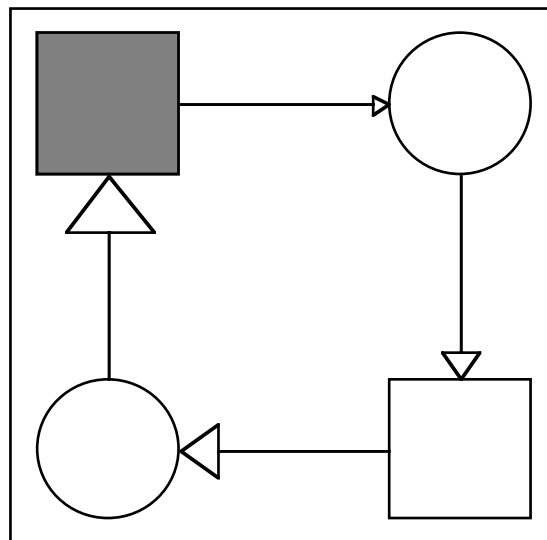


Design/CPN OE/OS Graph Manual

Version 1.1



University of Aarhus

Computer Science Department
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark
Tel: +45 89 42 31 88
Fax: +45 89 42 32 55

© 1998 University of Aarhus

© 1998 University of Aarhus

Computer Science Department

Ny Munkegade, Bldg. 540

DK-8000 Aarhus C, Denmark

Tel: +45 89 42 31 88

Fax: +45 89 42 32 55

e-mail: designCPN-support@daimi.aau.dk

Authors: Jens Bæk Jørgensen and Lars M. Kristensen.

Design/CPN is a trademark of Meta Software Corporation.

Macintosh is a registered trademark of Apple Computer, Inc.

Design/CPN OE/OS Graph Manual

Version 1.1

Table of Contents

INDEX	5
CHAPTER 1	7
Introduction to OE/OS Graphs	7
The History of the Design/CPN OE/OS Graph Tool	7
Example: Dining Philosophers	8
CHAPTER 2	12
How to Calculate an OE/OS Graph	12
Generation of OE/OS Graph Code	12
Generation of the OE/OS Graph	13
OEOS Menu	13
CHAPTER 3	19
How to Write OE/OS-Specifications	19
Example: Resource Allocation	19
Installing the OE/OS Specification	22
Generation of the OE/OS Graph	22
Key Functions	22
CHAPTER 4	24
How to Write OSP-Specifications	24
Example: Dining Philosophers	24
Example: Distributed Data Base	27
CHAPTER 5	33
How to Make Standard Queries	33
Reachability Properties	33
Boundedness Properties	36
Home Properties	38
Liveness Properties	41

OE/OS Graph Manual

Fairness Properties	44
How to Make Your Own Queries	45
Reference List	47

Index

—A—

AllReachable, 34
ApplyPermutation, 32
ApplyPermutationPair, 32
ApplyRotation, 26
atomic, 24
atomic colour sets, 24
Attributes/Options, 16

—B—

base colour sets, 24
BESLiveScc, 43
BESLiveSym, 43
BestLowerMultiSet, 37
BestUpperMultiSet, 37
Boundedness Properties, 36

—C—

Calculate Occ Graph, 14
Calculate Successors, 15
CalculateOEGraph, 22
CalculateSccGraph, 22
Change Marking, 12
Compatible equivalence specifications, 8
component sets, 40
consistency, 8
CreateCompSet, 41
CreateResSet, 30

—D—

DeadMarking, 42
Dining Philosophers, 8, 24
Display Arc, 17
Display Node, 16
Display Predecessors, 17
Display Scc Graph, 17
Display Successors, 17
Distributed Data Base, 27

—E—

Enter Occ Graph, 12
Enter Simulator, 12
equivalence specification, 8
EquivBE, 19
EquivMark, 19

—F—

Fairness Properties, 44
FairnessProperty, 45
FilterPermutations, 31
FilterPermutationsPair, 32

—G—

General Simulation Options, 12

—H—

Home Properties, 38
HomeMarking, 38
HomeMarkingExists, 39
HomeMarkingSym, 39
HomeSpace, 38
HomeSpaceSym, 39

—I—

InitialHomeMarking, 39
Installing, 22
IntersectCompSets, 41
IntersectResSets, 30

—K—

Key Functions, 22

—L—

ListDeadMarkings, 42
ListFairTIs, 45
ListHomeMarkings, 39
ListHomeScc, 39
ListImpartialTIs, 45
ListJustTIs, 45
ListPermutations, 30
Liveness Properties, 41
LowerInteger, 37
LowerMultiSet, 37

—M—

MarkingToKey, 22
MinimalHomeSpace, 38
ML Evaluate, 33

—O—

Occ menu, 12, 13
Occ State to Sim, 16
OE/OS graph code, 12
OE-graphs, 7
OS-graphs, 7
OSPSpec, 22
OSSpec, 22

—P—

PermExists, 30
permutation, 27
proj_ms1, 21
proj_ms2, 21

—R—

Reachability Properties, 33
ReachableSym, 35
Resource Allocation, 19
restriction sets, 30
Reswitch, 18
rotation, 24

—S—

[Save Report](#), 15
[SccListDeadMarkings](#), 42
[SccReachable](#), 34
[Show Statistics](#), 15
[Sim State to Occ](#), 16
Simulation Code Options, 12
Spec, 22
structured colour sets, 24
Sym, 35
SymPredicate, 35
Syntax Options, 12

—T—

TestPermutations, 31

TestPermutationsPair, 31
TestRotation, 26
TIsFairness, 45
[TIsLive](#), 42, 43
TIsStrictlyLive, 44
TIsStrictlyLiveScc, 44
TIsStrictlyLiveSym, 43
[Toggle Descriptor](#), 17

—U—

[Update Node](#), 18
[UpperInteger](#), 36

Chapter 1

Introduction to OE/OS Graphs

The History of the Design/CPN OE/OS Graph Tool

This manual describes a tool to calculate, analyse and draw occurrence graphs with equivalence classes and symmetries. Occurrence graphs with equivalence classes and symmetries (OE- and OS-graphs) is a compact class of occurrence graphs in which each node represents an equivalence class of markings and an arc represents an equivalence class of binding elements. OE/OS-graphs are often much smaller than ordinary occurrence graphs and can still be used to verify many dynamic properties of CP-nets.

