Computer Aided Verification of Lamport's Fast Mutual Exclusion Algorithm Using Coloured Petri Nets and Occurrence Graphs with Symmetries

Jens Bæk Jørgensen and Lars Michael Kristensen

Computer Science Department, University of Aarhus
Ny Munkegade, Bldg. 540
DK–8000 Aarhus C, Denmark
E-mail: {jbj,kris}@daimi.aau.dk

Abstract

In this paper, we present a new computer tool for verification of distributed systems. As an example, we establish the correctness of Lamport's Fast Mutual Exclusion Algorithm. The tool implements the method of occurrence graphs with symmetries (OS-graphs) for Coloured Petri Nets (CP-nets). The basic idea in the approach is to exploit the symmetries inherent in many distributed systems to construct a condensed state space. We demonstrate a significant increase in the number of states which can be analysed. The paper is to a large extent self-contained and does not assume any prior knowledge of CP-nets (or any other kinds of Petri Nets) or OS-graphs. CP-nets and OS-graphs are not our invention. Our contribution is development of the tool and verification of the example.

Index Terms: Modelling and Analysis of Distributed Systems, Formal Verification, Coloured Petri Nets, High-Level Petri Nets, Occurrence Graphs, State Spaces, Symmetries, Mutual Exclusion.

1 Introduction

Coloured Petri Nets (CP-nets) [1] is a language for modelling and analysis of distributed systems. The ideas behind CP-nets build upon those of ordinary Petri Nets (see, e.g., [2]) and those of Predicate/Transition Nets (see, e.g., [3]). CP-nets is at the same time theoretically well-founded and capable of modelling large distributed systems. A number of formal verification methods are available, by which the behaviour of a CP-net can be analysed. One of these methods is occurrence graphs (O-graphs) [4], also referred to as state spaces and reachability trees/graphs. The basic idea is to construct a directed graph with a node for each reachable state and an arc for each possible state change. An abundance of verification results can be derived from an O-graph. The method unfortunately suffers from the state explosion problem, which severely limits its practical usability. An approach to alleviate this problem is occurrence graphs with symmetries (OS-graphs) [4] [5], which are much more compact, but still enable us to obtain the same verification results as with O-graphs. Consequently, it is possible to investigate larger distributed systems, provided that they possess some kind of symmetry.

The applicability of OS-graphs is highly dependent on the existence of computer tools supporting the approach. Manual calculations of OS-graphs even for small systems are impossible. One contribution of this paper is to present our new computer tool supporting OS-graphs, and thereby developing the method from being theoretically promising to something which can be exploited in practice. Another contribution is the use of OS-graphs to establish the correctness of Lamport's Fast Mutual Exclusion Algorithm [6], in this paper referred to as **Lamport's Algorithm**.

Lamport's Algorithm is a mutual exclusion algorithm for shared-memory multiprocessors. A shared-memory multiprocessor is an architecture consisting of a number of CPUs connected to a common bus and with a single shared memory. It is assumed that the memory supports atomic read and write operations and that each process has a unique identifier, which is a positive integer. Fig. 1 depicts the code that process i executes in Lamport's Algorithm, when attempting to enter the critical section. The algorithm uses three global variables: x and y which are integers, and an array $b[1..N]$ of booleans, where N is the number of processes. The statement **await** *cond* represents a busy loop and can be seen as an abbreviation for **while** $\neg$*cond* **do** *skip*. Angle brackets are used to enclose the atomic statements, which are the reads and writes of x, y, and the entries of b. In this paper, we will

```
1  start:
2         <b[i] := true>;
3         <x := i>;
4         if <y ≠ 0> then
5                         <b[i] := false>;
6                         await <y = 0>;
7                         goto start;
8         fi;
9         <y := i>;
10        if <x ≠ i> then
11                        <b[i] := false>;
12                        for j := 1 to N
13                           do await <¬ b[j]> od;
14
15                        if <y ≠ i> then
16                                        await <y = 0>;
17                                        goto start;
18                        fi;
19        fi;
20
21        critical section;
22
23        <y := 0>;
24        <b[i] := false>;
```

Figure 1: Lamport's Algorithm.

not explain how Lamport's Algorithm works, because it is not important for our purpose. The curious reader is encouraged to consult [6].

The paper is organised as follows. In sect. 2, we present Coloured Petri Nets and create the model of Lamport's Algorithm to be used throughout the paper. In sect. 3, we introduce OS-graphs, and in sect. 4, the tool supporting OS-graphs is described. In sect. 5, we formulate correctness criteria for Lamport's Algorithm, and in sect. 6, we report on the use of the tool for the actual verification. Finally, in sect. 7, we draw some conclusions and discuss related and future work.

2 Coloured Petri Nets

In this section, we introduce **Coloured Petri Nets** (**CP-nets** or **CPN**). As we go along with the explanation of the basic concepts, we show how these can be used to model Lamport's Algorithm. Sect. 2.1 provides an informal introduction to CP-nets. Sect. 2.2 contains the formal definitions and may be skipped by readers already familiar with CP-nets. The complete CPN model of Lamport's Algorithm can be seen in fig. 2.

2.1 Informal Introduction to CP-nets

In contrast to many modelling languages, CP-nets is both state and action oriented. A state of a CP-net is represented by means of **places**. By convention, places are drawn as ellipses or circles with a name positioned inside. The basic idea in our CPN model is to describe the value of the program counters of the processes during the execution of Lamport's Algorithm. Therefore, fig. 2 has a place for each line in Lamport's Algorithm. A place is named according to the statement in that line. As an example, the place *setx_3* near the upper left corner of the drawing of the model (rotated 90 degrees in fig. 2) corresponds to the program counter being in a position, where the statement $< x := i >$ in line 3 is ready to be executed.

The global variables are also modelled by means of places. We have an accordingly named place for each of the variables x, y, and b. All places modelling variables are grayed in order to distinguish them from the places modelling the program counters. The graying has no formal meaning. It should be noted that in fig. 2, there are three places named y. These are conceptually the same place, but have been drawn as three copies in order to reduce the number of crossing arcs and thereby improve the legibility of the CPN model. A similar remark applies to the four places named b.

Each place in a CP-net has a **colour set** (a type[1]), which determines the kind of data the place may contain. An element of a colour set is called a **colour**. By convention, the colour set is written in italics next to the lower right corner of the place. From fig. 2, it can be seen that the place b has the colour set $PID \times BOOL$, and that the places x and y have the colour set PID_0N. The places *wait* and *done* have colour set $PID \times PID$. All

[1]An alternative and perhaps better name for Coloured Petri Nets might be "Typed Petri Nets". However, the term "coloured" has a historical explanation, and it has stuck. "Colour set" and "type" are used as synonyms in this paper.

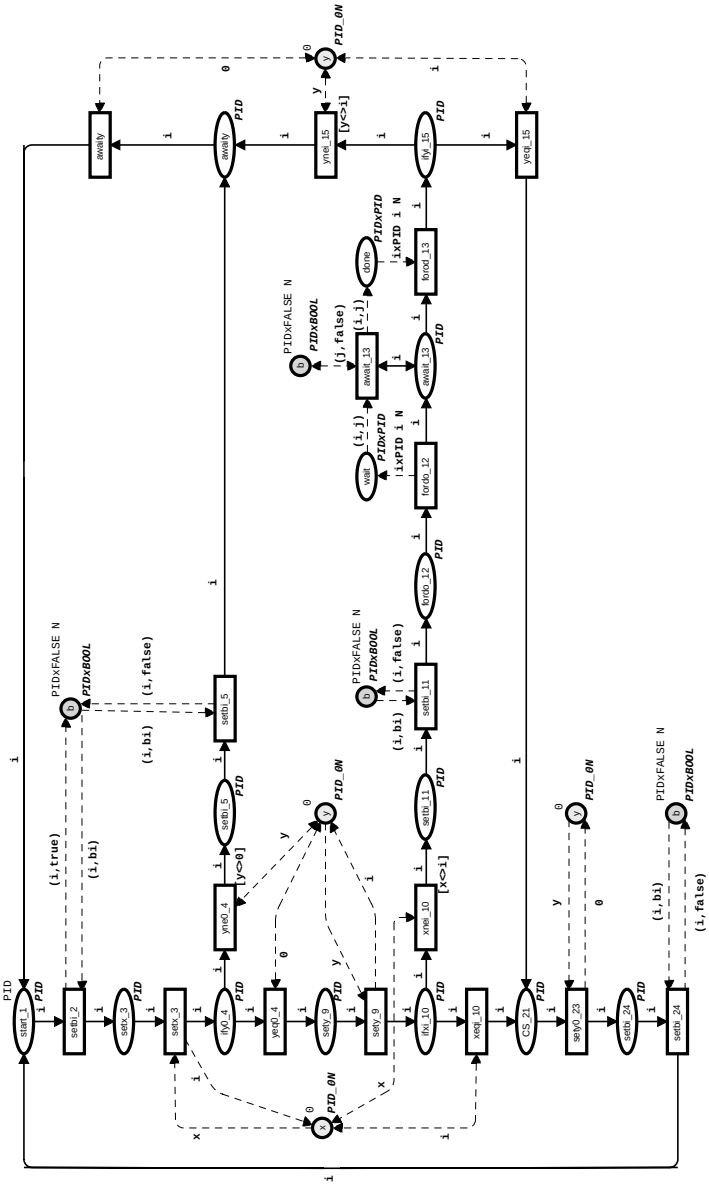

Figure 2: The CPN model of Lamport's Algorithm.

other places have colour set PID. PID stands for Process IDentifier. The definition of the colour sets are as follows:

$$PID\_0N = \{0, 1, \ldots, N\}$$
$$BOOL = \{true, false\}$$
$$PID = PID\_0N \setminus \{0\}$$
$$PID \times BOOL = \{(i, bi) | i \in PID \wedge bi \in BOOL\}$$
$$PID \times PID = \{(i, j) | i, j \in PID\}.$$

Thus, the place b can contain pairs consisting of an integer and a boolean. The places x and y can contain integers from 0 to N, and the places *wait* and *done* can contain pairs of integers from 1 to N. All other places can contain integers from 1 to N. The value 0 is special. It is used to signal when the values of the shared variables x and y do not correspond to any of the processes.

