

Split(Φ',col,col') returns a new set of restriction sets which is similar to Φ' except that the restriction containing *col* has been split into two: one containing *col* in the left-hand side and *col'* in the right-hand side and one containing the remaining colours.

An algorithm combining approximation and lazy listing in the symmetry check between markings is given below.

Algorithm: $P_M^{Approx+Lazy}(M_1,M_2)$

```
1: Φ' ← FindPermutationSymmetries_M (M1,M2)
2: return LazyList (1,Φ',M1,M2)
```

The algorithm approximates the set of permutation symmetries using the technique presented in Sect. 5 (line 1). To avoid the lengthy listing the permutation symmetries in the approximated set are checked using the LazyList algorithm (line 2).

6.2 Experimental Results of the $P_M^{Approx+Lazy}$ Algorithm

In this section we present experimental results obtained using an implementation of P_M based on the $P_M^{Approx+Lazy}$ algorithm. The implementation represents the approximated set of permutation symmetries as a set of restriction sets. During calculation a compact representation is maintained using depth-first recursive unfoldings.

Table 3 presents the generation statistics for the generation of the SSPS for different configurations of the two CP-nets using the $P_M^{Approx+Lazy}$ algorithm. The CP-net column gives the name and configuration of the CP-net for which the SSPS is generated as well as the number of permutation symmetries given by the symmetry specification. The next three columns give the time it took to generate the corresponding SSPS using the three algorithms P_M^{Basic}, P_M^{Approx}, and $P_M^{Approx+Lazy}$, respectively.

CP-net	$\lvert\Phi_S G\rvert$	Time (secs)		
		P_M^{Basic}	P_M^{Approx}	$P_M^{Approx+Lazy}$
C_2	2	0	0	0
C_3	6	0	0	0
C_4	24	0	0	0
C_5	20	0	0	0
C_6	720	23	1	0
C_7	5,040	–	60	1
C_8	40,320	–	–	1
C_9	362,880	–	–	2
C_{10}	3,628,800	–	–	3
C_{15}	$1.3 \cdot 10^{12}$	–	–	77
D_2	2	0	0	0
D_3	6	0	0	0
D_4	24	0	0	0
D_5	120	0	0	0
D_6	720	16	0	0
D_7	5,040	–	7	1
D_8	40,320	–	–	2
D_9	362,880	–	–	4
D_{10}	3,628,800	–	–	8
D_{12}	$4.7 \cdot 10^8$	–	–	30
D_{15}	$1.3 \cdot 10^{12}$	–	–	151

Table 3. Generation statistics for SSPS generation using the $P_M^{Approx+Lazy}$ algorithm.

From Table 3 it can be seen that using the $P_M^{Approx+Lazy}$ algorithm it is possible to generate SSPSs for CP-nets with very large symmetry groups. When applying $P_M^{Approx+Lazy}$ for testing the permutation symmetries the same number of permutation symmetries is of course tested as if the permutation symmetries are listed beforehand using the P_M^{Approx} approach. However, the compact representation maintained during calculation saves space since the permutation symmetries are represented by sets of restriction sets. Hence, when combining the idea of lazy listing with the idea of approximations as presented in Sect. 5 significant speed up is gained. The reason is that in practice the approximations are often very close to or even equal to the exact set of symmetries between two markings. Thus, the number of permutation symmetries which have to be tested from large permutation groups is usually very small. Furthermore, the memory use of $P_M^{Approx+Lazy}$ caused by the size Φ_{SG} is no longer a bottleneck of the practical applicability of the method.

7 Conclusions

We have presented techniques and algorithms to determine whether two markings of CP-nets are symmetric. The algorithms presented are based on general and model independent techniques. Hence, the algorithms can be automatically generated for arbitrary CP-nets. The techniques are implemented in the Design/CPN OPS tool [13] which automatically generates the predicates P_M

and P_{BE} needed for the Design/CPN OE/OS Tool [11, 10]. The Design/CPN OPS tool has been used to conduct the experimental results presented in this paper.

The approximation technique that P_M^{Approx} is based on is introduced in [8, 1]. The contribution of this paper is to automate and implement the technique as well as integrate the technique into SSPS generation. The technique is specific to markings of CP-nets and is as such not general for the symmetry method.