The first version of the Design/CPN OE/OS Graph Tool (OE/OS tool) was developed at the University of Aarhus in 1995-1996. The present version 1.1 was developed in 1997-1998 and improves version 1.0. The calculation of OE/OS-graphs has been made more time efficient and the integration of the OE/OS tool in Design/CPN has been enhanced. The OE/OS tool resembles the Design/CPN Occurrence Graph Tool (OG tool) [OG] and this manual assumes that the reader is familiar with the OG tool. The reader is also assumed to be familiar with the theoretical background of OE/OS graphs presented in [CPN 2] on which the OE/OS tool and the terminology and notation used in this manual are based. In the following OE-graph abbreviates **“occurrence graphs with equivalence classes”**, OS-graph abbreviates **“occurrence graphs with symmetries”** and OSP-graph abbreviates **“occurrence graphs with permutation symmetries”**. OSP-graphs is an important subclass of OS-graphs which again is a subclass of OE-graphs. A difference between the three classes is the dynamic properties which can be verified on the corresponding graph. In this sense OSP-graphs are stronger than OS-graphs which again is stronger than OE-graphs.

The OE/OS tool is fully integrated with Design/CPN 3.1. This means that you can switch between the editor/simulator and the OE/OS tool. When an OE/OS-graph node has been found, a representative for the equivalence class can be inspected in the simulator. This means that you can see the marking directly on the graphical representation of the CPN model. You can see the enabled transition instances, investigate their bindings, and make simulations. Analogously, when a marking has been found in the simulator, it can be added to the OE/OS-graph or used as the initial marking for a new OE/OS-graph.

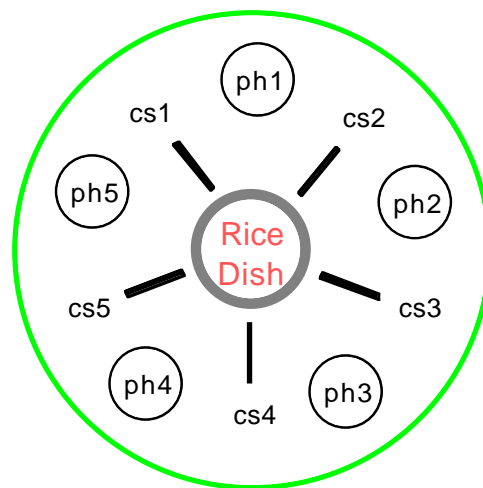
The user of the OE/OS tool is required to define the equivalence on markings and binding elements before an OE/OS-graph can be calculated by the tool. The user does this by implementing an **equivalence specification** which consists of two CPN ML functions. One which defines when two markings are equivalent and one which defines when two binding elements are equivalent. A large number of efficient built-in functions are provided which support the implementation of these predicates. It is the responsibility of the user to ensure the **consistency** of the provided equivalence specification. **Compatible equivalence specifications** are not supported by the tool.

Like the OG tool, the OE/OS tool has a large number of built-in standard queries. They can be used to investigate the standard properties of a CP-net, such as reachability, boundedness, home properties, liveness, and fairness. In addition to the standard queries there are a number of powerful search facilities allowing the user to formulate his own, non-standard queries.

Example: Dining Philosophers

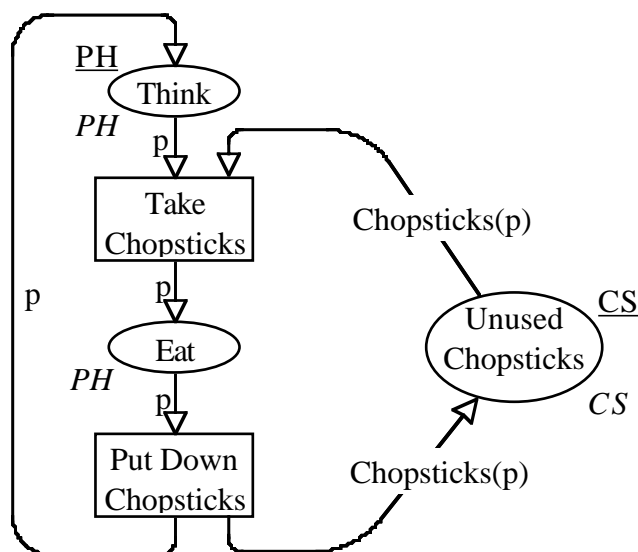
In this section we will briefly recall the concept of OE/OS-graphs. This will be done with off-set in ordinary occurrence graphs. The basic idea behind occurrence graphs is to make a directed graph with a node for each reachable marking and an arc for each occurring binding element. An introduction to occurrence graphs can be found in Sect. 5.1 of [CPN 1] and in Sect. 1.1 in [CPN 2]. An introduction to OE/OS-graphs can be found in Sect. 2 and Sect. 3 in [CPN 2].

In this manual we use the dining philosophers system as one of the main examples. Five Chinese philosophers are sitting around a circular table. In the middle of the table there is a delicious dish of rice, and between each pair of philosophers there is a single chopstick. Each philosopher alternates between thinking and eating. To eat, the philosopher needs two chopsticks, and he is only allowed to use the two which are situated next to him (on his left and right side). The sharing of chopsticks prevents two neighbours from eating at the same time.