A state of a CP-net is called a **marking**. A marking describes how **tokens** are distributed on the individual places. A token is a value, which is a member of the colour set of the corresponding place. The initial marking of a place is specified in the CPN model, by convention, next to the upper right corner of the place. The initial marking of the place $start\_1$ is PID, i.e., the tokens from 1 to N. This models that to begin with, the program counters of all processes are positioned at the start label. For each of the places x, y, and b, the initial marking describes the start value of the corresponding variable. Both x and y are equal to 0 initially. The initial marking of the b-place is determined by the expression $PID \times FALSE\ N$, which evaluates to a set of tokens modelling that all entries $b[i]$ are $false$ for $1 \leq i \leq N$. Initially, all other places are empty,

Besides from having different tokens on a place, it is also possible to have several tokens with the same colour. Therefore, the marking of a place is in general a multi-set[2]. A number of operations such as addition and scalar-multiplication are defined for multi-sets, and we will apply them freely is this paper. For details, see [1].

The actions of a CP-net are represented by **transitions**, which, by convention, are drawn as rectangles. Transitions and places are connected by **arcs**. In fig. 2, solid arcs are used for control flow and dashed arcs are used

[2]A multi-set is often referred to as a bag. Sets can be considered a special kind of multi-sets, and therefore, in this paper, we sometimes use a set-like notation for multi-sets.

for data manipulation. The graphical appearance of an arc has no formal meaning. The two kinds of arcs are only used to make a more clear presentation.

A transition removes tokens from the places connected to incoming arcs (input places) and adds tokens to the places connected to outgoing arcs (output places). The tokens to be removed from input places and added to output places are determined by the **arc expressions**, which are positioned next to the arcs.

In Lamport's Algorithm, the actions are execution of statements. Therefore, we have associated an accordingly named transition with each statement. E.g., the transition *setbi_2* (see fig. 3) models the execution of the statement $b[i] := true$ in line 2 of fig. 1.

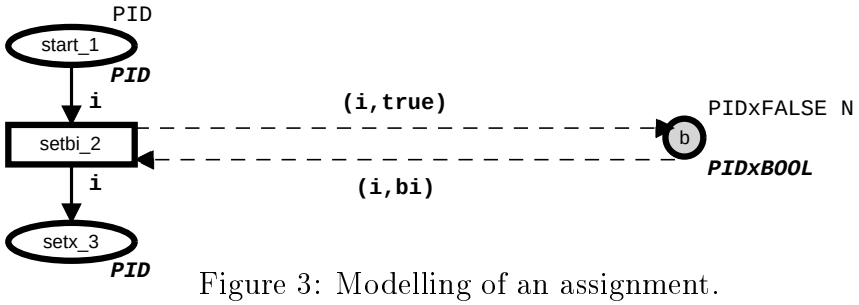

Figure 3: Modelling of an assignment.

The transition has two incoming arcs and two outgoing arcs. The arc expressions of the incoming arcs are i and (i, bi), where i and bi are variables of type PID and $BOOL$, respectively. To talk about an **occurrence** of the transition *setbi_2*, the variable i has to be bound to a value from PID, and bi has to be bound to a value from $BOOL$, in order to evaluate the arc expressions. A pair consisting of a transition and a binding of the variables of its surrounding arcs is called a **binding element**. A binding element may occur, iff the tokens to be removed exist on the respective input places.

Assume now that we bind the variable i to 1 and bi to $false$. Then, the expression on the incoming arc from *start_1* will evaluate to 1, and the expression on the incoming arc from b will evaluate to $(1, false)$. Since in the initial marking, denoted M_0, a 1-token is on *start_1*, and a $(1, false)$-token is on b, the described binding element, denoted $(setbi\_2, <i = 1, bi = false>)$, may occur. The binding element is said to be **enabled** in M_0. Several

binding elements may be enabled in the same marking. E.g., the binding element $(setbi\_2, < i = 2, bi = false >)$ is also enabled in M_0. The two binding elements may occur in the same **step**, since in M_0, they do not share any of the tokens on the input places. The two binding elements are said to be **concurrently enabled**. This corresponds to processes 1 and 2 being able to do this assignment independently of each other.

An occurrence of the binding element $(setbi\_2, < i = 1, bi = false >)$ will remove the 1-token from $start\_1$ and, similarly, remove the $(1, false)$-token from b. As determined by the arc expressions of the outgoing arcs, a 1-token will be added to $setx\_3$, and a $(1, true)$-token will be added to b. An occurrence of this binding element corresponds to process 1 executing the statement $<b[i] := true>$ in line 2 of fig. 1. In this way, an occurrence of a transition models the execution of an atomic statement in Lamport's Algorithm. All other assignments in Lamport's Algorithm are modelled in a similar fashion.

We will now describe how to model the other statements in Lamport's Algorithm, i.e., the **if**-, **await**-, **for**-, and **goto**-statements. Consider the **if**-statement starting in line 4 of Lamport's Algorithm. This statement is modelled by the part of the CPN model shown in fig. 4.

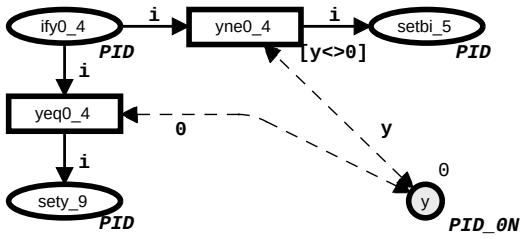

Figure 4: Modelling of an **if**-statement.

The condition $y \neq 0$ evaluates to true or false, and depending on this, one of the two branches in Lamport's Algorithm is chosen. The case where the condition is false is modelled by the transition $yeq0\_4$. It has two incoming arcs, one from the place $ify0\_4$ and one from the place[3] y. The arc expression on the arc from y is 0 and will evaluate to 0, independent of the binding of the variable i, i.e., the process executing the **if**-statement. Thus, the transition

[3]A double arc is a shorthand for two arcs with the same arc expression, one arc in each direction.

will only be enabled when y contains a 0-token, corresponding to y being 0 in Lamport's Algorithm. When the transition occurs, it puts the 0-token back on y and puts an i-token on the place $sety\_9$. The transition $yne0\_4$ models the case in which the condition $y \neq 0$ is true. The transition has two incoming arcs with arc expressions i and y, respectively. Associated with the transition is also a **guard**. Guards are, by convention, put in brackets and located next to the lower right corner of the transition. A guard is a boolean expression, which imposes an additional condition on enabling. The variables must be bound so that the guard evaluates to true. In this case, the boolean expression is $y <> 0$. The transition is therefore only enabled, when y is not bound to 0. The two **if**-statements starting in lines 10 and 15 are modelled in a similar fashion.

We now turn to the modelling of the **await**-statement in line 6. The **await**-statement is modelled by the part of the CPN model shown in fig. 5.

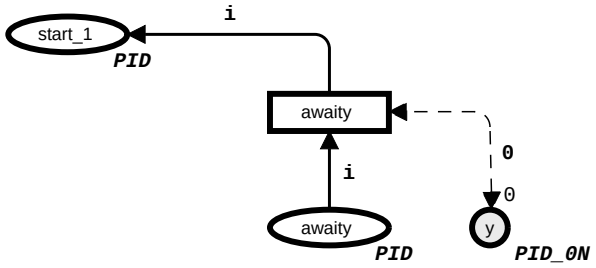

Figure 5: Modelling of an **await**-statement.

The transition has an incoming arc from *awaity* and from y. The arc expression from y evaluates to 0 independent of the binding of i on the arc from *awaity*. Thus, the transition is only enabled when y contains a 0-token, which corresponds to y being 0 in Lamport's Algorithm.

The **goto**-statements are modelled implicitly. Consider, e.g., the **goto**-statement immediately after the **await**-statement in line 7. In the model, we have drawn an arc from the transition modelling the execution of the **await**-statement to the place $start\_1$.

Finally, we consider the **for**-statement starting in line 12. It is modelled by the part of the CPN model shown in fig. 6. For reasons to become clear later (in sect. 6), we model a more general form of the **for**-statement. In Lamport's Algorithm, the **for**-statement is used to test each of the entries

in the b-array in turn starting from $b[1]$. In the model, we do not impose an order in which the entries are tested.

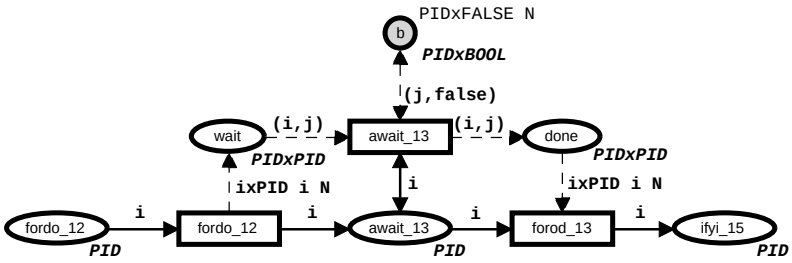

Figure 6: Modelling of a **for**-statement.

When process i enters the **for**-statement by occurrence of the transition $fordo\_12$, the multi-set denoted $i \times PID\ i\ N = \{(i,j)|j \in PID\}$ is put on the place $wait$, which contains the entries in the b-array that process i still needs to test. The transition $await\_13$ models the execution of the **await**-statement inside the **for**-statement, and is only enabled when a $(j, false)$-token is present on the b-place. An occurrence of that transition will remove an (i,j)-token from $wait$ and add it to the place $done$, which contains the entries in the b-array that process i has already tested. Process i leaves the **for**-statement, when the transition $forod\_13$ occurs. As it can be seen, this transition is only enabled, when place $done$ contains the multi-set $i \times PID\ i\ N$, i.e., when all the entries in the b-array have been tested.

We have now explained how to model all the basic constructs of Lamport's Algorithm. The creation of the complete model just consists in putting all the pieces together. The process might even be automated. No ingenuity is required — nor desired. This systematic strategy reduces the probability of accidental errors, and thus makes it unlikely that the constructed CP-net is not a proper model of the algorithm. Lamport's Algorithm is modelled in a similar way in [7].

2.2 Formal Definition of CP-nets

We now give a formal definition of CP-nets and their behaviour. The purpose of this section is twofold. First of all, to clear out any ambiguity that might be in the informal introduction to CP-nets in the previous section, and second, to

fix the notation to be used in this paper. The definitions and notation closely follow [1] and readers familiar with that reference may skip this section.

