Construction of Occurrence Graphs with Permutation Symmetries Aided by the Backtrack Method

Jens Bæk Jørgensen

Computer Science Department, University of Aarhus
Ny Munkegade, Bldg. 540
DK–8000 Aarhus C, Denmark

Phone: +45 89 42 31 88
Telefax: +45 89 42 32 55
E-mail: jbj@daimi.aau.dk

Abstract

This paper recalls the concept of occurrence graphs with permutation symmetries (OS-graphs) for Coloured Petri Nets. It is explained how so-called self-symmetries can help to speed up construction of OS-graphs. The contribution of the paper is to suggest a new method for calculation of self-symmetries, the Backtrack Method. The method is based on the so-called Backtrack Algorithm, which originates in computational group theory. The suggestion of the method is justified, both by identifying an important general complexity property and by obtaining encouraging experimental performance measures.

Topics. Coloured Petri Nets, reduced state spaces, occurrence graphs with permutation symmetries, self-symmetries, computational group theory, backtrack searches.

1 Introduction

Verification by occurrence graphs (also known as state spaces and reachability graphs) is obstructed by the well-known state explosion problem: Even for relatively small systems, the occurrence graphs are often so large that they cannot be constructed given the computer technology presently available. Alleviation of this inherent complexity problem is a major challenge of research. Several ingenious approaches have been proposed. Among them are *occurrence graphs with permutation symmetries (OS-graphs)* [7, 8, 11], which are defined for *Coloured Petri Nets* (*CP-nets* or *CPN*) [6].

The basic observation behind OS-graphs is that quite often, some markings of a CP-net are similar, and induce similar behaviours. Such markings are called *symmetric*. Based on symmetric markings, a reduced occurrence graph called an OS-graph can be constructed, which is typically much smaller than the ordinary full occurrence graph (O-graph). The OS-graph and the O-graph contain the same information, and, if the graphs are finite, a lot of useful verification results are easily derivable from either of them.

In an OS-graph, the nodes correspond to equivalence classes of markings and the arcs correspond to equivalence classes of binding elements. The crucial task in construction of OS-graphs is to decide whether two given markings, or two given binding elements, are equivalent. In [3], it is proved that the equivalence test is computationally at least as hard as the graph isomorphism problem [12], and it is speculated that the test may even be NP-complete. Thus, it is unlikely to find general efficient methods for construction of OS-graphs. We must resort to pursue usable heuristics.

The work behind this paper began with a literature search within the area of computational group theory in the pursuit of usable heuristics. As concluded in [4], computational group theory seems to be a viable area to look into. This paper explains how so-called self-symmetries [1, 7] are applicable. Then, a usable heuristics for calculation of self-symmetries is presented and justified — the Backtrack Method, which is based on the so-called Backtrack Algorithm [2, 9].

The paper is organised as follows: Section 2 describes OS-graphs in more detail. Section 3 introduces the concept of self-symmetries and explains why they are useful. The Backtrack Algorithm is outlined and the Backtrack Method is defined in Section 4. Sections 5 and 6 demonstrate the feasibility of the Backtrack Method to calculate self-symmetries by means of two examples. The conclusions are drawn in Section 7. The reader is assumed to be familiar

with CP-nets including the terminology and notation from [6].

2 OS-Graphs

In this section, we informally explain the concept of OS-graphs. For a more thorough description and formal definitions, the reader is referred to either of [7, 8, 11].

OS-graphs are based on the notion of *symmetry*, which appears when a system is composed of similar components, whose identities are immaterial with respect to occurrence graph analysis. As an example, consider the well-known dining philosophers system. A state of this system, in which the eating philosophers are numbers 1 and 3, is symmetric with a state, in which the eating philosophers are numbers 3 and 5. The first state can be mapped to the second by the permutation, which rotates philosopher i into philosopher $i+2$ (modulo the number of philosophers). Symmetry is also present in many real-world systems.

Definition of an OS-graph for a CP-net requires that two equivalence relations are present — one on the set of markings and one on the set of binding elements. The OS-graph has a node for each equivalence class of reachable markings (for two equivalent markings, either both or none of them are reachable). The OS-graph has an arc between two nodes, if and only if there is a marking in the equivalence class of the source node in which a binding element is enabled, and whose occurrence leads to a marking in the equivalence class of the destination node. There is exactly one arc for each equivalence class of binding elements with this property.

The equivalence relations are required to be on a certain form. They must be induced by *symmetry groups*, which are algebraic groups. One symmetry group is associated with each colour set of the considered CP-net. The symmetry group determines how the colours are allowed to be permuted, e.g., by arbitrary permutations, by rotations only (for an finite, ordered colour set), or by no permutations at all.

For the *atomic* colour sets, which are the colour sets defined without reference to other colour sets (e.g., booleans or enumeration types), the symmetry groups are chosen by the user. For the other colour sets, the *structured* colour set (e.g., Cartesian products or lists), the symmetry groups are inherited, i.e., automatically derived, from the user-chosen symmetry groups.

The symmetry groups induce a group of *permutation symmetries*, whose

elements are functions that can be applied to markings and binding elements. Two markings are *equivalent*, also called *symmetric*, if and only if there exists a permutation symmetry mapping one of the markings to the other. Similarly for binding elements.