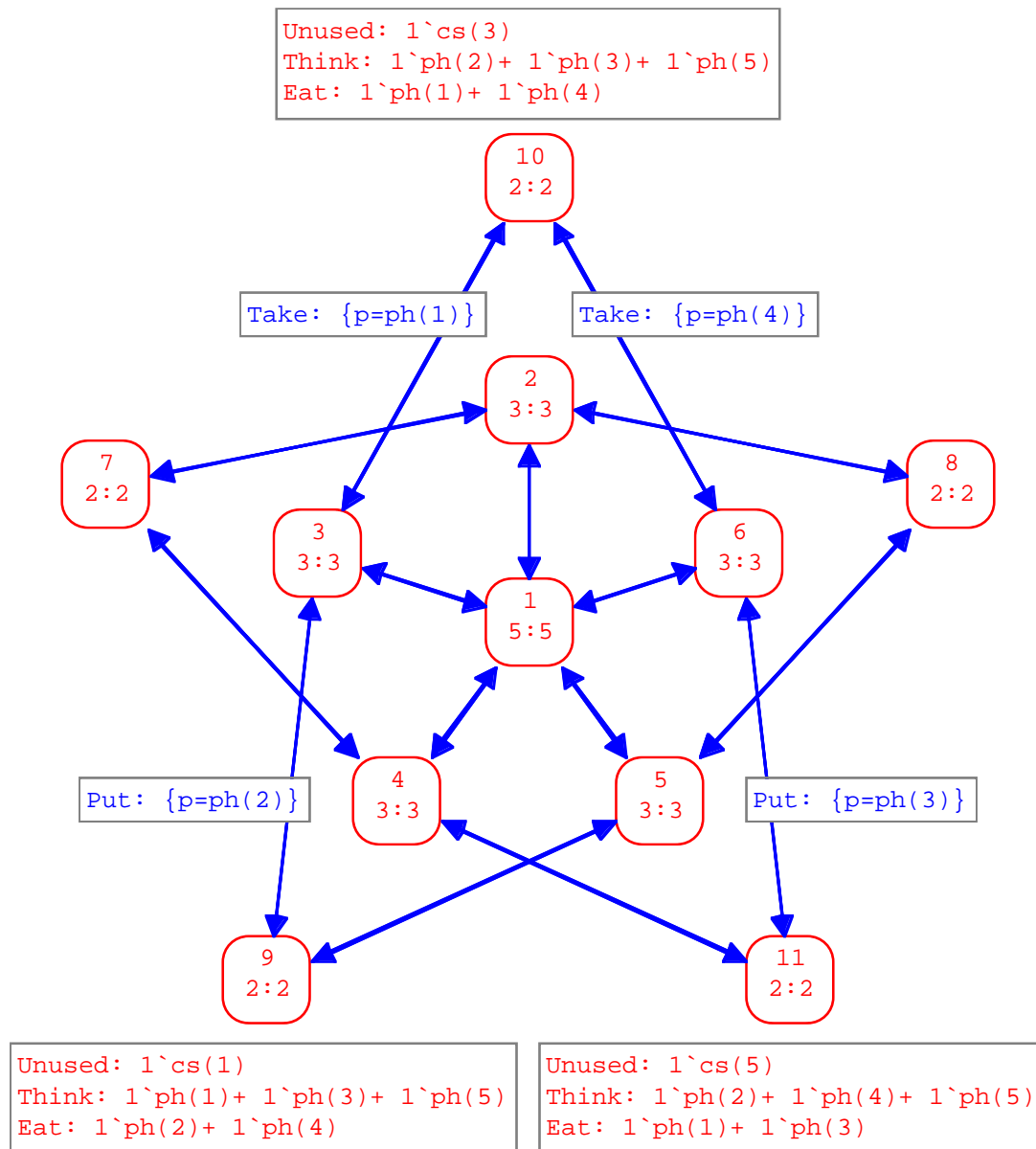


The dining philosopher system is modelled by the CP-net shown below. The PH colour set represents the philosophers, while the CS colour set represents the chopsticks. The function *Chopsticks* maps each philosopher into the two chopsticks next to him.

```
val n = 5;
color INDEX = int with 1..n;
color PH = union ph:INDEX declare ms;
color CS = union cs:INDEX declare ms;
var p : PH;
fun Chopsticks(ph(i))
  = 1`cs(i)+1`cs(if i=n then 1 else i+1);
```

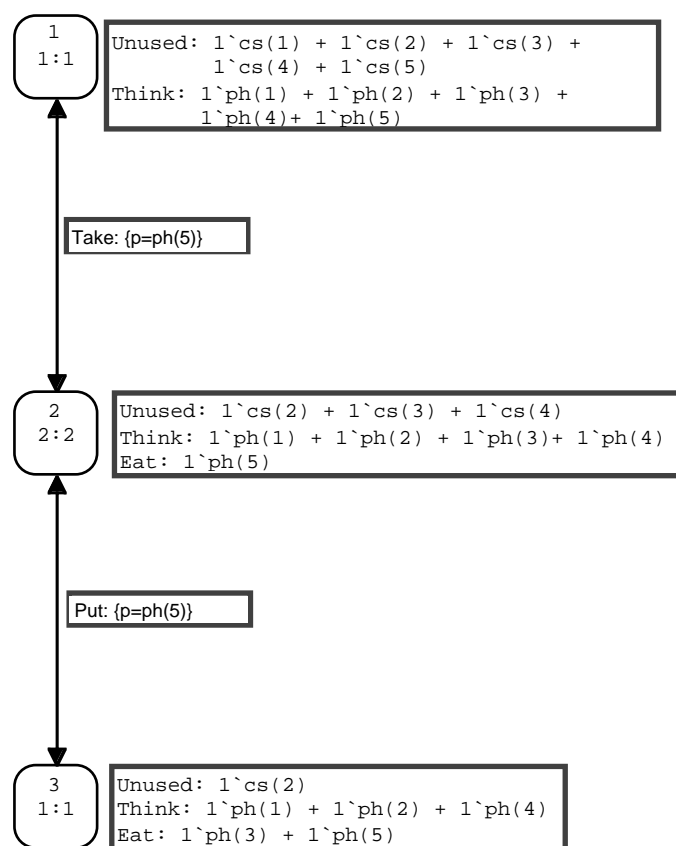


An occurrence graph for the dining philosophers is shown below. Each node represents a reachable marking, while each arc represents the occurrence of a single binding element – leading from the marking of the source node to the marking of the destination node. To improve readability, we have only shown the detailed contents of some of the markings and some of the binding elements. It should be noted that all arcs are double arcs (i.e., represents two individual arcs).



If we consider the philosopher system then all philosophers behaves in the same. Each of the philosophers has a special relationship with his two neighbours in the sense that he shares a chopstick with each of them. This is also reflected in the occurrence graph above. If we consider the markings corresponding to the nodes 9,10 and 11 in which two philosophers are eating it can be observed that they are

very alike. For instance, the marking of node 11 can be mapped to the marking of node 9 by mapping each philosopher into its neighbour and similarly for the chopsticks. By extending this idea we consider two markings to be equivalent if one can be obtained from the other by a rotation of the indices. Similarly for binding elements. The basic idea in OE/OS graphs is to lump together such equivalent markings and equivalent binding elements. If this is done for the philosopher system we obtain the following graph. Now each node represents an equivalence class of markings. For instance node 2 represents the nodes 2,3,4,5, and 6 in the occurrence graph. Similarly the arcs represent equivalence classes of binding elements.