Structure of CP-nets

Before giving the formal definition of a CP-net, we fix some notation and terminology. The term **net expressions** refers to the expressions describing colour sets, initial markings, arc expressions, and guards. Related to net expressions, we introduce the following notation:

- $Type(expr)$ denotes the type of an expression $expr$.
- $Var(expr)$ denotes the set of variables in an expression $expr$.
- $Type(v)$ denotes the type of a variable v.
- $Type(vars)$, where $vars$ is a set of variables, denotes the set of types $\{Type(v)|v \in vars\}$.
- S_{MS} denotes the set of multi-sets over a set S.
- $Bool$ denotes the set of booleans, i.e., $Bool = \{true, false\}$.

We now formally define CP-nets. Explanation follows the definition.

Definition 1 *A* **CP-net** *is a tuple* $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ *satisfying the requirements below:*

1. Σ *is a finite set of non-empty types, called* **colour sets**.
2. P *is a finite set of* **places**.
3. T *is a finite set of* **transitions**.
4. A *is a finite set of* **arcs** *such that* $P \cap T = P \cap A = T \cap A = \emptyset$.
5. N *is a* **node** *function. It is defined from* A *into* $P \times T \cup T \times P$.
6. C *is a* **colour** *function. It is defined from* P *into* Σ.
7. G *is a* **guard** *function. It is defined from* T *into expressions such that:* $\forall t \in T : [Type(G(t)) = Bool \wedge Type(Var(G(t))) \subseteq \Sigma]$.

8. *E is an* **arc expression** *function. It is defined from A into expressions such that:*

   $\forall a \in A : [Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$,

   where $p(a)$ is the place of $N(a)$.

9. *I is an* **initialisation** *function. It is defined from P into expressions without free variables such that:*

   $\forall p \in P : [Type(I(p)) = C(p)_{MS}]$. □

Item 1 determines the set of colour sets and hence the colours which can be referred to in the net expressions. In the CPN model of Lamport's Algorithm, $\Sigma = \{PID\_0N, PID, BOOL, PID \times BOOL, PID \times PID\}$. Items 2, 3, and 4 specify the places, transitions, and arcs. Item 5, the node function, determines the source and destination of arcs. Note that an arc always connects a place and a transition. Item 6, the colour function, associates a colour set with each place. In the CPN model of Lamport's Algorithm, the colour function maps the place b into $PID \times BOOL$, the places x and y into PID_0N, the places *wait* and *done* into $PID \times PID$, and all other places into PID. Item 7, the guard function, ensures that guards are expressions which evaluate to a boolean, and that the types of the variables in the guards are in Σ. Likewise, items 8 and 9, the arc expression function and the initialisation function, ensure similar, appropriate type constraints.

In the rest of this paper, we will assume that a CP-net CPN is given, $CPN = (\Sigma, P, T, A, N, C, G, E, I)$.

Normally, a CP-net is created in terms of a CPN diagram, i.e., a graphical representation as in fig. 2, and not by specifying a 9-tuple as in def. 1. Fig. 2 is created using the tool Design/CPN [8], which supports construction and analysis of CP-nets. For declarations of colour sets, variables, and functions; and for net expressions, this tool uses CPN ML, which is an extension of the functional programming language Standard ML (SML) (see, e.g., [9]). The declarations for the CPN model of Lamport's Algorithm can be seen in fig. 7.

In line 2, the number of processes N is specified. In this case, $N = 3$. Lines 8-12 declare the colour sets. Lines 15-17 declare the variables and their type. Finally, the function $PID \times FALSE$ used to specify the initial marking on the place b, and the function $i \times PID$ used in the modelling of the **for**-statement are declared. Both are typical SML-style recursive functions.

```
1  (* Number of processes - in this case 3 *)
2  val N = 3;
3
4  (* non-zero predicate *)
5  fun nonzero i = (i<>0);
6
7  (* Declaration of the colour sets *)
8  color PID_0N = int with 0..N declare ms;
9  color BOOL = bool;
10 color PID = subset PID_0N by nonzero declare ms;
11 color PIDxPID = product PID * PID;
12 color PIDxBOOL = product PID * BOOL;
13
14 (* Declaration of the variables *)
15 var x,y : PID_0N;
16 var i,j : PID;
17 var bi : BOOL;
18
19 (* Function used to specify the initial marking on b *)
20 fun PIDxFALSE 0 = empty
21  |  PIDxFALSE i = 1`(i,false)+(PIDxFALSE (i-1));
22
23 (* Function used in the for-statement *)
24 fun ixPID i 0 = empty
25  |  ixPID i j = 1`(i,j)+(ixPID i (j-1));
```

Figure 7: Declarations for the CPN model of Lamport's Algorithm.

Behaviour of CP-nets

We now turn to the formal definition of behaviour of CP-nets. First, we fix some more notation.

- $Var(t)$, for a transition $t \in T$, denotes the set of variables of t present in either the guard G(t) or in an arc expression of one of the surrounding arcs denoted A(t). Formally:

  $Var(t) = \{v | v \in Var(G(t)) \vee \exists a \in A(t) : v \in Var(E(a))\}$.

- $A(x_1, x_2)$ for $(x_1, x_2) \in P \times T \cup T \times P$ denotes the set of connecting arcs. Formally:

  $A(x_1, x_2) = \{a \in A | N(a) = (x_1, x_2)\}$.

As a consequence, if x_1 and x_2 are not connected, $A(x_1, x_2) = \emptyset$.

- $E(x_1, x_2)$ for $(x_1, x_2) \in P \times T \cup T \times P$ denotes the expression of (x_1, x_2). Formally:

  $E(x_1, x_2) = \sum_{a \in A(x_1, x_2)} E(a)$.

It should be noted that the sum in the definition of $E(x_1, x_2)$ is well-defined because of item 8 in def. 1, which ensures that all terms in the sum are of the same multi-set type. Having fixed the notation, we define the concept of a binding. *expr* <*b*> denotes the result of evaluating an expression *expr*, whose variables are bound to values as determined by *b*.

Definition 2 *A* **binding** *of a transition $t \in T$ is a function b defined on $Var(t)$ such that:*

1. $\forall v \in Var(t) : b(v) \in Type(v)$.
2. $G(t)$<b>.

$B(t)$ denotes the set of all bindings for t. □

Item 1 ensures that only values of the correct type can be bound to a variable. Item 2 expresses that in order for b to be a binding of t, the guard must evaluate to true in b. In the following, bindings will be written on the form $<v_1 = c_1, v_2 = c_2, \ldots v_n = c_n>$, when $Var(t) = \{v_1, v_2, \ldots v_n\}$. Now, we formally define markings, binding elements, and steps.

Definition 3 *A* **marking** *M is a function defined on P such that $M(p) \in C(p)_{MS}$ for all $p \in P$. The set of all markings is denoted $\mathbb{M}$. The initial marking is denoted M_0.*

A **binding element** *is a pair (t, b), where $t \in T$ and $b \in B(t)$. The set of all binding elements is denoted BE, while the set of binding elements for a specific transition $t \in T$ is denoted $BE(t)$.*

A **step** *is a non-empty and finite multi-set over BE. The set of all steps is denoted $\mathbb{Y}$.* □

By defining a step as a multi-set of binding elements, we allow multiple occurrences of a binding element in a given step. We now give the formal definition of enabling.

Definition 4 *A step* $Y \in \mathbb{Y}$ *is* **enabled** *in a marking* $M \in \mathbb{M}$*, iff the following property is satisfied:*

$\forall p \in P : \sum_{(t,b)\in Y} E(p,t)\langle b\rangle \leq M(p)$.

$M[Y\rangle$ *denotes that* Y *is enabled in* M. □

The definition states that each binding element $(t, b) \in Y$ must be able to get the tokens specified by $E(p, t)\langle b\rangle$ — which is the multi-set of tokens removed from p, when t occurs with the binding b — without having to share these with other binding elements in Y. The summation is a multi-set sum, i.e., if (t, b) appears in Y multiple times, this multiplicity is taken into account in the sum. If a binding element for a transition t is included in an enabled step in a marking M, we will say that t is enabled in M.

When a step Y is enabled, it may occur. When Y occurs, it removes tokens from the input places and adds tokens to the output places of the included transitions, according to the following definition, which also introduces the concepts of occurrence sequences and reachability.

Definition 5 *When a step* Y *is enabled in a marking* M_1*, it may* **occur**, *changing the marking* M_1 *to another marking* M_2 *defined by:*

$\forall p \in P : M_2(p) = \left(M_1(p) - \sum_{(t,b)\in Y} E(p,t)\langle b\rangle\right) + \sum_{(t,b)\in Y} E(t,p)\langle b\rangle$.

In this case, we say that M_2 *is* **directly reachable** *from* M_1 *by the occurrence of the step* Y*, which we denote* $M_1[Y > M_2$.

A **finite occurrence sequence** *is a sequence of markings and steps:*

$M_1[Y_1 > M_2[Y_2 > M_3 \ldots M_n[Y_n > M_{n+1}$

such that[4] $n \in \mathbb{N}$ *and* $M_i[Y_i > M_{i+1}$ *for* $i = 1, \ldots, n$.

Analogously, an **infinite occurrence sequence** *is a sequence of markings and steps:*

$M_1[Y_1 > M_2[Y_2 > M_3 \ldots$

such that $M_i[Y_i > M_{i+1}$ *for* $i = 1, 2, \ldots$.

[4] $\mathbb{N} = \{0, 1, 2, \ldots\}$ denotes the set of non-negative integers.

A marking M' is **reachable** *from a marking M, iff there exists a sequence of steps $Y_1, Y_2, \ldots Y_n$ such that:*

$$M[Y_1 > M_2[Y_2 > M_3 \ldots M_n[Y_n > M'.$$

The set of markings which are reachable from M is denoted $[M>$. □

If a binding element for a transition t is included in a step Y, which occurs in a marking M, we will say that t occurs in M.

Quite often, the purpose of creating a CP-net is to investigate whether certain dynamic properties hold. An example of such a property is the existence of dead markings, corresponding to deadlocks of a considered system. In sect. 5.2, we formally define a number of dynamic properties for CP-nets and use them to verify Lamport's Algorithm.