When the symmetry groups are chosen such that inherent symmetries of the considered CP-net are captured (in a well-defined way), the OS-graph is said to be based on a *consistent permutation symmetry specification*. The consistency ensures that the OS-graph in fact does contain the same information as the corresponding O-graph.

The algorithm to construct OS-graphs is a straightforward modification of the standard algorithm to construct O-graphs [7]. The test of equality before a new marking is inserted, is replaced by a test of equivalence. In addition, an equivalence test before insertion of a new arc is introduced.

For some CP-nets, e.g., those containing only atomic colour sets, the equivalence test can be performed efficiently [1]. If the group of permutation symmetries is finite, the test may be performed as a brute force application of all permutation symmetries, one by one, to see if there exists one that maps a newly generated marking to one already represented in the graph. This works satisfactory for CP-nets with a small number of permutation symmetries. E.g., this is the case if all colour sets are small. However, in general, the computational complexity of the brute force approach is prohibitive. Often, the number of permutation symmetries is exponential in the size of the system under consideration. Efficient heuristics to aid the equivalence test are needed.

Computer tool support for OS-graphs is provided by the *Design/CPN OS-Graph Tool* [10].

3 Self-symmetries

In this section, we describe the concept of self-symmetries [1, 7], and argue that self-symmetries are usable to speed up the construction of OS-graphs.

Let a CP-net and a consistent permutation symmetry specification, determining the set Φ of permutation symmetries, be given. The *set of self-symmetries* for a marking[1] M is:

[1]Sets of self-symmetries are defined analogously for markings of places and for arbitrary subsets of colour sets. A set of self-symmetries is what in standard algebraic terminology is known as a *set stabiliser* or *isotropy subgroup*.

$$\Phi_M = \{\phi \in \Phi | \phi(M) = M\}.$$

The set of self-symmetries can be used to speed up the construction of the OS-graph because of two reasons, which will be described below. Subsequently, construction of OS-graphs by exploiting self-symmetries is compared with the standard construction algorithm.

3.1 Fewer Equivalence Tests

The first reason to use self-symmetries was originally recognised by Kurt Jensen and Peter Huber, but first described and formalised in [1]: Fewer equivalence tests can be conducted.

The consistency of the permutation symmetry specification ensures that for all reachable markings M_1, M_2, all binding elements b, and all permutation symmetries ϕ:

$$M_1[b > M_2 \Rightarrow \phi(M_1)[\phi(b) > \phi(M_2).$$

Hence, for $\phi \in \Phi_{M_1}$:

$$M_1[b > M_2 \Rightarrow M_1[\phi(b) > \phi(M_2).$$

Thus, if two enabled binding elements are symmetric by a self-symmetry of M_1, only one of them needs to be processed, i.e., it is only necessary to investigate occurrence of one of the binding elements. The other binding element is known in advance to correspond to an arc and a destination node already included in the graph.

This property is very important with respect to speeding up the construction of OS-graphs. As noted in Section 1, the crucial and main time-consuming task in this procedure is the equivalence test, i.e., to test if a newly generated marking must be inserted in the graph. Thus, if the number of newly generated markings can be reduced, a number of expensive equivalence tests can be saved.

Self-symmetries allow for generation of fewer new markings: Each time a new marking is inserted in the OS-graph, the set of self-symmetries is calculated. When a marking is processed, the set of enabled binding elements (if non-empty) is partitioned into what is called *self-symmetry equivalence classes*. Two binding elements are *self-symmetry equivalent*, if and only if there exists a self-symmetry of the considered marking mapping one of the

binding elements to the other. Then, instead of processing all enabled binding elements, only one representative from each self-symmetry equivalence class is processed. As a consequence, fewer new markings are generated, and, thus need to be tested for insertion. Hence, fewer equivalence tests are conducted.

3.2 Faster Equivalence Tests

The second reason to use self-symmetries was first recognised and described in [1]: Faster equivalence tests can be obtained.

Two markings M_1 and M_2 are equivalent if and only if the set

$$\Phi_{M_1,M_2} = \{\phi \in \Phi | \phi(M_1) = M_2\}$$

is non-empty. As noted in Section 2, the brute force test of all permutations in Φ to compute Φ_{M_1,M_2} is only sensible if Φ is small. In general, more ingenuity is required. The set of self-symmetries Φ_{M_1} is a subgroup of Φ, and induces a partition of Φ into what in algebra is called cosets[2]. A *coset* of Φ_{M_1}, for $\psi \in \Phi$, is a set on the form:

$$\psi \circ \Phi_{M_1} = \{\psi \circ \gamma | \gamma \in \Phi_{M_1}\}.$$

It is easy to see that for $\phi, \phi' \in \psi \circ \Phi_{M_1}$:

$$\phi(M_1) = \phi'(M_1).$$

Thus, to test $\Phi_{M_1,M_2} = \emptyset$, it is sufficient to test one permutation from each coset of Φ_{M_1}.

3.3 Comparison with the Standard Algorithm

Compared to the standard algorithm to construct OS-graphs (Proposition 2.5 in [7]), exploitation of self-symmetries induces additional computations: Each time a new marking is inserted in an OS-graph, its set of self-symmetries must be calculated. Moreover, when a marking is processed, the enabled binding elements must be partitioned into the self-symmetry equivalence classes.