As it can be observed the OE/OS-graphs has only 3 nodes and 4 arcs in contrast to the occurrence graph which had 11 nodes and 30 arcs. For each node and arc in the above graph we have shown a representative for the equivalence class which the node/arc represents.

Chapter 2

How to Calculate an OE/OS Graph

Before an occurrence graph can be calculated, it is necessary to generate the **OE/OS graph code**, i.e., the ML code which is used to calculate, analyse and draw OE/OS-graphs. The OE/OS-graph code is generated in a way which is similar to the switch from the editor to the simulator.

Generation of OE/OS Graph Code

To generate the OE/OS-graph code the following steps must be performed (in the specified order):

- a) Make sure that you are using Design/CPN version 3.1 (or later) and the CPN ML image provided together with it.
- b) Use **Syntax Options** to select *OG Tool Violations*. You may also want to select the five check boxes for missing and duplicate place, transition and page names.
- c) Use **General Simulation Options** to unselect *Time*. To choose the setting it may first be necessary to use **Simulation Code Options**.
- d) Use **Enter Simulator** to make a syntax check and to enter the simulator.
- e) Use **Change Marking** (or a simulation) to obtain the marking which you want to use as the initial marking of your OE/OS-graph. – If you want to use the initial marking of the CPN model as the initial marking of your OE/OS-graph, nothing needs to be done.
- f) Invoke **Enter OE/OS Graph** (in the File menu). This will create the OE/OS-graph code. For large nets it takes a while – comparable to the time for a full simulator switch. – If you do not need to customize the initial marking of the CPN model **Enter OE/OS Graph** can be invoked directly in the editor.

When **Enter OE/OS Graph** terminates, a new **OEOS menu** is added to the menu bar (at the rightmost end). This menu contains all the commands which are used to perform the calculation and

drawing of OE/OS-graphs. The menu works very similar to the Occ menu in the OG tool. The functionality of the individual items in the menu will be explained below.

We propose that you now try to generate the OE/OS-graph code for the dining philosopher system. To do this use the CPN model called “DiningPhilosophersOS”. It can be downloaded from the Design/CPN WWW pages.

Generation of the OE/OS Graph

Before an OE/OS-graph can be calculated the user are required to implement the equivalence specification, in the following referred to as an **OE/OS/OSP-specification** depending on the class of graphs we have in mind. How this is done is the subject of Chap. 3 and Chap 4.

Once this implementation is complete, the OE/OS-graph can be calculated in exactly the same way as an occurrence graph is calculated in the OG tool. In particular the branching and stop options are respected.

OEOS Menu

In this section we will briefly review the individual item in the OEOS menu:

*OEOS
Calculate OE/OS Graph
*Calculate Successors
*Calculate Scc Graph

*Show Statistics
*Save Report

*OE/OS State to Sim
*Sim State to OE/OS

*Attributes/Options

*Display Node
*Display Arc
*Display Successors
*Display Predecessors
*Display Scc Graph

*Toggle Descriptor
*Update Node.

Most of the commands are quite similar to the corresponding command in the OG tool. However, there are differences which we will make explicit below.

Calculate OE/OS Graph

This command calculates the OE/OS-graph. It implements the algorithm of Prop. 2.5 in [CPN 2]. The OE/OS tool stores equivalence classes using **representatives**. Therefore each node in the OE/OS-graph (an **OE/OS node**) is represented by a marking, a representative for the equivalence class of markings. Similarly the arcs in the OE/OS-graph (an **OE/OS arc**) are represented by a binding element, a representative for the equivalence class of binding elements. Besides from this, the command works in exactly the same way as in the OG tool.

How to Calculate an OE/OS Graph

Calculate Successors

This command calculates the immediate successors of the selected OE /OS node(s).

Calculate Scc Graph

This command calculates the Scc-graph of the OE/OS-graph. The Scc-graph is used by many of the query functions in Chap.5.

Show Statistics

This command gives information about the size of the OE/OS-graph and the size of the Scc-graph.

The OE/OS-graph will always have at least one node (even if **Calculate Occ Graph** and **Calculate Successors** have not been used). By convention node number 1 is the equivalence class containing the initial marking. If the Scc-graph has not been calculated, the second part of the statistics is missing.

The information from **Show Statistics** can also be accessed via the following set of ML functions:

fun	NoOfNodes	unit ->
int		
fun	NoOfArcs	unit ->
int		
fun	NoOfSecs	unit ->
int		
fun	EntireGraphCalculated	unit ->
bool		
fun	SccNoOfNodes	unit ->
int		
fun	SccNoOfArcs	unit ->
int		
fun	SccNoOfSecs	unit ->
int		
fun	SccGraphCalculated	unit ->
bool		

Save Report

This command is not supported in this version of the OE/OS tool. Selecting the corresponding menu item will not have any effect.

OE/OS State to Sim

This command “moves” the representative of an equivalence class in the OE/OS-graph to the simulator. The representative to be moved is specified by either selecting the equivalence class (the OE/OS node) prior to the invocation of the command or by specifying the equivalence class in a dialogue box.

Sim State to OE/OS

This command allows you to “move” a simulator state to the OE/OS-graph. If its equivalence class is already in the OE/OS-graph, the current simulator state itself will not be explicitly represented. I.e., the command **Sim State to OE/OS** followed by the command **OE/OS State to Sim** may change the state of the simulator.

Attributes/Options

This command allows you to change the diagram defaults for OE/OS attributes and the values of OE/OS options. This is done exactly in the same way as in the OG tool. For more information see Chap. 7 in [OG].

Display Node

This command draws a new OE/OS node – providing a graphical representation of the specified node. The node is drawn at the centre of the current page

If the node already exists on the current page, the corresponding OE/OS node becomes selected and nothing further happens. Hence, it is impossible to draw the same OE/OS node more than once on a page (but it can be drawn on different pages).

OE/OS nodes can also be drawn by means of the following ML functions:

```
fun DisplayNodes          Node list ->  
unit  
  
fun DisplayNodePath      Node list ->  
unit
```

The first function draws the nodes in the list (on the current page, reusing existing nodes). The second function checks whether the nodes form a path (i.e., whether there is an arc between each node and its immediate successor). If this is the case, the nodes and arcs are drawn (on the current page, reusing existing nodes and arcs). If there are multiple arcs between two neighbouring nodes, they are all

drawn. If the nodes do not form a path, the exception `NotAPath` is raised

Display Arc

This command draws an OE/OS arc – providing a graphical representation of the specified arc. If necessary the command also draws the source and destination node of the arc. Otherwise the command works in a way which is analogous to **Display Node**.