3 Occurrence Graphs with Symmetries

This section introduces the verification method of occurrence graphs with symmetries, which we are going to use to establish correctness of Lamport's Algorithm. The section is structured as follows. Sect. 3.1 briefly sums up the concept of full occurrence graphs (O-graphs). In sect. 3.2, occurrence graphs with symmetries (OS-graphs) are described in an informal way. OS-graphs are formally defined in sect. 3.3, which may be skipped by readers familiar with [4].

3.1 O-Graphs

One of the classical verification methods for CP-nets employs occurrence graphs. In its simplest form, an occurrence graph for a CP-net is a directed graph with a node for each reachable marking and an arc for each occurring binding element. This kind of graphs are called full occurrence graphs or **O-graphs**. Except for concurrency properties[5], all dynamic properties for a CP-net[6] can be derived from its O-graph — in particular, the properties to be used for the verification of Lamport's Algorithm.

[5]When working with O-graphs, we only consider steps consisting of one single binding element.

[6]Only CP-nets with a finite number of reachable markings are considered.

As mentioned in sect. 1, a serious drawback of the occurrence graph method is that it suffers from the state explosion problem: Even for relatively small CP-nets, the occurrence graphs are often so large that they cannot be constructed in practice given the computer technology presently available. Alleviation of this inherent complexity problem is a major challenge of research. Several theoretical methods have been proposed. Among them are OS-graphs. They are defined in [4]. The main ideas will be repeated here.

3.2 Informal Introduction to OS-graphs

Lamport's Algorithm treats all processes in the same way. The processes are symmetric in a sense to be illustrated in the following. In the CPN model for $N = 3$, consider the two markings M_1 and M_2 shown below. Multi-sets are written in the notation from [1]: As a sum using the symbol "+", where the number of appearances of each element is the coefficient preceeding the symbol ‘ (pronounced "back quote" or "of").

$$
\begin{array}{lcl}
M_1(setx\_3) & = & 1‘1 \\
M_1(start\_1) & = & 1‘2 + 1‘3 \\
M_1(b) & = & 1‘(1, true) + 1‘(2, false) + 1‘(3, false) \\
M_1(x) & = & 1‘0 \\
M_1(y) & = & 1‘0
\end{array}
$$

$$
\begin{array}{lcl}
M_2(setx\_3) & = & 1‘2 \\
M_2(start\_1) & = & 1‘1 + 1‘3 \\
M_2(b) & = & 1‘(1, false) + 1‘(2, true) + 1‘(3, false) \\
M_2(x) & = & 1‘0 \\
M_2(y) & = & 1‘0.
\end{array}
$$

For all other places p, $M_1(p) = M_2(p) = empty$, where $empty$ denotes the empty multi-set. In both markings, all processes but one are on the place $start\_1$. The remaining one is on the place $setx\_3$. The two markings differ by which process is on $setx\_3$. In M_k, the marking of $setx\_3$ is k for $k = 1, 2$.

M_1 and M_2 are symmetric, in the sense that one can be obtained from the other by interchanging the colours 1 and 2. The crucial observation about symmetric markings is that they describe states of the system that are similar: If we know the possible behaviours of the system starting from

M_1, then we do not need to explore the possible behaviours from M_2. An indication of this is to consider the set of binding elements BE_k, which are enabled in M_k, for $k = 1, 2$:

$$\begin{aligned} BE_1 &= \{(setbi\_2, < i = 2, bi = false >), \\ &\quad (setbi\_2, < i = 3, bi = false >), (setx\_3, < i = 1, x = 0 >)\} \end{aligned}$$

$$\begin{aligned} BE_2 &= \{(setbi\_2, < i = 1, bi = false >), \\ &\quad (setbi\_2, < i = 3, bi = false >), (setx\_3, < i = 2, x = 0 >)\}. \end{aligned}$$

BE_1 is symmetric to BE_2, i.e., BE_2 can be obtained from BE_1 by interchanging 1 and 2. Now, consider the marking $M_1^{'}$ reached when, e.g., the binding element $(setx\_3, < i = 1, x = 0 >)$ occurs in M_1; and the marking $M_2^{'}$ reached when the binding element $(setx\_3, < i = 2, x = 0 >)$ occurs in M_2. $M_1^{'}$ is identical to M_1, and $M_2^{'}$ is identical to M_2, except for the places listed below:

$$\begin{aligned} M_1^{'}(x) &= 1`1 \\ M_1^{'}(setx\_3) &= empty \\ M_1^{'}(ify0\_4) &= 1`1 \end{aligned}$$

$$\begin{aligned} M_2^{'}(x) &= 1`2 \\ M_2^{'}(setx\_3) &= empty \\ M_2^{'}(ify0\_4) &= 1`2. \end{aligned}$$

It is easy to see that $M_1^{'}$ and $M_2^{'}$ are symmetric, i.e., that $M_1^{'}$ can be obtained from $M_2^{'}$ by interchanging 1 and 2.

The property illustrated above is that symmetric markings have symmetric sets of enabled binding elements, and symmetric sets of directly reachable markings. Using induction, this property can be expanded to finite and infinite occurrence sequences.

The CPN model of Lamport's Algorithm contains many markings that are symmetric in this way. The basic idea in OS-graphs is to lump together symmetric markings and symmetric binding elements.

Definition of an **OS-graph** for a CP-net requires the presence of two equivalence relations — one on the set of markings and one on the set of binding elements. The OS-graph has a node for each reachable equivalence

class of markings[7]. The OS-graph has an arc between two nodes, iff 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. Typically an OS-graph is much smaller than the corresponding O-graph, but always contains as much information.

The two equivalence relations are induced by an algebraic group of functions called **permutation symmetries**. A permutation symmetry maps markings to markings and binding elements to binding elements. Two markings are **equivalent** (or **symmetric**), iff there exists a permutation symmetry mapping one of the markings to the other. Similarly for binding elements[8].

The user defines the group of permutation symmetries by writing a **permutation symmetry specification**. A permutation symmetry specification assigns a **symmetry group** to each **atomic** colour set appearing in the CP-net. A colour set defined without reference to other colour sets is atomic. In the CPN model of Lamport's Algorithm, there are two atomic colour sets: PID_0N and $BOOL$. A symmetry group determines how the colours of an atomic colour set are allowed to be permuted. E.g., a symmetry group may specify that all colours can be permuted arbitrarily, or that they must all be fixed, i.e., cannot be changed. Many intermediate forms exists, e.g., all rotations of a finite, ordered colour set.

A permutation symmetry specification for the CPN model of Lamport's Algorithm capturing that processes corresponding to the integers in the set $\{1, ..., N\}$ behave in a symmetric way, and that the integer 0 is a special value used for initialisation purposes, can be described as follows: We assign the symmetry group to PID_0N, that allows arbitrary permutations in the set $\{1, .., N\}$, and insists that 0 is fixed. This symmetry group has $N!$ elements. $BOOL$ is assigned the singleton symmetry group consisting of the identity function id only. Thus, the values $true$ and $false$ cannot be swapped. They are (of course) fundamentally different.

A **structured** colour set is one, which is not atomic. The symmetry

[7]A reachable equivalence class is one, which contains a reachable marking. As we shall see, for two equivalent markings, either both of them are reachable or none of them are reachable.

[8]A permutation symmetry can also be used to map colours to colours. We will speak about two colours being equivalent (or symmetric), iff there exists a permutation symmetry mapping one to the other.

group for a structured colour set is inherited from the symmetry groups of its **base colour sets**, i.e., the colour sets that it is built from. In the CPN model of Lamport's Algorithm, there are three structured colour sets: PID, $PID \times BOOL$, and $PID \times PID$. PID inherits its symmetry group from its base colour set PID_0N. An element of the symmetry group for PID_0N induces a permutation on PID. Likewise, $PID \times BOOL$ inherits its symmetry group from the symmetry groups of PID and $BOOL$: An element of the symmetry group of $PID \times BOOL$ is a pair, where the first element is a member of the symmetry group of PID, and the second element is a member of the symmetry group of $BOOL$. $PID \times PID$ inherits its symmetry group from the symmetry group of PID: An element of the symmetry group of $PID \times PID$ is a pair, where the first and the second element are identical members of the symmetry group of PID.

The purpose of a permutation symmetry specification is to capture inherent symmetries of the model. A permutation symmetry specification in accordance with the model, in a way to be defined precisely in sect. 3.3, is said to be **consistent**. As we will see, the permutation symmetry specification described above for the CPN model of Lamport's Algorithm is consistent. But if we, e.g., assigned a symmetry group to PID_0N that allowed arbitrary permutations in the set $\{0, 1, \ldots, N\}$, and, hence, had not insisted that 0 should stay fixed, the resulting permutation symmetry specification would not be consistent. To see this, consider, e.g., the transition *awaity* in fig. 5. A necessary requirement for this transition to be enabled, is that the place y contains a 0-token. Thus, if we allowed to swap 0 with another colour, we could obtain two symmetric markings, where *awaity* was enabled in one of them, but not in the other. These two marking would not contain the same information, and it would be wrong to consider them symmetric. Consequently, a consistency requirement is crucial.

3.3 Formal Definition of OS-graphs

In this section, we introduce the concepts necessary to formally define OS-graphs. All definitions and propositions are taken from [4] and are included here to make this paper self-contained. Readers familiar with [4] may skip this section. First the basics.

Definition 6 *A* **permutation symmetry specification** *is a function SG that maps each atomic colour set $S \in \Sigma$ into a subgroup $SG(S)$ of the set of*

permutations of S. $SG(S)$ is called the **symmetry group** *of S.*

A **permutation symmetry** *for SG is a function ϕ that maps each atomic colour set $S \in \Sigma$ into a permutation $\phi_s \in SG(S)$. The set of all permutation symmetries for SG is denoted Φ_{SG}.* □

The permutation symmetry specification SG_L for the CPN model of Lamport's Algorithm, informally described in sect. 3.2, is formally defined below. $PERM(I)$ is the set of all permutations of a finite set I.

$SG_L(PID\_0N) = \{\phi \in PERM\{0, ..., N\} | \phi(0) = 0\}$
$SG_L(BOOL) = \{id\}$.