So, a natural question to ask is whether exploitation of self-symmetries actually risks slowing down the construction of an OS-graph. The answer

[2]The term coset is, without confusion, used as an abbreviation for left coset — as opposed to right cosets, which are not considered in this paper.

is that it is possible, but unlikely: If the self-symmetry equivalence classes for all markings inserted in the graph are trivial, i.e., of size 1, a slow-down may happen, because all enabled binding elements are still processed, i.e., no equivalence tests are saved.

On the other hand, one single equivalence test has the same complexity as calculation of a set of self-symmetries. Thus, each time a set of self-symmetries can prevent a new marking from being generated, but discarded because it is already represented in the graph, much is gained. Think, e.g., of an OS-graph with many thousands of nodes.

Even if the self-symmetry equivalence classes are trivial in all markings inserted in the OS-graph, exploitation of self-symmetries may speed up construction of the OS-graph anyway — namely if a more efficient equivalence test sometimes is possible.

Computation of the self-symmetry equivalence classes in a given marking involves a number of (self-symmetry) equivalence tests, which, in the worst case, is quadratic in the size of the set of enabled binding elements. Fortunately, an equivalence test for binding elements is much faster than an equivalence test for markings [1] (p. 52). Moreover, there is only the set of self-symmetries to test; not the entire symmetry group.

An obvious drawback of exploitation of self-symmetries is that more space is needed: Memory is required to store the set of self-symmetries until the self-symmetry equivalence classes have been computed.

4 The Backtrack Method

In this section, we define the Backtrack Method, which we will suggest for calculation of self-symmetries for markings of CP-nets. The Backtrack Method is based on the Backtrack Algorithm [2, 9], an algorithm from computational group theory, which was invented by the mathematician C.C. Sims in the early seventies. We sketch the basic ideas of the Backtrack Algorithm, present the method for calculation of self-symmetries, and describe an important complexity property.

4.1 The Backtrack Algorithm

The *Backtrack Algorithm* searches a subgroup Ψ of the group Σ_n of all permutations of the set $\{1, \ldots, n\}$. The algorithm solves the following general

problem: Given a property $P \in [\Psi -> \{true, false\}]$ on the permutations in Ψ, find the subset $\overline{P} = \{\phi \in \Psi | P(\phi)\}$ of permutations satisfying P. P must adhere to the two essential *Backtrack Algorithm prerequisites*: First, P must be efficiently decidable, i.e., there must exist an efficient algorithm to calculate $P(\phi)$ for $\phi \in \Psi$. Second, $\overline{P}$ must constitute a subgroup of Ψ.

The Backtrack Algorithm at any time maintains a result group , $\subseteq \overline{P}$ of permutations already found to satisfy P. , is initialised to consist of the identity function, which in known to satisfy P because of the second Backtrack Algorithm prerequisite. The Backtrack Algorithm exploits that it is sufficient to test only one permutation from each coset of , for satisfaction of P. For each coset, either all or none of the permutations of that coset satisfy P — a generalisation of the reason motivating the use of self-symmetries in the construction of OS-graphs described in Section 3.2.

The search domain of the Backtrack Algorithm is represented as a tree. Each time a permutation satisfying P is found, , is extended and its cosets are recomputed[3]. This corresponds to a pruning of the tree representation of the search domain.

The Backtrack Algorithm is implemented in the general-purpose mathematics software package *GAP* (*Groups, Algebra, and Programming*) [13]. This implementation was used in the practical experiments to be described in Sections 5 and 6.

4.2 Calculation of Self-symmetries

The *Backtrack Method* for calculation of self-symmetries treats the places of the given CP-nets in some order. The marking of each place, which is a multi-set, is split into a number of sets — one set for each positive coefficient appearing, containing all elements with that coefficient. As shown in [1], the set of self-symmetries for the marking is the intersection of the sets of self-symmetries for these sets.

Now, assume that it is possible to find a bijective correspondence between the sets derived from the marking, as described above, and sets that can be given as argument to the Backtrack Algorithm. For each of these sets, the

[3]Technical remarks: , is represented as a set of *generators*. Finding a permutation satisfying P adds one new generator to , , but many more elements.

, remains a subgroup throughout the computation. Thus, computation of the cosets are always well-defined

Backtrack Algorithm is applicable, because calculation of the set of self-symmetries for a set is a special instance of the general problem solved by the algorithm. The property *the permutation ϕ is a self-symmetry of the set s* can be found using the property $P(\phi) : \phi(s) = s$. P adheres to both of the Backtrack Algorithm prerequisites: P is efficiently decidable and a set of self-symmetries, as noted previously, constitutes a subgroup.

The Backtrack Method for calculation of self-symmetries amounts to application of the Backtrack Algorithm to each of the sets derived from the marking combined with a computation of intersection. The method is represented as pseudo code in Figure 1. It is assumed that a CP-net, a group of permutation symmetries `SymGroup`, and a marking `M` are given. `PlaceSet` denotes the set of places of the CP-net. Upon termination of the Backtrack Method, the set of self-symmetries for `M` is contained in `SelfSyms`.