OE/OS arcs can also be drawn by means of the following ML functions, which work in a way which is totally analogous to `DisplayNodes` and `DisplayNodePath`:

```
fun DisplayArcs          Arc list -> unit
fun DisplayArcPath      Arc list -> unit
```

Display Successors

This command draws the immediate successor nodes and the immediate successor arcs of the selected node(s).

Display Predecessors

This command draws the predecessor nodes and predecessor arcs of the selected node(s). It works in a way which is totally analogous to **Display Successors**.

Display Scc Graph

This command draws the Scc-graph using a standard layout.

Toggle Descriptor

This command toggles the existence of the OE/OS node/arc descriptor of the selected OE/OS node/arc(s). If the descriptor does not exist, it is created. If it exists, it is deleted. The descriptor will typically describe the representative of the OE/OS node/arc. Since the OE/OS tool stores equivalence classes using representatives the `Mark-structure` and `ArcToBE/ArcToTI` functions can be used to obtain information about the representative.

Update Node

This command updates the information in the text string of the selected OE/OS node(s). It only has an effect if the occurrence graph has been extended since the OE node was drawn.

Details and Limitations (can be skipped in a first reading)

When you make a modification of the CPN diagram, it is necessary to regenerate all the OE/OS-graph code from scratch. This also means that the OE/OS-graph (if any) is lost. When the modification is made in the simulator it is sufficient to invoke **Reswitch** and **Enter Occ Graph**.

The OE/OS tool respects the relevant mode attributes. The OE/OS-graph is calculated for those parts of the net which would participate in a simulation. Please note that with OE/OS-graphs, it only makes sense to use code segments in a very limited fashion, e.g., to initialise a CPN model.

Currently OE/OS-graphs with time has not been theoretically developed and hence it does not make sense to construct timed OE/OS-graphs. It is therefore recommended to unselect time in **General Simulation Options**.

Free variables on output arcs are not allowed – unless they are variables of a small color set.

Chapter 3

How to Write OE/OS-Specifications

This chapter describes how to write OE- and OS-specifications. Before an OE/OS-graph can be calculated two functions must be implemented which defines the equivalence on markings and binding elements. Throughout this chapter we will use the resource allocation system from occurrence 1.2 in [CPN 1] as an example.

An OE/OS-specification consists of two functions `EquivMark` and `EquivBE` with the following functionality.

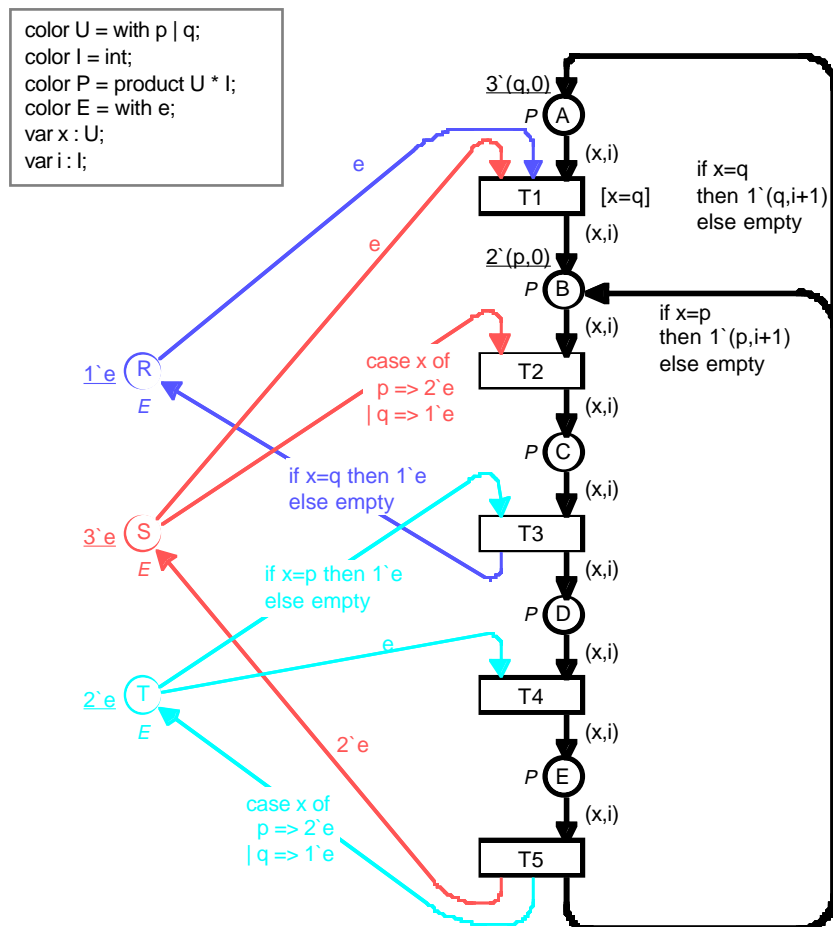
```
fun EquivMark NodeRec * NodeRec -> bool
fun EquivBE Bind.Elem * Bind.Elem ->
bool
```

It should be noted, that the two functions are not required to be named `EquivMark` and `EquivBE` - any legal ML identifier will work.

The fact that `EquivMark` takes two node records as argument enables us to use the `OEMark`-structure to refer to markings when writing the function. The `OEMark`-structure can be used to refer to marking in the same way as the `Mark`-structure. Similarly the functionality of `EquivBE` enables us to use the `Bind`-structure to refer to the binding elements (see Chap. 3 in [OG]).

Example: Resource Allocation

The resource allocation system is modelled by the CP-net shown below.