An example of a permutation symmetry $\phi \in \Phi_{SG_L}$ is the following, where the function $(l\ k)_I$ swaps the values k and l in the set I:

$\phi : PID\_0N \mapsto (1\ 2)_{\{0,...,N\}}$
$\phi : BOOL \mapsto \{id\}$.

ϕ induces the following mappings on the structured colour sets:

$\phi : PID \mapsto (1\ 2)_{\{1,...,N\}}$
$\phi : PID \times BOOL \mapsto ((1\ 2)_{\{1,...,N\}}, id)$
$\phi : PID \times PID \mapsto ((1\ 2)_{\{1,...,N\}}, (1\ 2)_{\{1,...,N\}})$.

As mentioned in sect. 3.2, each permutation symmetry $\phi \in \Phi_{SG}$ induces a function which maps markings into markings. $\phi(M)$ is simply a substitution of each colour (value) $v \in S$, where S is some colour set, by $\phi(S)(v)$. A function mapping binding elements to binding elements is induced similarly. E.g., consider the markings and binding elements used in the example from sect. 3.2. $\phi \in \Phi_{SG_L}$ defined above maps M_1 to M_2. Moreover, ϕ maps the binding element $(setx\_3, < i = 1, x = 0 >)$ to the binding element $(setx\_3, < i = 2, x = 0 >)$, and ϕ also maps M_1' to M_2'.

Def. 7 formally defines consistency of a permutation symmetry specification. The transition of a given arc a is denoted $t(a)$.

Definition 7 *A permutation symmetry specification SG is* **consistent**, *iff the following properties are satisfied for all $\phi \in \Phi_{SG}$, all $t \in T$, and all $a \in A$:*

1. $\phi(M_0) = M_0$.

2. $\forall b \in B(t) : \phi(b) \in B(t)$.

3. $\forall b \in B(t(a)) : E(a) <\phi(b)>= \phi(E(a) <b>)$. □

Item 1 ensures that each permutation symmetry maps the initial marking to itself. Item 2 says that each permutation symmetry must map binding elements into binding elements. In particular, this means that no transition is allowed to have an asymmetric guard, i.e., a guard that treats two symmetric colours differently. Item 3 states that arc expressions and permutation symmetries must commute. Thus, asymmetric arc expressions are ruled out. It is important to notice that all three properties are local and structural. They can be checked without considering occurrence sequences.

When a consistent permutation symmetry specification is given, the important dynamic property proved in [4] and stated in the next proposition holds. It formalises that symmetric markings have symmetric sets of enabled binding elements, and symmetric sets of directly reachable markings, as illustrated in sect. 3.2. Thus, the proposition justifies that it is sufficient to explore the possible behaviours of the system for one marking of each equivalence class.

Proposition 1 *A consistent permutation symmetry specification SG satisfies the following property:*

$$\forall M_1, M_2 \in \mathbb{M}\ \forall b \in BE\ \forall \phi \in \Phi_{SG} : M_1[b>M_2 \Leftrightarrow \phi(M_1)[\phi(b)>\phi(M_2).$$ □

We now formally define the two equivalence relations that are derived from the group of permutation symmetries, determined by a permutation symmetry specification SG.

Definition 8 *The relation* $\approx_{\mathbb{M}} \subseteq \mathbb{M} \times \mathbb{M}$ *is defined by:*

$$M \approx_{\mathbb{M}} M^* \Leftrightarrow \exists \phi \in \Phi_{SG} : M = \phi(M^*).$$

The relation $\approx_{BE} \subseteq BE \times BE$ *is defined by:*

$$b \approx_{BE} b^* \Leftrightarrow \exists \phi \in \Phi_{SG} : b = \phi(b^*).$$ □

The fact that $\Phi_{SG} \in [\mathbb{M} \to \mathbb{M}]$ and $\Phi_{SG} \in [BE \to BE]$ both constitute algebraic groups ensures that the two relations $\approx_{\mathbb{M}}$ and $\approx_{BE}$ are indeed equivalence relations. The set of all equivalence classes for $\approx_{\mathbb{M}}$ is denoted

$\mathbb{M}_{\approx}$. Similarly with $\approx_{BE}$ and $BE_{\approx}$. The equivalence class of an element x is denoted $[x]$. This notation is naturally extended to sets: $[X] = \bigcup_{x \in X}[x]$.

Now OS-graphs are formally defined.

Definition 9 *Let a consistent permutation symmetry specification for CPN be given. The* **OS-graph** *is the directed graph $OSG = (V, A, N)$ where:*

1. $V = \{C \in \mathbb{M}_{\approx} \mid C \subseteq [M_0>\}$.
2. $A = \{(C_1, B, C_2) \in V \times BE_{\approx} \times V \mid \exists (M_1, b, M_2) \in C_1 \times B \times C_2 : M_1[b>M_2\}$.
3. $\forall a = (C_1, B, C_2) \in A : N(a) = (C_1, C_2)$. □

Item 1 defines the set of nodes — one node for each reachable equivalence class of markings. Item 2 similarly defines the set of arcs. Item 3 is necessary, because we utilise a definition of directed graphs, which is slightly different from what normally appears in classical literature on graph theory. Apart from the set of nodes and the separately defined set of arcs, we have a function mapping each arc to a pair of nodes — the first component being the source and the second the destination. In this way, multiple arcs between two nodes are allowed, and this may appear in OS-graphs.

4 A Computer Tool Supporting OS-graphs

This section describes the newly developed **Design/CPN OS Graph Tool (OS-tool)** [10], which supports generation, analysis, and drawing of OS-graphs. The OS-tool is embedded in Design/CPN [8], the general tool for CP-nets mentioned in sect. 2, which supports editing, simulation, and occurrence graph analysis of CP-nets. The existing support for O-graphs in Design/CPN (O-tool) [11] has served as a basis for the implementation of the OS-tool. Sect. 4.1 provides an overview of the OS-tool, while sect. 4.2 uses the drawing facilities of the tool to compare O- and OS-graphs.

4.1 Overview of the OS-tool

Fig. 8 gives an overview of the various parts of the OS-tool. The grey boxes in the figure represent parts which are either modified or new compared to the O-tool. The white boxes are parts which are identical to parts in the O-tool. The OS-tool consist of three major parts. A *Graphical User Interface (GUI)*, a *CPN ML* part, and an *Interface* between these two parts.

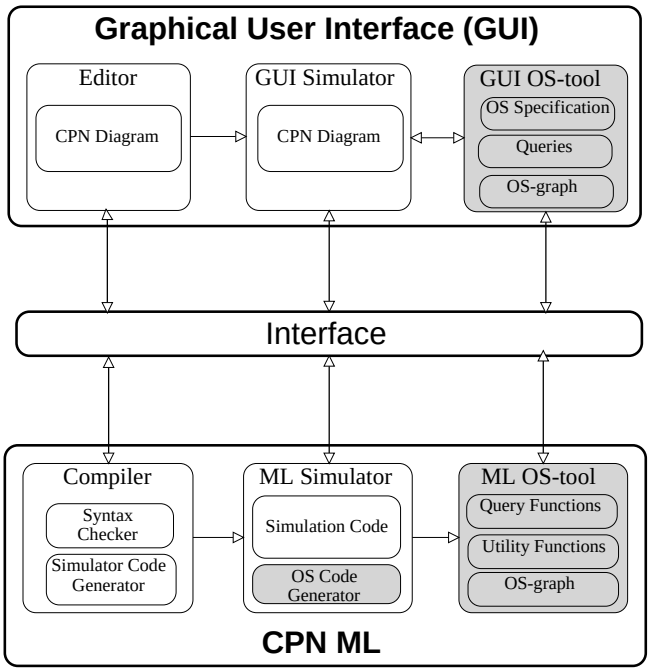

Figure 8: Overview of the OS-tool.

The Graphical User Interface is the front-end of the application. When the user has created a *CPN Diagram* in the *Editor*, the *Compiler* in the *CPN ML* part can be invoked. The Compiler has two parts: First, the CPN diagram is syntax checked by the *Syntax Checker*. If the CPN diagram represents a legal CP-net, then the *Simulation Code Generator* is invoked to generate the *Simulation Code* for the *ML Simulator*. Once this code has been generated, the CPN model can be simulated — the user can examine markings and execute steps directly on the CPN Diagram in the *GUI Simulator*. In the ML Simulator, we have implemented an *OS Code Generator*. This code generator uses the Simulation Code and the user-written *OS Specification* (a permutation symmetry specification), provided through the *GUI OS-tool*, to generate the necessary code for the *ML OS-tool*. The OS Specification is written using the *Utility Functions*. When the code for the ML OS-tool has been generated, the user can start generate and draw (parts of) an *OS-graph*, and make *Queries* using the *Query Functions* to investigate properties of his CPN model.

The OS-tool stores equivalence classes using representatives: Each node in the OS-graph is represented by a marking from its equivalence class. Analogously for arcs and binding elements.

Before an OS-graph can be generated, the user is required to implement a permutation symmetry specification. In the current version of the OS-tool, this consists of writing two CPN ML functions: A predicate $EquivMark$ defining when two markings are equivalent, and a predicate $EquivBE$ defining when two binding elements are equivalent. These two predicates must reflect the symmetry groups that the user has assigned to the atomic colour sets, and they must implement the rules saying how structured colour sets inherit their symmetry groups from their base colour sets. Moreover, the user must make sure that the predicates implement a consistent permutation symmetry specification. In the current version of the tool, this is not checked automatically. In a future version, the user will only have to assign a symmetry group to each of the atomic colour sets. The tool will then automatically generate $EquivMark$ and $EquivBE$.

When the predicates $EquivMark$ and $EquivBE$ have been written, a predefined function that generates the OS-graph can be invoked. When the generation has finished, the user is ready to analyse the OS-graph to get information about the considered CP-net. The function that generates the OS-graph implements an algorithm from [4]. This algorithm is a natural modification of the algorithm to construct a normal state space, i.e., an O-graph: The test of equality before a new node is inserted, is replaced by a test for equivalence. Similarly, the algorithm to construct OS-graphs precedes insertion of an arc with a test for equivalence.