Four auxiliary functions are used: `PosCoefs` returns the set of positive coefficients appearing in the multi-set given as argument. `FindSet` returns the set of all elements in the multi-set given as first argument appearing with the coefficient given as second argument. `SelfSymFunc` takes a set `s` as argument, and returns a function, which takes a permutation as argument, and returns true if the permutation is a self-symmetry for `s`, and false otherwise. `BacktrackAlg` denotes the Backtrack Algorithm.

```
SelfSyms := SymGroup;
forall p in PlaceSet do
      forall c in PosCoefs(M(p)) do
      begin
          s := FindSet(M(p),c);
          SelfSymProperty := SelfSymFunc s;
          SelfSyms := BacktrackAlg(SelfSyms, SelfSymProperty);
      end;
end;
```

Figure 1: The Backtrack Method.

Note in Figure 1 that the calculation of intersection is done implicitly, i.e., by successive reductions of the initial search domain. Permutations, which are not self-symmetries for the first place treated, cannot be self-symmetries for the considered marking as such. Therefore, the group of self-symmetries for the place treated first is used as search domain in the treatment of the next place and so forth.

4.3 Fast Tester Property

In our process of experimenting with the Backtrack Method and trying to understand the Backtrack Algorithm itself, we discovered an important complexity property: As formally proved in [9], when the Backtrack Algorithm searches Σ_n for a property P satisfied by all permutations, i.e., $\overline{P} = \Sigma_n$, only $n - 1$ permutations are tested. This property will be referred to as the *fast tester property*, because $n - 1$ is a very small number compared to the number of permutations to test in a brute force approach, namely the size of Σ_n, which is $n!$. The fast tester property generalises to say that when a subgroup of Σ_n of size less than or equal to $m!$ is searched for a property satisfied by all members, at most $m - 1$ permutations are tested.

The fast tester property has a great impact on the successful use of the Backtrack Method: Often, a CP-net deliberately has a high degree of redundancy, in the sense that the markings of some places are determined by the markings of other places. Therefore, it is often the case that after having treated only a few places, the set of self-symmetries is actually computed. No further reductions of the search domain can be made. For the remaining places, in each application of the Backtrack Algorithm, the property given as argument is constantly true. Therefore the remaining places will be treated very quickly.

The experiment described in Section 5 below will exhibit the practical value of the fast tester property.

5 Experiment 1 — Data Base Managers

In this section, we describe an experiment with the Backtrack Method on an example CP-net, the *data base manager system* shown in Figure 2. We introduce the example, show how the experiment is implemented in GAP, present performance measures, and discuss observations.

5.1 Experiment

The system consists of a number, `n`, of sites (in Figure 2, `n` = 4). Each site contains a copy of a data base, which is maintained by a *data base manager*. The data bases must be kept consistent, i.e., when a manager on one site updates his copy of the data base, he must send a *message* to the others, so that they can update their copies accordingly. While the other managers are

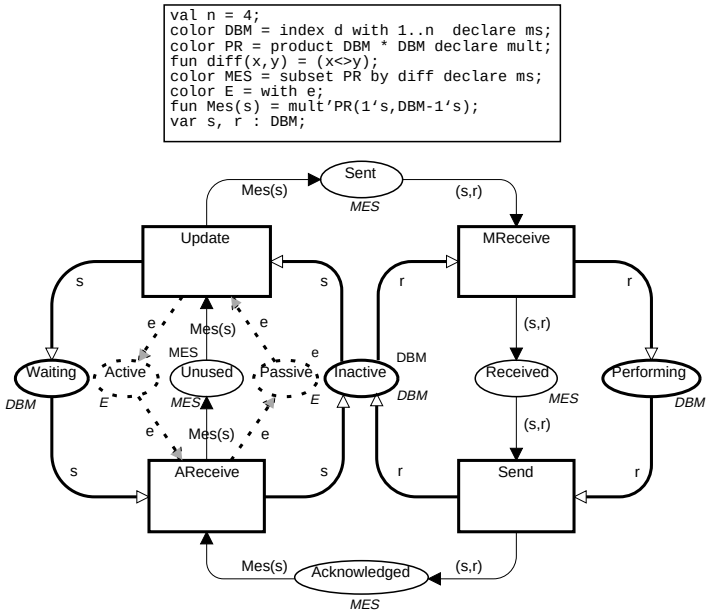

Figure 2: The data base manager system.

performing their updates, the manager who first did the update is waiting. When a manager has updated his site, he sends an acknowledgement to the waiting data base manager. When all sites have been updated, the waiting manager returns to being inactive. The CP-net describes the synchronisation mechanism, not the actual data contained in the data bases. For a more detailed description, see [6].

In the system with `n` managers, the colour set `DBM` describing the data base managers is atomic and naturally represented by the set $\{1, 2, \ldots, n\}$. The colour set `MES` describing the messages is structured and represented by the set of pairs $\{(s, r) \in \texttt{DBM} \times \texttt{DBM} | s \neq r\}$.