The need for compact representations and avoidance of testing the entire group of permutation symmetries is, however, not specific to CP-nets. The algorithms and experimental results presented in this paper are therefore also relevant in other formalisms than CP-nets.

During SSPS generation we store an arbitrary marking from each equivalence class (the first state from the equivalence class encountered during generation of the SSPS). Another strategy is to calculate a canonical representative for each equivalence class. The symmetry check can then be reduced to a simple equivalence check. In [15] we have presented an algorithm for calculation of canonical markings of CP-nets. The algorithm requires the calculation of the minimal marking obtained as a result of applying a set of permutation symmetries. The algorithm for calculation of canonical markings of CP-nets presented in [15] encounters the same problem as the P_M^{Basic} algorithm presented in Sect. 4: applying the entire group of permutation symmetries is unfeasible in practice. In [15] we use algebraic techniques to reduce the number of iterations. Even with the use of algebraic techniques the canonicalization of markings experiences problems in practice due to the memory use required when working with large sets of permutation symmetries. The lazy listing approach presented in this paper can directly be used in the problem studied in [15] and it is envisioned that the lazy listing approach can alleviate the bottleneck caused by the memory use in [15].

It is possible to combine the use of algebraic techniques and the techniques presented in this paper to obtain a solution for P_M. However, since the approximation techniques only applies when two markings are compared the approximation techniques presented in this paper cannot be used directly in the algorithm for calculation of canonical markings of CP-nets.

References

1. R.D. Andersen, J.B. Jørgensen, and M. Pedersen. Occurrence Graphs with Equivalent Markings and Self-Symmetries. Master's thesis, Department of Computer Science, University of Aarhus, Denmark, 1991. Only available in Danish: Tilstandsgrafer med ækvivalente mærkninger og selvsymmetrier.
2. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. On Well-Formed Coloured Nets and Their Symbolic Reachability Graph. In K. Jensen and G. Rozenberg, editors, *High-level Petri Nets*, pages 373–396. Springer-Verlag, 1991.
3. E.M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetries in Temporal Model Logic Model Checking. In Springer-Verlag, editor, *Proceedings of CAV'93*, volume 697 of *Lecture Notes in Computer Science (LNCS)*, pages 450–462. Springer-Verlag, 1993.
4. E.A. Emerson and A. Prasad Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9, 1996.
5. D.J. Floreani, J. Billington, and A. Dadej. Designing and Verifying a Communications Gateway Using Coloured Petri Nets and Design/CPN. In J. Billington and W. Reisig, editors, *Proceedings of ICATPN'96*, volume 1091 of *Lecture Notes in Computer Science*, pages 153–171. Springer-Verlag, 1996.
6. C.N. Ip and D.L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, 9, 1996.
7. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, 1992.
8. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, 1994.
9. K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9, 1996.
10. J.B. Jørgensen and L.M. Kristensen. *Design/CPN OE/OS Graph Manual.* Department of Computer Science, University of Aarhus, Denmark, 1996.
    Online: `http://www.daimi.au.dk/designCPN/`.

11. J.B. Jørgensen and L.M. Kristensen. Computer Aided Verification of Lamport's Fast Mutual Exclusion Algorithm Using Coloured Petri Nets and Occurrence Graphs with Symmetries. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):714–732, July 1999.
12. L. M. Kristensen. *State Space Methods*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 2000.
13. L. Lorentsen. *Design/CPN OPS Graph Manual*. Department of Computer Science, University of Aarhus, Denmark, 2002.
    Online: `http://www.daimi.au.dk/~louisel/`.
14. L. Lorentsen and L.M. Kristensen. Modelling and Analysis of a Danfoss Flowmeter System. In M.Nielsen and D.Simpson, editors, *Proceedings of the 21th International Conference on Application and Theory of Petri Nets (ICATPN'2000)*, volume 1825 of *Lecture Notes in Computer Science*, pages 346–366. Springer-Verlag, 2000.
15. L. Lorentsen and L.M. Kristensen. Exploiting stabilizers and paralellism in state space generation with the symmetry method. In *Proceedings of the Second International Conference on Application of Concurrency to System Design (ICACSD'01)*, pages 211–220. IEEE, 2001.
16. K. Schmidt. How to Calculate Symmetries of Petri nets. *Actae Informaticae*, 36(7):545–590, 2000.
17. A. Valmari. The State Explosion Problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.