We will consider the OE-specification stating that two markings are equivalent iff they are identical when ignoring the cycle counters. Similarly for binding elements. The `EquivMark` function for the resource allocation system is shown below.

```

fun EquivMark (n1,n2) =
  (proj_ms1 (OEMark.Res'A 1 n1) ==
   proj_ms1 (OEMark.Res'A 1 n2)) andalso
  (proj_ms1 (OEMark.Res'B 1 n1) ==
   proj_ms1 (OEMark.Res'B 1 n2)) andalso
  (proj_ms1 (OEMark.Res'C 1 n1) ==
   proj_ms1 (OEMark.Res'C 1 n2)) andalso
  (proj_ms1 (OEMark.Res'D 1 n1) ==
   proj_ms1 (OEMark.Res'D 1 n2)) andalso
  (proj_ms1 (OEMark.Res'E 1 n1) ==
   proj_ms1 (OEMark.Res'E 1 n2)) andalso
  (OEMark.Res'R 1 n1) == (OEMark.Res'R 1
  n2)
  andalso
  (OEMark.Res'S 1 n1) == (OEMark.Res'S 1
  n2)

```

How to Write OSP-Specifications

```
andalso
(OEMark.Res'T 1 n1) == (OEMark.Res'T 1
n2)
```

On the places *A*, *B*, *C*, *D* and *E* the cycle counters are removed and the resulting multi-sets are compared. This is done using the function `proj_ms1` which takes a multi-sets of pairs and returns the corresponding multi-set of the first components. Two projection functions working on multi-set are available.

```
fun proj_ms1 (''a * 'b) ms -> ''a ms
fun proj_ms2 ('a * ''b) ms -> ''b ms
```

`proj_ms2` is similar to `proj_ms1` except that the corresponding multi-set of the second components is returned.

The function simply compares the marking on each of the places *R*, *S* and *T*.

The function `EquivBE` for the resource allocation system is written as a pattern match over all binding elements and can be seen below.

```
fun EquivBE (
  Bind.Res'T1 (1, {x=x1, i=_}),
  Bind.Res'T1 (1, {x=x2, i=_})) = (x1 =
x2)
| EquivBE (
  Bind.Res'T2 (1, {x=x1, i=_}),
  Bind.Res'T2 (1, {x=x2, i=_})) = (x1 =
x2)
| EquivBE (
  Bind.Res'T3 (1, {x=x1, i=_}),
  Bind.Res'T3 (1, {x=x2, i=_})) = (x1 =
x2)
| EquivBE (
  Bind.Res'T4 (1, {x=x1, i=_}),
  Bind.Res'T4 (1, {x=x2, i=_})) = (x1 =
x2)
| EquivBE (
  Bind.Res'T5 (1, {x=x1, i=_}),
  Bind.Res'T5 (1, {x=x2, i=_})) = (x1 =
x2)
| _ => false;
```

As it can be seen two binding elements are equivalent iff they have the same transition and if the variable *x* is bound to the same value.

Installing the OE/OS Specification

Once the implementation of the equivalence specification is done, the equivalence specification have to be installed. This is done by evaluating the following ML code.

```
OESet.Equivalence {  
  Mark = EquivMark,  
  Bind = EquivBE,  
  Spec = OESpec}
```

The `Spec` member is used to specify the class of the equivalence specification. `OESpec` indicates that the equivalence specification is an OE-specification. Other possibilities are `OSSpec` and `OSPSpec` which is used to denote symmetry specification and permutation symmetry specification respectively. This information is used by the tool when the user applies the standard query functions. We will return to this in Chap. 5.

Generation of the OE/OS Graph

Once the installation of the equivalence specification is done, the OE-graph can be calculated using **Calculate OE/OS Graph** in the Occ menu. Many of the query functions in Chap. 5 uses the Scc-graph. The Scc-graph can be calculated by invoking **Calculate Scc Graph** in the Occ menu. Calculating the OE-graph for the resource allocation system yields an OE-graph with 13 nodes and 20 arcs.

The OE-graph and the Scc-graph can also be calculated by means of the following two ML functions.

```
fun CalculateOEGraph    unit -> unit  
  
fun CalculateSccGraph  unit -> unit
```

Key Functions

In order to make the search for the markings more efficient, a search tree is used as explained in occurrence. 1.7 in [CPN 2]. In order for this search-technique to work it is required that equivalent markings are mapped to identical keys. The user specifies a mapping from markings to keys by writing a function `MarkingToKey` with the following functionality.

```
fun MarkingToKey  NodeRec -> string
```

How to Write OSP-Specifications

The function which the user must provide takes an argument of type `NodeRec` which makes it easy to use the `OEMark`-structure when defining the mapping. The function is installed by evaluating the following piece of ML code.

```
OESet.EncodeMarking (MarkingToKey)
```

In the current version of the tool it is only possible to change the encoding of markings when the OE/OS-graph consist of a single node. This is the case right after the **Enter OE/OS Graph** command has been completed.

As long as the equivalence specification preserves the number of tokens on the places , i.e, if for all places in the CP-net equivalent markings will have the same number of tokens on each place it is not necessary to change the default encoding. This is for instance the case with the equivalence specification for the resource allocation system above as well as a permutation symmetry specification (an OSP-specification).

Chapter 4

How to Write OSP-Specifications

In this chapter we consider permutation symmetries and show how to implement an OSP-specification.

A permutation symmetry specification assigns an algebraic group to each of the atomic colour sets in the considered CP-net. In the present version of the tool this is done in an indirect fashion. Like for the OE/OS-specifications the user still has to supply the functions `EquivMark` and `EquivBE`. The functions must be written in such a way that the assignment of the algebraic groups to the **atomic colour sets** and the inheritance of the symmetry groups of **structured colour sets** from their **base colour sets** are emulated. In the next version of the tool this will no longer be necessary - the user will only have to assign the algebraic groups to the atomic colour sets. The rest will be handled automatically.

In this chapter we will consider two examples. The Dining Philosophers from Chap.1 and the Distributed Data Base from Sect. 1.3 in [CPN 1]. We will show how to write the OSP-specification for these two examples and introduce the utility functions provided by the tool to do so. In the implementation of the OSP-specification below, it should be noted that for introductory reasons the primary concern has been to keep things simple. More efficient implementations of the OSP-specification can be written to obtain a faster calculation of the OSP-graph.

Example: Dining Philosophers

First we will consider the dining philosophers system. The CP-net for the dining philosophers can be seen in Chap. 1. The CP-net uses two indexed colour sets, PH for the philosophers and CH for the chopsticks. It uses the auxiliary colour set INDEX which is **atomic**. It is assigned the symmetry group **rotation**. The colour sets PH and CH are made structured by means of a union construction. In this way they inherit their symmetry groups from the base colour set INDEX and thereby the rotations of the philosophers and the chopsticks are synchronized.