We will assume that the symmetry group for `DBM` is the group of all permutations, i.e., equal to Σ_n. The symmetry group for the structured colour set `MES` is inherited from the symmetry group for `DBM`, by the mapping in $[\Sigma_n \times \texttt{MES} \rightarrow \texttt{MES}]$ given by $(\phi, (s, r)) \mapsto (\phi(s), \phi(r))$.[4]

[4]To give a complete description of the permutation symmetry specification, the symmetry group for the atomic unit colour set `E`, used to model synchronisation, is $\{id\}$. The symmetry group for auxiliary structured colour set `PR`, necessary to define the colour set

The objective in the experiment is for different values of `n` to measure the time used by the Backtrack Method for calculation of the set of self-symmetries for a reachable marking. For all `n`, the considered markings are similar: One data base manager is on the place `Waiting`, and the others are distributed evenly on the places `Inactive` and `Performing`, i.e., the size of the marking of `Inactive` and the size of the marking of `Performing` differ at most with 1. The place `Acknowledged` is empty.

The CP-net for the data base manager system deliberately has a high degree of redundancy. Therefore, the markings of the places `Sent`, `Received`, `Passive`, and `Active` are determined by the markings of the other places.

5.2 Implementation

An extract of the GAP code implementing the experiment is shown in Figure 3. The code implements the Backtrack Method for this specific example.

Some remarks on the calculation: The Backtrack Method prescribes calculation of the set of self-symmetries for a marking of a CP-net as intersection of a number of sets of self-symmetries for sets. First of all, in all reachable markings of the data base manager system, no proper multi-sets appear, i.e., no multi-sets with coefficients greater than 1. Therefore, the splitting of a marking into sets yield exactly one set for each place. Second, the places `Passive` and `Active`, and the empty place `Acknowledged` are ignored in the calculation. This can safely be done because a permutation which is a self-symmetry for the markings of the places actually treated, trivially also is a self-symmetry for the markings of these three places; again, due to redundancy in the CP-net, and the fact that any permutation symmetry is a self-symmetry for the empty multi-set.

The two predefined functions `SubgroupProperty` and `Stabilizer` are used. `SubgroupProperty` implements the Backtrack Algorithm. It takes a permutation group and a property satisfying the Backtrack Algorithm prerequisites as arguments. `Stabilizer` is a special instantiation of the Backtrack Algorithm, which computes the stabiliser in the given permutation group of the given set, i.e., the set of self-symmetries. `Stabilizer` is directly applicable for the three places with the atomic colour set `DBM`. In contrast, `SubgroupProperty` is used for the places with the structured colour set `MES`. For these places, the `Stabilizer` function is not applicable,

`MES`, is inherited. As we shall see, neither `E` nor `PR` need to be considered in the experiment.

because `Stabilizer` takes sets of integers as argument, not sets of pairs. `SubgroupProperty` is called with user-defined functions as property arguments. The shown code is for `n` = 10 managers, and thus the initial search domain is the group Σ_{10}, called `S10` in the code extract. The name of a place abbreviates the marking of that place.

When the computation terminates, G6 is the set of self-symmetries for the marking under consideration

```
G1 := Stabilizer(S10, Waiting);
G2 := Stabilizer(G1, Performing);
G3 := Stabilizer(G2, Inactive);
G4 := SubgroupProperty(G3, selfsym_Sent);
G5 := SubgroupProperty(G4, selfsym_Received);
G6 := SubgroupProperty(G5, selfsym_Unused);
```

Figure 3: GAP code — data base manager system.

5.3 Statistics

Table 1 contains the main statistics gathered in the experiment. The table has four columns showing, respectively, `n`, the size of the initial search domain (`n!`), the size of the set of self-symmetries for the considered marking, and finally the time used for the calculation. All measures of time presented in this paper are in milliseconds of CPU time on a Sun4 workstation[5].

`n`	Search Domain	Self-symmetries	Time
3	6	1	80
4	24	2	140
5	120	4	230
10	3,628,800	2,880	2,410
15	approx. $1.3 \cdot 10^{13}$	25,401,600	15,000
20	approx. $2.4 \cdot 10^{19}$	approx. $1.3 \cdot 10^{13}$	50,100

Table 1: Main statistics — data base manager system.

[5]The measures were obtained in late 1993, explaining why a Sun4 — as opposed to a more modern computer — was used.

As basis for two of the observations to be described in Section 5.4, a fine-grained measure of the times to calculate the individual groups `Gi` in the code extract shown in Figure 3, i.e., for $\mathtt{n} = 10$, was made, and is shown in Table 2.

i	1	2	3	4	5	6
Time	360	210	190	310	330	1,010

Table 2: Statistics — calculation of `Gi`.

5.4 Observations

We now list and discuss a number of observations made during the experiment.

Number of self-symmetries

The data base managers can only be on three possible places. The markings considered for Table 1 all distribute the `n` data base managers as much as possible. Distribution enforces discrimination of tokens. Because of the way permutation symmetries are defined as maps from markings to markings (in [7]), two tokens on different places can never be interchanged by a self-symmetry for the considered marking.

For a reachable marking M of the considered CP-net, the size of the set of self-symmetries is $|M(\mathtt{Inactive})|! \cdot |M(\mathtt{Performing})|!$.