The algorithm is shown in fig. 9. It uses a number of auxiliary functions: $Node/Arc$ creates a node/arc in the OS-graph for the given equivalence class, and $Node$ moreover adds its argument to the set $Waiting$ of unprocessed nodes. *Select* picks a node from a given set. *Represented* uses the predicates $EquivMark$ and $EquivBE$, provided by the user, to determine whether the equivalence class of the given node/arc is already in the OS-graph.

4.2 A First use of the OS-tool

In this section, we will illustrate the drawing facilities of the OS-tool. With respect to verification, drawing is of minor importance. Generation of the OS-graph followed by suitable queries is the way to verify systems. However, drawings are very adequate for presentation purposes. Here, we will use them

```
Waiting := ∅;
Node([M0]);
repeat
        Select(M1,[Waiting]);
        forall (b,M2) such that M1[b>M2 do
        begin
                if not(Represented(M2)) then
                   Node([M2])
                fi;
                if not(Represented([M1],[b],[M2])) then
                   Arc([M1],[b],[M2])
                fi
        end
        Waiting := Waiting − {[M1]}
until Waiting = ∅;
```

Figure 9: Algorithm to generate an OS-graph.

to compare the O- and OS-graph for the CPN model of Lamport's Algorithm, for $N = 3$.

Part of the O-graph is shown in fig. 10. To enhance readability, we have only shown some of the markings and some of the binding elements. Node 1 is the initial marking. The text placed right above the node describes the marking. Empty places are not listed. In the initial marking, three binding elements are enabled. They correspond to the three output arcs from node 1. Consider the arc leading from node 1 to node 2. From the text placed on this arc, it can be seen that an occurrence of the binding element $(setbi\_2, < i = 3, bi = false >)$, in the initial marking, leads to the marking of node 2. This marking is described by the text right above node 2.

When the permutation symmetry specification SG_L for the CPN model of Lamport's algorithm is implemented, the OS-graph can be generated and drawn. Part of it is shown in fig. 11. As in fig. 10, we have associated texts with the nodes and arcs, which describe the corresponding marking or binding element, chosen as representatives for the equivalence classes.

Let us in detail compare the partial O-graph in fig. 10 with the partial OS-graph in fig. 11. We will argue that they contain the same information, namely all occurrence sequences of the CPN model with at most two single steps. Consider node 1 in the OS-graph. This node represents the set of markings, which are equivalent to the initial marking. Because the permu-

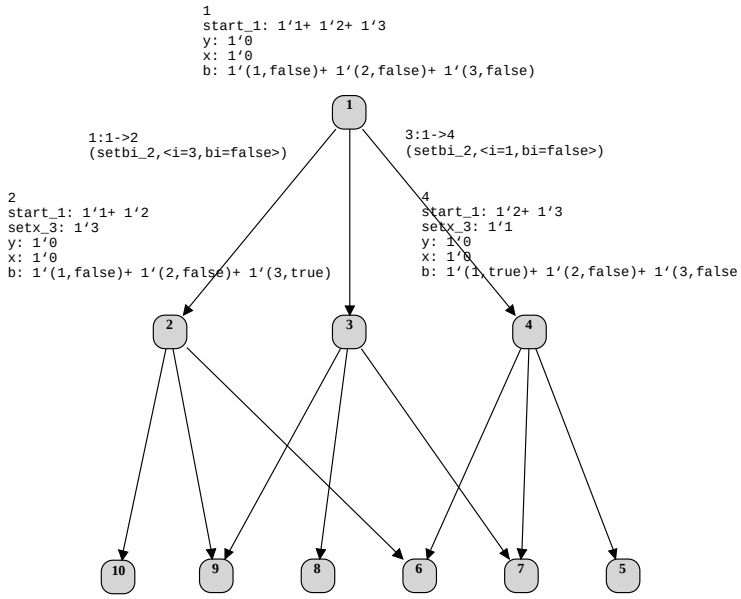

Figure 10: Part of O-graph for the CPN model of Lamport's Algorithm.

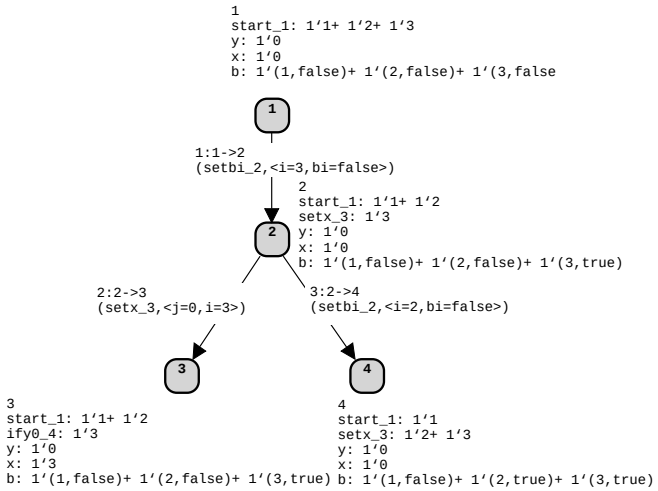

Figure 11: Part of OS-graph for the CPN model of Lamport's Algorithm.

tation symmetry specification is consistent, we know from item 1 of def. 7 that the size of this equivalence class is 1. Hence, node 1 in the OS-graph represents the equivalence class consisting exactly of node 1 in the O-graph. Nothing is saved yet.

Things, however, improve when we consider the immediate successors of node 1 in the two graphs. In the O-graph, node 1 has three successors; in the OS-graph, only one successor. This is because nodes 2, 3, and 4 in the O-graph are symmetric, i.e., belong to the same equivalence class. E.g., node 2 can be mapped into node 4 by swapping the processes 1 and 3. The occurring binding elements, which lead from node 1 to the nodes 2, 3, and 4, are also symmetric, and therefore, the OS-graph has only one arc from node 1 to node 2.

In a similar fashion, nodes 5, 8, and 10 of the O-graph are symmetric. They are all markings in which two different processes have executed one statement each, and they are represented by node 4 in the OS-graph. The same goes for the nodes 6, 7, and 9. They are all markings in which one process has executed two statements, and are represented by node 3 in the OS-graph.

5 Correctness of Lamport's Algorithm

In this section, we describe how to verify Lamport's Algorithm by means of OS-graphs. In sect. 5.1, some properties expressing the correctness of Lamport's Algorithm are listed. In sect. 5.2, these are translated into dynamic properties of the CPN model. Finally in sect. 5.3, we consider how to verify dynamic properties for CP-nets using OS-graphs.

5.1 Properties of Lamport's Algorithm

In [12], a number of properties that mutual exclusion algorithms must posses in order to be correct are discussed. These properties are 1 to 4 listed below:

1. *Mutual exclusion*: At any time, no more than one process is in the critical section.

2. *Persistent reachability of the critical section*: When several processes attempt to enter the critical section, eventually one will do so. It is not possible to have a situation in which all processes are starved.

3. *No deadlocks*: No execution of the mutual exclusion protocol can lead to a situation in which there is no activity among the processes, i.e., a situation in which all processes are blocked.

4. *Independence*: The behaviour of a process outside the mutual exclusion protocol does not influence the protocol.

In addition to these minimal requirements, there are some additional properties, which we would like to verify. They are:

5. *Return to start*: In any execution, it is always possible to return to a state in which all processes are positioned at the start label.

6. *No dead code*: Any statement always has the possibility of being executed by some process in the future.

Obviously, there are logical relations between some of these properties. E.g., *No dead code* implies *No deadlocks*.

5.2 Translation into CPN Dynamic Properties

Now, we explain how the properties formulated for Lamport's Algorithm in the previous section can be verified by means of the CPN model. Each property of Lamport's Algorithm is translated into a dynamic property of the CPN model. The necessary formal definitions are given as we proceed. For a more complete description of dynamic properties for CP-nets, the reader is encouraged to consult [1].

Mutual exclusion

An **integer bound** for a place p is a limit on the number of tokens on p in all reachable markings. The **best integer bound** for p is the maximal number of tokens on p in any reachable marking. Formally:

Definition 10 *$n \in \mathbb{N}$ is an* **integer bound** *for $p \in P$, iff*

$$\forall M \in [M_0>: |M(p)| \leq n.$$

If an integer bound exists, p is said to be **bounded**. *For a bounded place p, the* **best integer bound** *is the minimal $n \in \mathbb{N}$ such that n is an integer bound.* □

The *Mutual exclusion* property can be verified by considering the place CS_21 in the CPN model (see fig. 2): When CS_21 contains a token with colour i, it corresponds to process i being in the critical section. If 1 is an integer bound for CS_21, then at any time at most one process will be in the critical section.

Persistent reachability of the critical section

A transition t is **impartial**, iff in any infinite occurrence sequence starting in the initial marking, t has infinitely many occurrences. Formally:

Definition 11 *Let IOS be the set of infinite occurrence sequences starting in M_0 and $OC_t(\sigma)$ be the number of occurrences of a transition $t \in T$ in an infinite occurrence sequence $\sigma \in IOS$.*

A transition $t \in T$ is **impartial**, *iff*

$$\forall \sigma \in IOS : OC_t(\sigma) = \infty. \qquad \square$$

The *Persistent reachability of the critical section* property can be verified by considering the transition $sety0\_23$: When it occurs, process i is leaving the critical section. If $sety0\_23$ is impartial, then we cannot have an infinite occurrence sequence in which the critical section is not left and, hence, not entered by some process an infinite number of times. Thus, the critical section remains always reachable.

However, if no infinite occurrence sequence exists, the impartiality property is trivially fulfilled. We therefore also have to establish the existence of an infinite occurrence sequence.

No deadlocks

A marking M is **dead**, iff no binding element is enabled in M. Formally:

Definition 12 *A marking $M \in \mathbb{M}$ is a* **dead**, *iff*

$$\forall x \in BE : \neg M[x> . \qquad \square$$

The *No deadlocks* property can be proved directly by proving that the CPN model has no dead markings: Then, at any time during execution, at least one transition will be enabled and, hence, at least one process will be able to execute a statement.

Independence

For this property, we only need the basic concepts of markings and enabling already defined in defs. 3 and 4.

The *Independence* property is established, if we can verify that a process cannot be forced to enter the mutual exclusion protocol in order to unblock processes, which are executing the mutual exclusion protocol.