How to Write OSP-Specifications

The function `EquivMark` performs a naive testing of all possible rotations. For n philosophers there are n possible rotations. The basic idea in `EquivMark` is to represent rotations as functions. The function `EquivMark` for the dining philosopher system is as follows and will be commented below.

```
(* list of indices *)
fun listindex 0 = [0]
  | listindex i =
      i::(listindex (i-1))

(* Rotation of lenght l of PH *)
val PHRot l =
  (fn (ph(i)) = ph((i+l) mod n));

(* Generate all rotations on PH *)
val PHRotations =
  (map PHRot (listindex
n));

(* Rotations of lenght l of CS *)
val CHRot l =
  fn (cs(i)) = cs((i+l) mod n));

(* Generate all rotations on CS *)
val CHRotations =
  (map CHRot (listindex
n));

(* Synchronize rotations *)
fun merge [] [] = []
  | merge x::xs y::ys =
      (x,y)::(merge xs ys)

val rotations =
  (merge PHRotations
CHRotations)

fun EquivMark (n1,n2) =
  predlist
    (fn (x1,x2) =>
      (TestRotation x1
        [(OEMark.System'Think 1 n1,
          OEMark.System'Think 1 n2),
         (OEMark.System'Eat 1 n1,
          OEMark.System'Eat 1 n2)])
      andalso
      (TestRotation x2
        [(OEMark.System'Unused 1 n1,
          OEMark.System'Unused 1
n2)]))
    rotations
end;
```

In the first part all rotations on PH and CS are generated and represented as a list of functions respectively. Using `merge` the rotations of philosophers and chopsticks are synchronized. In `EquivMark` the rotations are simply tested in turn to see if one of the rotations maps the marking of `n1` into the marking of `n2`. `EquivMark` uses the utility function `predlist` which given a predicate and a list determines whether some member in the list satisfies the predicate. The function terminates as soon as such a member is found. `EquivMark` also uses the function `TestRotation`:

```
fun TestRotation ('a -> 'a) *  
                  ('a ms * 'a ms) list ->  
bool
```

TestRotation takes a function representing a rotations and a list of pairs of multi-sets. It returns true if the rotation maps the first component into the second component of each of the multi-set pairs. Otherwise false is returned.

An additional utility functions related to rotations is also supported:

```
fun ApplyRotation ('a -> 'a) *  
                  'a ms -> 'a ms
```

ApplyRotation takes a rotation and a multi-set as argument and applies the provided rotation to the multi-set.

We will now consider how to write `EquivBE`. It must capture that a necessary condition for two binding elements to be equivalent is that the involved transitions are equal. Because both of the transitions in the CP-net have only one variable of colour set `INDEX` this is also a sufficient condition. The `EquivBE` function is shown below.

```
fun EquivBE (  
  Bind.System'Take (1,_),  
  Bind.System'Take (1,_)) = true  
| EquivBE (  
  Bind.System'Put = (1,_),  
  Bind.System'Put = (1,_)) = true  
| EquivBE (_,_) = false
```

Once the implementation of the two functions is complete the equivalence specification have to be installed. This is done by evaluating the following ML code.

```
OESet.Equivalence {  
  Mark = EquivMark,  
  Bind = EquivBE,
```

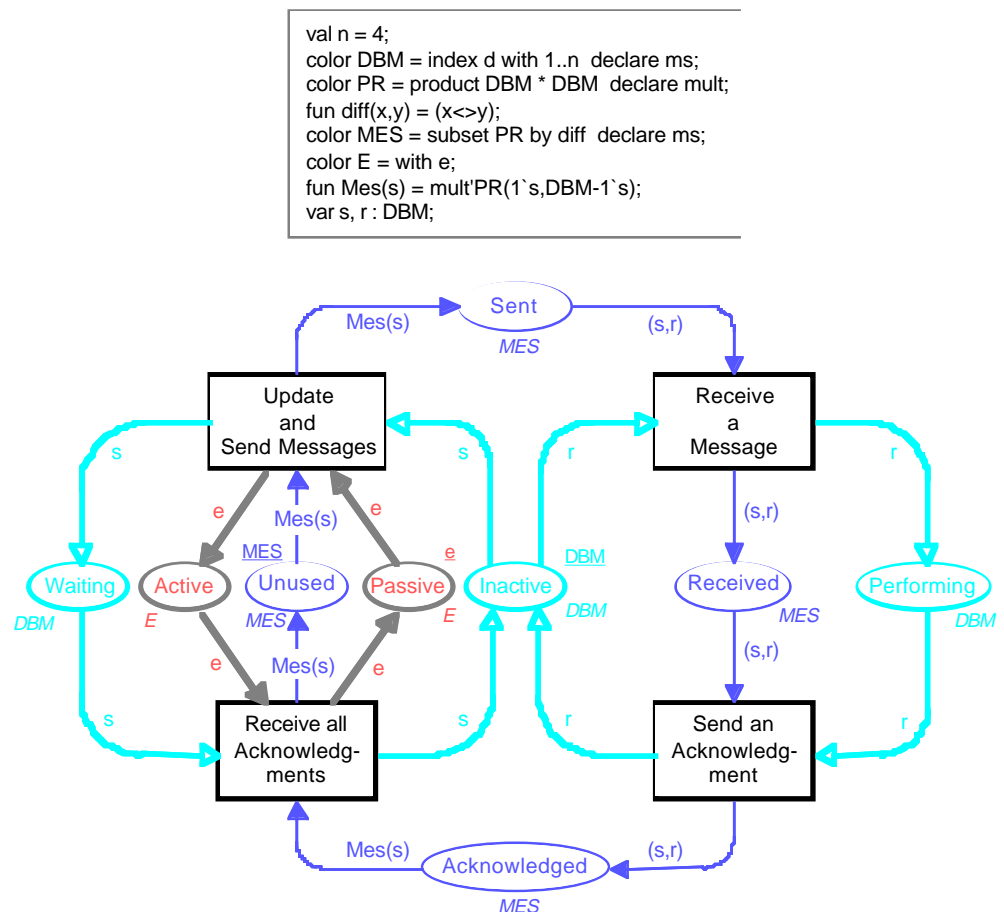
How to Write OSP-Specifications

`Spec = OSPSpec}`

Note that `OSPSpec` is used since we have an OSP-specification. Now the OSP-graph can be calculated. Since permutation symmetry specifications preserve the number of tokens on all places, i.e., a necessary condition for two markings to be equivalent is that they have the same number of tokens on each of the places there is no need to change the default encoding of markings when dealing with OSP-specifications.