The more distributed the tokens in a marking are, the lower is the number of self-symmetries for that marking. E.g., for the reachable markings considered for Table 1, the formula above equals $\lfloor \mathtt{n}/2 \rfloor! \cdot \lfloor (\mathtt{n}-1)/2 \rfloor!$. It is not possible to find reachable markings that have fewer self-symmetries. The reachable marking with the highest number is the initial marking, where all data base managers are on `Inactive`, and for which all $\mathtt{n}!$ permutations are self-symmetries.

The experiment (supplemented with more measures than those presented in Section 5.3) showed that the time for the calculation was low, when the number of self-symmetries was high, and vice versa. Therefore, the time for the calculation of the set of self-symmetries of other reachable markings of

the data base manager system will often be lower than the ones shown in Table 1. In other words, the table shows worst-case performance measures.

Fast testing

In [1] (p. 144), calculation of the set of self-symmetries for a set over a structured colour set is characterised as potentially complex. Calculation of the groups `G4` to `G6` involved the structured colour set `MES`. In spite of that, the calculations were done remarkably fast. For `n` = 20, the set of self-symmetries was found in less than one minute from a search domain of size $2.4 \cdot 10^{19}$. The reason for the impressive speed is the fast tester property of the Backtrack Algorithm: In the GAP code of Figure 3, after having constructed `G2`, i.e., after having treated only two places, the search domain was reduced to the set of self-symmetries for the given marking. Consequently, `G2` = `G3` = `G4` = `G5` = `G6`. For `n` = 10, `G2` had size 2,880 which is less than 7!. Therefore, the number of tests conducted by the Backtrack Algorithm for the places corresponding to `G3` to `G6` were less than 6 for each place.

That the actual times used for the calculations of the individual groups differed, as can be seen from Table 2, was due to the fact that the tests had different complexities. The tests done in the calculation of `G3` were applications of permutations to a subset of the atomic colour set `DBM`, while the tests done in the calculation of `G4` to `G6` were applications of permutations to subsets of the more complex structured colour set `MES`. The time used for the calculation of `G6` was much higher than the other times, because the involved tests were applications of permutations to the relative large marking of the place `Unused`, which had 81 tokens.

Implicit intersection

The Backtrack Method makes successive reductions of the initial search domain instead of explicit intersection. Explicit intersection would involve, for each place to calculate its set of self-symmetries in Σ_n. Thus, for each place, the initial search domain would be of size `n`!. Implicit intersection is a straightforward but very attractive approach, because it may produce large reductions of the search domain quickly, and thus speeds up the calculation.

In this example, the search domain was reduced two times, and each time with a considerable factor: For `n` = 10, the size of the initial search domain

was $10! = 3,628,800$. After the first calculation, it was cut down to the size of `G1` which was $9! = 362,880$, or 10 times lower than the initial search domain. Calculating `G2` reduced the search domain to size $4! \cdot 5! = 2,880$, or 126 times lower than the previous and $1,260$ times less than the initial.

Order of places

In the GAP code of Figure 3, the places were not treated in arbitrary order. For each step, i.e., each `Gi`, the place chosen to be treated, was the one that we intuitively thought would yield the largest and/or fastest reduction of the search domain. If the places were treated in an arbitrary order instead, a serious deterioration of performance could appear. E.g., if the places were treated in the reverse order of what was actually done, i.e., in order `Unused`, `Received`, `Sent`, `Inactive`, `Performing`, `Waiting`, the time for the calculation for `n` $= 10$ would be $66,860$ milliseconds. That is about 28 times as much as it was when the places were treated in a more sensible order. The lesson learned is that it is important to carefully devise a sensible order in which to treat the places.

Approximation methods

An approximation method uses necessary requirements for a permutation symmetry to be a self-symmetry for a given marking, to reduce a given search domain. I.e., an approximation method discards (many) permutation symmetries which are known not to be self-symmetries.

Any reachable marking of the data base manager system has the property that its set of self-symmetries can always be calculated using efficient approximation methods defined in [1]. This is because the reachable markings appearing in the places with the structured colour set `MES` always are sets on a certain form: All pairs in such a set have the same first component[6], namely the identity of the data base manager who initiated the update. It was proved in [1] (Corollary 8.23) that calculation of the set of self-symmetries for such sets can be done efficiently. This is not the case for sets containing general pairs, i.e., where the first and second components may take arbitrary values.

[6]Except for the place `Unused`, but it can safely be ignored from the calculations because of redundancy in the considered CP-net.

Even though an efficient approximation method may directly yield the set of self-symmetries, a general problem is to know that this is actually the case. If it is not known, computational resources will be wasted in trying to cut down on a set that cannot be reduced further. The Backtrack Algorithm is able to detect when no further reductions are possible very quickly, due to the fast tester property.

6 Experiment 2 — Cyclic Sets

In this section, we consider a supplementary experiment with the Backtrack Method and Backtrack Algorithm. This experiment constitutes a harder test than the experiment of Section 5.

We consider subsets of Cartesian product colour sets having a certain cyclic structure, which in [1] (p. 125) are identified as causing hard computational problems with respect to calculating their sets of self-symmetries. We introduce the subsets, show how the experiment is implemented in GAP, present performance measures, and discuss observations.

As opposed to the previous section, this section does not directly consider complete markings of CP-nets, but merely sets that can be thought of as arising when a complete marking is split up as prescribed by the Backtrack Method.