Entering the mutual exclusion protocol corresponds to occurrence of the transition *setbi_2*. All other transitions of the CPN model are internal to the protocol. What we want to show, is that if *setbi_2* is the only enabled transition, then all processes are outside the protocol, i.e., on the place *start_1*.

Return to start

A set of markings X is a **home space**, iff it is possible from any reachable marking to reach one of the markings in X. Formally:

Definition 13 *A set of markings $X \subseteq \mathbb{M}$ is a* **home space**, *iff*

$$\forall M \in [M_0>: X \cap [M> \neq \emptyset.$$ □

The *Return to start* property holds, if the set of markings X described next constitutes a home space: A marking M belongs to X, iff it is identical to the initial marking for all places but x, which is allowed to contain any single PID-token — in contrast to y, x will never be equal to 0, except from at the very beginning.

No dead code

A transition t is **live**, iff from any reachable marking, we can reach a marking in which t is enabled. Formally:

Definition 14 *A transition $t \in T$ is* **live**, *iff*

$$\forall M' \in [M_0> \exists M'' \in [M'> \exists x \in BE(t) : M''[x>.$$ □

The *No dead code* property holds, if all transitions are live: Liveness of a transition means that the corresponding statement always has the possibility of being executed.

No fairness

In addition to the properties listed in sect. 5.1, yet another property of Lamport's Algorithm is easy to derive from the CPN model. The algorithm is *not* fair: Any process wanting to enter the critical section may be starved forever. In the CPN model in fig. 2, an infinite occurrence sequence starving any given process can easily be constructed.

5.3 Verification by Means of OS-graphs

The dynamic properties for CP-nets introduced in sect. 5.2, can be proved by considering the OS-graph. It is worthwhile also to construct the strongly connected components (SCCs) of the OS-graph and consider the **SCC-graph** [4]. Investigating the SCC-graph instead of the OS-graph may significantly speed up the check of a dynamic property. Using Tarjan's algorithm (see, e.g., [13]) or a similar algorithm, the construction of the SCC-graph is an inexpensive operation. Its time complexity is linear in the size of the OS-graph.

The reader interested in how the individual dynamic properties are verified using the OS- and the SCC-graph is referred to [4] or [14]. The crucial observation to make here is that to use the OS-tool, it is not necessary to know these details. The user simply invokes the appropriate query function and gets back a result. E.g., if the user wants to verify the mutual exclusion property, formulated as an integer bound on the place CS_21, he simply invokes a function, which takes a place as argument and returns the best integer bound as result. Since all other properties of Lamport's Algorithm in the previous section were formulated as dynamic properties of the CPN model, they can all be verified using the query functions, which are part of the OS-tool.

6 Carrying out the Verification

In this section, we consider the actual verification of Lamport's Algorithm using the OS-tool. Sect. 6.1 describes necessary preparations. Sect. 6.2 reports on the application of the OS-tool, and includes statistics gathered to compare O- and OS-graphs. Finally, in sect. 6.3, the obtained verification results are discussed.

6.1 Preparation of the Verification

In order to use the OS-Tool for verification of Lamport's Algorithm, we have to prove that the permutation symmetry specification SG_L is consistent, i.e., prove that the three requirements in def. 7 are fulfilled. The proof, which is included in full detail in [14], consists of a large number of cases, all of which are truly trivial. We will not present the proof in this paper. One thing related to the proof should, however, be noted at this point. In sect. 2.1, we modelled a more general form of the **for**-statement in Lamport's Algorithm. We did not specify the order in which the entries in the b-array were to be tested. Had we done so, the permutation symmetry specification would not have been consistent. The reason is that if the entries are to be tested in turn staring from $b[1]$, then an ordering is imposed on the processes in Lamport's Algorithm. Hence, all processes are not treated in the same way from a symmetric point of view.

Once the permutation symmetry specification is proved consistent, the OS-tool can be applied. Verification of Lamport's Algorithm amounts to the following steps, which will be discussed below.

1. Implementation of the permutation symmetry specification.
2. Generation of the OS-graph.
3. Generation of the SCC-graph for the OS-graph.
4. Invocation of suitable query functions.

Item 1 consists of implementing the predicates $EquivMark$ and $EquivBE$ previously discussed in sect. 4.1. The utility functions provided by the OS-tool to support the implementation of the predicates are described, together with the underlying data structures, in [15].

In this paper, we will not describe how to implement the two predicates. They are included in full detail in [14]. For a CPN model like the one for Lamport's Algorithm, it is very easy to program a naive version of $EquivMark$ and $EquivBE$. One way to implement, e.g., $EquivMark$ is just to let it test all permutation symmetries in turn. If one is found that maps the first marking given as argument to the second, true is returned. Otherwise false is returned. However, for efficiency reasons, it is important to write the predicates in a more clever way. We experienced that this was manageable,

although both algorithmic and programming errors were made and had to found and corrected.

When the permutation symmetry specification has been implemented, the OS-graph and the SCC-graph can be generated (items 2 and 3). This is fully automatic — two generation functions are available via menus. Finally, suitable query functions (item 4) can be invoked to produce the desired verification results.

6.2 Application of the OS-tool

An inherent property of the occurrence graph method is that any graph is generated for a fixed value of the system parameters — in this case the number of processes N. Thus, the algorithm was verified for a set of fixed values. The computing power available determines the possible values of N. The results presented here were obtained on a SUN Sparc Workstation with 256 MB of RAM.

In addition to generating and analysing the OS-graphs, we also considered O-graphs. This is a main point, because the overall goal of using OS-graphs is to save space, and we want to demonstrate that this was actually accomplished. Table 1 contains the sizes of the O- and OS-graphs. The columns with headline Ratio shows the reduction factor for the OS-graph compared with the O-graph. It holds the number of nodes and arcs, respectively, for the O-graph divided with the corresponding number for the OS-graph. The outermost right column lists the factorial $N!$ of N, i.e., the size of the group of permutation symmetries.

Table 1: Sizes of O- and OS-graphs.

N	Nodes			Arcs			N!
	O-graph	OS-graph	Ratio	O-graph	OS-graph	Ratio	
2	380	191	2.0	716	358	2.0	2
3	19,742	3,367	5.9	58,272	9,788	6.0	6
4	1,914,784	83,235	23.0	9,046,048	383,030	23.6	24

Due to the state explosion problem, O-graphs could only be generated for values of N up to 3. In spite of this, for $N = 4$, we actually do know the size

of the O-graph. It is calculated from the OS-graph. Using algebraic group theory, we have designed an efficient algorithm to do so without unfolding. The details of the method are described in [16]. This algorithm is interesting, because it enables us to compare the sizes of the O- and OS-graph, even when generation of the O-graph is impossible. The algorithm also turned out to be a significant test to justify that the implementation of the permutation symmetry specification, i.e., the predicates $EquivMark$ and $EquivBE$ was correct, in the sense that it captured the intended assignment of symmetry groups to the atomic colour sets, and the inheritance rules for the structured colour sets. Moreover, the algorithm was suitable to increase our confidence in the consistency of the chosen permutation symmetry specification. For $N \leq 3$, if a discrepancy between the size of a generated O-graph and the size calculated from the OS-graph appeared, then we knew that something was wrong. Using this test, we corrected two non-trivial errors (see [14]) in our initial implementation of $EquivMark$. When an accordance between the sizes obtained by generation and calculation was recorded, it was very strong evidence that the CPN model and the permutation symmetry specification were as intended. In this way, the algorithm was used to narrow the gap between the abstract permutation symmetry specification, i.e., the assignment of algebraic groups to the atomic colour sets, and its implementation.

Now, consider the time used for the verification. Generation of SCC-graphs and evaluation of query functions take a relatively short time. The dominant time-consuming task is to generate the OS-graphs (or the O-graphs when we want to compare). These generation times are contained in table 2. An empty entry (-) signals that the measure could not be obtained.

Table 2: Generation times for O- and OS-graphs.

N	Seconds of CPU time		
	O-graph	OS-graph	Ratio
2	5	4	1
3	2,259	84	27
4	-	17,472	-

6.3 Discussion of the Verification

With OS-graphs, we could verify Lamport's Algorithm for all $N \leq 4$. Results from queries in the OS-tool showed that the correctness properties listed in sect. 5 were true.

From table 1, it can be seen that for a given N, the O-graph is almost $N!$ bigger than the OS-graph. This is remarkable. Because no more than $N!$ permutation symmetries are available, an equivalence class cannot be bigger than $N!$. Therefore, $N!$ is a theoretical limit on the size of the O-graph divided by the size of the OS-graph. I.e., the reduction obtained is almost maximal.

From table 2, it can be seen that for a given N, generation of the OS-graph was faster than generation of the O-graph. Even though we only have two observations, they indicate what seems to be a general fact: What it lost on a more expensive test on equivalence of markings and binding elements, is accounted for by having fewer nodes and arcs to generate; and also to compare with before a new node or arc can be inserted in the OS-graph. However, for $N = 4$, it took about five hours to generate the OS-graph. Thus pursuing more time-efficient generation methods are of paramount interest.

At a first glance, the values of N, for which Lamport's Algorithm can be verified, might not impress. We would of course like as large values as possible. Can anything be done with respect to creating a model more suitable for occurrence graph analysis? The answer is yes, but we pay a price with respect to the credibility of the verification. If we model the **for**-statement in a more coarse fashion, we are be able to do the verification for all $N \leq 6$. The way to modify the modelling of the **for**-statement is to have one transition, which is enabled when all $b[i]$'s are false, instead of testing all the entries of the b-array individually. This is a bit dangerous though, because it violates the assumption about atomicity in Lamport's Algorithm. A non-atomic statement is modelled as if it was atomic, jeopardising the correctness of the model. Anyway, for $N = 6$, the OS-graph has 83,895 nodes and 360,933 arcs. The O-graph is very big: 34,258,216 nodes and 175,300,026 arcs.

As explained in the beginning of this section, a slightly generalised version of Lamport's Algorithm was the subject for our verification, because of a problem caused by the **for**-statement with respect to applying OS-graphs. The model of the generalised algorithm has a larger O-graph than the model of the original algorithm. Thus, even though OS-graphs yield big savings, in some cases, the starting point for using them is worse than the starting point

for using O-graphs. However, it is still worthwhile to use OS-graphs: For $N = 3$, the O-graph for the CPN model of the original algorithm has 11,978 nodes and 32,226 arcs. The OS-graph for the CPN model of the generalised algorithm has only 3,367 nodes and 9,788 arcs.