6.1 Experiment

Given Σ_{10} as the group of permutation symmetries, we will consider selected subsets of Cartesian products of $\{1,\ldots,10\}$, i.e., subsets of $\{1,\ldots,10\}^k$ for some $k \in \{1,\ldots,10\}$. As an example, for $k = 3$, the set considered will be $\{(1,2,3),(2,3,1),(3,1,2)\}$.

Given an exponent $k \in \{1,\ldots,10\}$ and the subset $s_k \subseteq \{1,\ldots,10\}^k$ defined by:

$$s_k = \{(1,2,\ldots,k-1,k),(2,3,\ldots,k,1),\ldots,(k,1,\ldots,k-1)\},$$

we want to find the set of self-symmetries for s_k, i.e., the subgroup[7]:

[7]The precise meaning of $\phi(s_k)$ is that $\phi \in [\{1,\ldots,10\}^k \to \{1,\ldots,10\}^k]$ defined by $\phi(i_1,\ldots,i_k) = (\phi(i_1),\ldots,\phi(i_k))$ is extended naturally to subsets of $\{1,\ldots,10\}^k$.

$\{\phi \in \Sigma_{10} | \phi(s_k) = s_k\}$.

The objective in the experiment is for different values of k to measure the time used for this calculation by the Backtrack Algorithm.

None of the approximation methods from [1] are able to produce useful results. E.g., one of the approximation methods applies projections: A self-symmetry for s_k must map the j'th projection of s_k to itself for all $j \in \{1, \ldots, k\}$. However, this is only a necessary requirement, not a sufficient one. All k projections are equal to the entire set $\{1, \ldots, k\}$, and thus only provide trivial information on the set of self-symmetries.

A self-symmetry for s_k, must indeed map the subset $\{1, 2, \ldots, k\}$ to itself, but is not allowed to make an arbitrary permutation of the elements. Only the k rotations $j \mapsto (j + l)\ mod\ k$, where $l \in \{0, \ldots, k-1\}$ is a constant, maps s_k to s_k. A self-symmetry for s_k is, of course, allowed to permute the elements of the set $\{k+1, \ldots, 10\}$, not appearing in s_k, arbitrarily. Thus, s_k only has $k \cdot (10-k)!$ self-symmetries, and the projection approximation method returned a set of size $k! \cdot (10-k)!$.

6.2 Implementation

An extract of the (trivial) GAP code implementing the experiment for $k = 10$ is shown in Figure 4. `selfsym_s10` is a user-defined function given as property argument to `SubgroupProperty`.

```
G := SubgroupProperty(S10, selfsym_s10);
```

Figure 4: GAP code — cyclic sets.

6.3 Statistics

Table 3 contains the statistics gathered in the experiment. In all cases, the initial search domain was Σ_{10}, which is of size $10! = 3,628,800$. The table has three columns, showing, respectively, k, the size of the set of self-symmetries for s_k, and finally the time used for the calculation.

k	Self-symmetries	Time
2	80,640	3,290
3	15,120	91,529
4	2,880	224,130
5	600	334,300
6	144	881,250
7	42	2,269,830
10	10	3,407,430

Table 3: Statistics — cyclic sets.

6.4 Observations

We now add two observations to the list from Section 5.4.

Complexity of structured colour sets

For larger values of k, the time to calculate the set of self-symmetries for s_k is high. From Table 3, it can be seen that for s_{10}, the calculation took close to one hour (remember, the measures of time are in milliseconds). Thus, if there are, say, 25 sets like s_{10} in some marking of a CP-net, calculation of the set of self-symmetries takes at least one day. A set of self-symmetries must be calculated each time a new marking is inserted in an OS-graph during construction. An OS-graph usually has many thousands of nodes. Thus, for large values of k, the computational complexity of the calculation of the set of self-symmetries for s_k may be too high to be practically usable.

Fortunately, in practice, sets like s_k appear in markings of CP-nets typically only for small values of k, say $k \leq 3$. If there are colour sets giving rise to, e.g., sets like s_{10}, it is unlikely that it is possible to analyse such CP-nets with OS-graphs anyway. Sets like s_2 are typical, e.g., over a Cartesian product colour set used to model packets with a sender and a receiver in a communication protocol. For small values of k, Table 3 showed that the sets of self-symmetries for s_k were calculated fast.

CP-nets allow other structured colour sets than Cartesian products. With respect to calculation of self-symmetries, two of these other structured colour sets may be treated similarly to Cartesian products. For record colour sets, this is obvious, as they only differ from Cartesian products because the en-

tries/fields are named. With respect to list colour sets, two lists with different lengths cannot be mapped to each other by a permutation symmetry, because of the way such a mapping is defined (in [7]). Therefore, a set of lists can be split into a number of sets, one set for each length appearing. These sets can then be treated as sets over a Cartesian product colour set. Generally in CP-nets, lists may grow arbitrary long. However, if a CP-net is to be analysed with OS-graphs, the user will typically have to enforce narrow restrictions on the lengths of the lists that can appear in reachable markings. Again, this will correspond to s_k's, for only small values of k.