As an aside, after our own verification of Lamport's Algorithm, we discovered that **for**-statements have also been identified as causing problems with respect to exploiting symmetries in verification in [17].

7 Conclusions

The main contributions of this paper are the presentation of our newly developed OS-tool supporting verification of CP-nets by means of OS-graphs, and the demonstration of the OS-graph method on a non-trivial example.

Using OS-graphs, it was possible to verify the crucial properties of Lamport's Algorithm. Once the permutation symmetry specification was proved consistent and implemented in terms of the predicates $EquivMark$ and $EquivBE$, the verification was very easy and almost automatic: Generate an OS-graph and an SCC-graph, and invoke suitable query functions in the OS-tool.

In our search for a good example to demonstrate the OS-tool for verification, the inspiration to consider Lamport's Algorithm came from Balbo et al. [7]. Here, the authors verify Lamport's Algorithm using Coloured Stochastic Petri Nets [18] [19] and place invariants. Balbo et al. verify Lamport's Algorithm on a model in which the **for**-statement is modelled in the coarse fashion described at the end of sect. 6. An advantage of the approach of Balbo et al. is that Lamport's Algorithm is verified for an arbitrary value of N.

In the original presentation of Lamport's Algorithm in [6], Lamport himself establishes correctness. He uses an axiomatic method decorating the algorithm text with assertions. Lamport concentrates on establishing deadlock freedom and mutual exclusion. As in [7], the properties are proved for an arbitrary value of N. Both Balbo et al. and Lamport conduct complex and lengthy mathematical proofs. For the mutual exclusion property, the former only sketch the proof, while the latter more generally relies on a number of proof sketches.

Balbo et al. also study the performance of Lamport's Algorithm. It is an important subject, but outside the scope of the work we present in this paper.

With respect to the logical behaviour of the algorithm, we establish similar properties to Balbo et al. and Lamport, plus other important properties. The main virtue of our proof is that it is almost automatic and, hence, much less error-prone. We do not need to engage in detailed or complex mathematical arguments. One qualification should be made though: The complexity in our approach lies in implementing a permutation symmetry specification and in proving that it is consistent. If these two tasks were automated, the proof would be fully automatic. Although this would improve and ease the approach, the present situation is acceptable. This is because a manual proof of consistency of the permutation symmetry specification reduces to checking a number of trivial cases. Based on this, we claim that our results are quite reliable.

Our approach, however, has some drawbacks. First of all, it is necessary to fix the system parameter — in this case the number of processes. Secondly, the number of processes, which can be handled presently, is restricted to $N \leq 4$. Therefore, it is relevant to ask if we could have done better with respect to the chosen method of verification, e.g., if we had combined symmetries with other methods for condensing occurrence graphs. One idea is to consider Haddad's structural reductions [20]. However, as can be seen from an inspection of the CPN model in fig. 2, the conditions which are required in order to use structural reductions are not present. Yet another idea is to apply Valmari's stubborn sets [21]. It is generally recognised that stubborn sets and symmetries can be applied simultaneously, thus yielding an even smaller occurrence graph. Unfortunately, unlike symmetries, none of the versions of stubborn sets that we know of preserve, e.g., the best integer bound for places, used to prove mutual exclusion. Also, for CP-nets, no tool support for stubborn sets exists.

Exploiting the symmetries present in many distributed systems has also been done in related approaches like [22] in which arbitrary transition systems are considered. Here, symmetries are combined with binary decision diagrams (BDDs) to design an efficient model checking algorithm. With respect to symmetries, the basic ideas of this approach are to a large extent a reinvention of the ideas behind OS-graphs. Also, the ideas of Well-formed Coloured Nets (WNs) [23] resemble those of OS-graphs. Detection of symmetries in WNs can be fully automated, thus effectively eliminating the need of conducting a consistency proof.

The verification of Lamport's Algorithm showed three areas in which the OS-tool must be improved. First of all, writing the permutation symmetry

specification (the predicates $EquivMark$ and $EquivBE$) was error-prone and time-consuming, since it had to be done manually. Presently, we are working on an improved interface for permutation symmetry specifications: The user is only asked to assign his chosen symmetry groups to the atomic colour sets. The OS-tool then automatically generates $EquivMark$ and $EquivBE$. A preliminary prototype of the new interface exists. It is documented in [24]. Secondly, proving the consistency of the permutation symmetry specification is tedious, because of the many cases in the proof, which need to be considered. Therefore, it would be preferable, if the tool could check most or all of these cases automatically. This can be done in a way similar to the checking of a proposed place invariant as described in [4]. Finally, the time used for the generation of the OS-graph should be improved. One way of doing this is to take advantage of a special kind of symmetries called self-symmetries. The details of this idea are described in [4] [15].

In [25], the OS-tool has been used to study the correctness of other well-known mutual exclusion algorithms. Here, the authors were not in advance familiar with OS-graphs nor our tool. It took them less than two weeks to become familiar with the approach and to carry out the verification. These examples and our verification of Lamport's Algorithm confirm that OS-graphs, with the emergence of the OS-tool, is a step towards practical formal verification of non-trivial distributed systems.

Acknowledgements

Thanks to Kurt Jensen and Søren Christensen for help in this project; thanks to Thomas Hildebrandt for useful comments on the paper. We acknowledge grants from University of Aarhus Research Foundation and the Faculty of Science at University of Aarhus.

References

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

[2] T. Murata, "Petri Nets: Properties, Analysis and Applications," in *Proceedings of the IEEE, Vol. 77, No. 4*, pp. 541–580. IEEE Computer Society, 1989.

[3] H. Genrich, "Predicate/Transition Nets," in *High-level Petri Nets*, K. Jensen and G. Rozenberg, Eds., pp. 3–43. Springer Verlag, 1991.

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

[5] K. Jensen, "Condensed State Spaces for Symmetrical Coloured Petri Nets," *Formal Methods of System Design, Vol. 9, 1/2*, pp. 7–40, 1996, Special Issue on Symmetry in Automatic Verification. Kluwer Academic Publishers.

[6] L. Lamport, "A Fast Mutual Exclusion Algorithm," in *ACM Transactions on Computer Systems, Vol. 5, No. 1*, pp. 1–11. Association for Computing Machinery, 1987.

[7] G. Balbo, S. Bruell, O. Chen, and G. Chiola, "An Example of Modeling and Evaluation of a Concurrent Program Using Colored Stochastic Petri Nets: Lamport's Fast Mutual Exclusion Algorithm," in *IEEE Transactions on Parallel and Distributed Systems*, T.-Y. Feng, Ed., pp. 221–240. IEEE Computer Society, 1992.

[8] K. Jensen, S. Christensen, P. Huber, and M. Holla, *Design/CPN Reference Manual*, Computer Science Department, University of Aarhus, Denmark,
On-line version: `http://www.daimi.aau.dk/designCPN/`.

[9] J.D. Ullman, *Elements of ML Programming*, Prentice Hall, 1993.

[10] J.B. Jørgensen and L.M. Kristensen, *Design/CPN OS Graph Manual*, Computer Science Department, University of Aarhus, Denmark,
On-line version: `http://www.daimi.aau.dk/designCPN/`.

[11] S. Christensen, K. Jensen, and L.M. Kristensen, *Design/CPN Occurrence Graph Manual*, Computer Science Department, University of Aarhus, Denmark,
On-line version: `http://www.daimi.aau.dk/designCPN/`.

[12] M. Raynal, *Algorithms for Mutual Exclusion*, North Oxford Academic, 1986.

[13] A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, 1985.

[14] 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," Tech. Rep., Computer Science Department, University of Aarhus, Denmark, 1996,
On-line version: `http://www.daimi.aau.dk/designCPN/`.

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

[16] J.B. Jørgensen and L.M. Kristensen, "Efficient Calculation of the Size of the O-graph from the OS-graph," Tech. Rep., Computer Science Department, University of Aarhus, Denmark, 1996,
On-line version: `http://www.daimi.aau.dk/designCPN/`.

[17] C.N. Ip and D.L. Dill, "Better Verification Through Symmetry," *Formal Methods of System Design, Vol. 9, 1/2*, pp. 41–75, 1996, Special Issue on Symmetry in Automatic Verification. Kluwer Academic Publishers.

[18] C. Dutheillet and S. Haddad, "Aggregation and Disaggregation of States in Colored Stochastic Petri Nets: Application to a Multi-processor Architecture," in *Proceedings of the 3rd International Workshop on Petri Nets and Performance Models, Kyoto, Japan.* 1989, pp. 40–49, IEEE Computer Society Press.

[19] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, "Stochastic Well-formed Colored Nets and Multi-processor Modeling Applications," Tech. Rep., University Paris 6, France, 1990.

[20] S. Haddad, "A Reduction Theory for Coloured Nets," in *High-level Petri Nets*, K. Jensen and G. Rozenberg, Eds., pp. 399–425. Springer-Verlag, 1991.

[21] A. Valmari, "Stubborn Sets of Coloured Petri Nets," in *Proceedings of the 12th International Conference on Application and Theory of Petri Nets, Gjern, Denmark*, G. Rozenberg, Ed., 1991, pp. 102–121.

[22] E.M. Clarke, T. Filkorn, and S. Jha, "Exploiting Symmetries in Temporal Model Logic Model Checking," in *Proceedings of the 5th International Conference on Computer Aided Verification, Elounda, Greece*, C. Courcoubetis, Ed. 1993, pp. 450–462, Springer-Verlag.

[23] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, "On Well-Formed Coloured Nets and Their Symbolic Reachability Graph," in *High-level Petri Nets*, K. Jensen and G. Rozenberg, Eds., pp. 373–396. Springer Verlag, 1991.

[24] V. Becuwe and L. Joly, "An Improved Interface for Permutation Symmetry Specifications for Coloured Petri Nets," Tech. Rep., Computer Science Department, University of Aarhus, Denmark, 1996.

[25] V. Becuwe and L. Joly, "Computer Aided Verification of Mutual Exclusion Algorithms Using Coloured Petri Nets, Full Occurrence Graphs, and Occurrence Graphs with Symmetries," Tech. Rep., Computer Science Department, University of Aarhus, Denmark, 1996.