Better than brute force

As noted above, the Backtrack Algorithm may be too slow to calculate the set of self-symmetries for s_k, for larger values of k. However, the algorithm is much faster than the obvious alternative, the brute force test of each permutation in the initial search domain.

With the Backtrack Algorithm, calculation of the set of self-symmetries for s_{10} took about one hour. The initial search domain contained more than 3.6 million permutations. If the brute force approach should be preferable, it requires that it is possible to test approximately $1,000$ permutations per second. Each test is an application of a permutation to the argument s_{10}, a set of size 10 consisting of tuples of size 10, and comparison of the result of the application with s_{10}. Using GAP, the time for such a single test was found to be 10 milliseconds. Hence, the brute force approach only tests approximately 100 permutations per second.

Thus, the Backtrack Algorithm is about 10 times faster than the brute force approach under some very unfavourable conditions. When the algorithm only has to find 10 permutations out of more than 3.6 million, it will rarely be able to make non-trivial reductions of the search domain, i.e., remove more than one permutation at a time. A non-trivial reduction is only possible each time a new self-symmetry is found, in this example at most 10 times.

7 Conclusions

In this paper, we have informally presented OS-graphs and argued that self-symmetries can be used to speed up their construction. The contribution is

the subsequent suggestion of the Backtrack Method for calculation of self-symmetries, and the justification of the suggestion. The justification had two parts: First, we described the important fast tester complexity property of the Backtrack Algorithm. Subsequently, we presented empirical tests of the performance of the Backtrack Method. The first experiment considered markings of the data base manager system for various numbers of managers. Very large groups of permutation symmetries were searched, and the sets of self-symmetries were calculated in a remarkably short time. Then, we exposed the Backtrack Method, or more precisely the Backtrack Algorithm, for certain sets with a cyclic structure, which constituted very hard tests. Under conditions often met in practice, the algorithm performed well. The two experiments gave rise to a list of observations of importance for calculation of self-symmetries.

The Backtrack Method is well-suited to be combined with approximation methods. A number of efficient approximation methods are defined in [1, 5]. The approximation methods can be used to reduce, often significantly, the initial search domain before the Backtrack Method is applied.

The Backtrack Method/Algorithm is only a heuristics. The performance is not always satisfactory. One problematic case was discussed in Section 6, but in more extreme cases, the Backtrack Algorithm performs even worse: If a marking only has the identity function as self-symmetry, the algorithm traverses the entire search domain without ever being able to use cosets for non-trivial reductions because the cosets remain trivial, i.e., of size 1. This corresponds to a brute force approach, and if the symmetry group is large, this is not practically feasible.

In spite of a bad worst case complexity, based on the observations described in this paper, we believe the Backtrack Method is a promising powerful vehicle for calculation of self-symmetries for markings of many CP-nets met in practice, and thus an aid for more efficient construction of OS-graphs.

Acknowledgements

Thanks to Jørgen Brandt and Afshin Foroughipour for help to understand the theory behind the Backtrack Algorithm (documented in [9]). Thanks to Rikke Drewsen Andersen, Søren Christensen, Kurt Jensen, and Lars Michael Kristensen for reading and commenting this paper.

The work has been supported by grants from University of Aarhus Re-

search Foundation and the Faculty of Science at University of Aarhus.

References

[1] R.D. Andersen, J.B. Jørgensen, and M. Pedersen. Occurrence Graphs with Equivalent Markings and Self-symmetries. Master's thesis, Computer Science Department, University of Aarhus, Denmark, 1991. Only available in Danish: Tilstandsgrafer med ækvivalente mærkninger og selvsymmetrier.

[2] G. Butler. *Fundamental Algorithms for Permutation Groups*, volume 559 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[3] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetries in Temporal Model Logic Model Checking. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification, Elounda, Greece*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[4] E.A. Emerson and A.P. Sistla. Symmetry and Model Checking. *Formal Methods of System Design, Vol. 9, 1/2*, 1996. Special Issue on Symmetry in Automatic Verification. Kluwer Academic Publishers.

[5] A. Foroughipour. Construction of OS-Graphs with Permutation Symmetries of a Coloured Petri Net Using Algebraic Algorithms. Master's thesis, Computer Science Department, University of Aarhus, Denmark, 1994.

[6] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1992.

[7] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Vol. 2, Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1994.

[8] K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods of System Design, Vol. 9, 1/2*, 1996. Special Issue on Symmetry in Automatic Verification. Kluwer Academic Publishers.

[9] J.B. Jørgensen. Calculation of Self-symmetries for Markings of Coloured Petri Nets Using a Backtrack Algorithm. Technical report, Computer Science Department, University of Aarhus, Denmark, 1994.

[10] J.B. Jørgensen and L.M. Kristensen. *Design/CPN OS-Graph Manual.* Computer Science Department, University of Aarhus, Denmark. Online: `http://www.daimi.aau.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. *Submitted to IEEE Transactions on Parallel and Distributed Systems*, 1996.

[12] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem — Its Structural Complexity.* Birkhauser Boston, 1993.

[13] M. Schönert. *GAP - Groups, Algorithm and Programming. A Reference Manual for GAP, Version 3.1.* Lehrstuhl für Mathematik, RWTH Aachen, Germany, 1992.