Example: Distributed Data Base

We will now consider the distributed database system from Sect. 1.3 in [CPN 1]. The CP-net for the database system is shown below.



The database system uses the indexed colour set `DBM` for the database managers and the product colour set `MES` for the messages. We will show how to implement the permutation symmetry specification given on page 52 in [CPN 2]. In short, this specification can be described in the following way. The colour set `DBM` is atomic. It is assigned the symmetry group **permutation**.

The colour set MES is a structured colour set and therefore inherits its symmetry group from its base colour set which is DBM. The colour set E is also atomic and assigned the symmetry group consisting of the identity element only.

The function EquivMark expressing the equivalence on markings are as follows and will be explained below.

```
fun EquivMark (n1,n2) =
  let

    val ms1_inac = OMark.DataBase'Inactive 1
    n1;
    val ms1_inac = OMark.DataBase'Inactive 1
    n2;

    val ms1_wait = OMark.DataBase'Waiting 1
    n1;
    val ms1_wait = OMark.DataBase'Waiting 1
    n2;

    val ms1_perf =
      OMark.DataBase'Performing 1
    n1;
    val ms1_perf =
      OMark.DataBase'Performing 1
    n2;

    val ms1_unus = OMark.DataBase'Unused 1
    n1;
    val ms1_unus = OMark.DataBase'Unused 1
    n2;

    val ms1_sent = OMark.DataBase'Sent 1 n1;
    val ms1_sent = OMark.DataBase'Sent 1 n2;

    val ms1_rec = OMark.DataBase'Received 1
    n1;
    val ms1_rec = OMark.DataBase'Received 1
    n2;

    val ms1_ack =
      OMark.DataBase'Acknowledged 1
    n1;
    val ms1_ack =
      OMark.DataBase'Acknowledged 1
    n2;

    val ms1_act = OMark.DataBase'Active 1
    n1;
    val ms2_act = OMark.DataBase'Active 1
    n2;

    val ms1_pas = OMark.DataBase'Passive 1
    n1;
```

How to Write OSP-Specifications

```
val ms2_pas = OEMark.DataBase'Passive 1
n2;

val cands =
  (if (PermExists ms1_inac ms2_inac)
  andalso
    (PermExists ms1_wait ms2_wait)
  andalso
    (PermExists ms1_perf ms2_perf)
  andalso
    (PermExists ms1_unus ms2_unus)
  andalso
    (PermExists ms1_sent ms2_sent)
  andalso
    (PermExists ms1_rec ms2_rec) andalso
    (PermExists ms1_ack ms2_ack) andalso
    (ms1_act == ms2_act) andalso
    (ms1_pas == ms1_pas)
  then
    ListPermutations (
      IntersectResSets
        [CreateResSet ms1_inac
ms2_inac,
        CreateResSet ms1_wait
ms2_wait,
        CreateResSet ms1_perf
ms2_perf],
      DBM)
  else
    [])

in
  TestPermutationsPair (cands,
    [(ms1_unus,ms2_unus),
     (ms1_sent,ms2_sent),
     (ms1_rec,ms2_rec),
     (ms1_ack,ms2_ack)])

end;
```

The first part of the body defines a number of values because of two objectives. We do not want to access the internal representation of markings more than once (for efficiency reasons) and we want shorter names for the markings of the places in the rest of the code.

Given two nodes `n1` and `n2`, it is first tested whether there might exist a permutation symmetry which maps the marking of `n1` into the marking of `n2`. This is done using the functions `PermExists`. The functionality of `PermExists` is as follows.

```
fun PermExists    'a ms -> 'b ms -> bool
```

PermExists takes two multi-set *m1* and *m2* and determines whether there exists a permutation which maps *m1* into *m2*. **PermExists** uses the coefficient multi-set to determine whether this is the case or not. For atomic colour sets assigned the symmetry group consisting of all permutations, **PermExists** expresses a necessary and sufficient condition for a permutation symmetry to exist. For structured colour sets **PermExists** only expresses a necessary condition. For the places with colour set *E* it is simply tested whether they contain the same multi-set.

If the check above is successful (i.e, a permutation symmetry might exist) **restriction sets** as outlined on pages 95-96 in [CPN 2] are exploited. For each place with colour set *DBM*, a restriction set is created. This is done using the function **CreateResSet**:

```
fun CreateResSet `a ms -> `a ms ->
                                `a ResSet
```

CreateResSet takes two multi-set *s* as argument and if the two multi-sets can be mapped into each other, the corresponding restriction set is returned. The intersection of all the restriction set obtained is then calculated using the function **IntersectResSets**:

```
fun IntersectResSets `a ResSet list ->
                                `a ResSet
```

IntersectResSets takes a list of restriction sets and calculates the intersection.

We are now done with the places with colour set *DBM*. To check the remaining places with colour set *MES* we do a naive testing of all permutations that are candidates to map *n1* into *n2*. First we supply the restriction set obtained from the places with colour set *DBM* to the function **ListPermutations**:

```
fun ListPermutations `a ResSet * `a ms
    -> `a Perm list
```

ListPermutations takes a restriction set over some set *A* and the set *A* itself (represented as a multi-set) and creates the list of permutations determined by the restriction set. The list of permutation obtained in this way is now used as argument to the function **TestPermutationsPair** together with the marking of the places with colour set *MES* in *n1* and *n2*. The functionality of **TestPermutationsPair** is as follows:

```
fun TestPermutationsPair `a Perm list
    * ((`a * `a) ms * (`a * `a) ms) list ->
    bool
```

How to Write OSP-Specifications

TestPermutationsPair takes a list of permutations and a list of pairs of multi-sets over a product colour set . It returns true if one of the permutations maps the first component to the second component of each of the multi-set pairs. It returns false otherwise. The first time it finds a permutation that works for all pairs in the list, it stops.

We will now show how to write **EquivBE**. It must capture that a necessary condition for two binding elements to be equivalent is that the transitions of the two binding elements are identical. So assume that the two transitions in the binding elements are equal. If we inspect the CP-net we can split the transitions into two groups depending on the number of variables. If the transition has one variable which will be of colour set DBM then the binding elements can be mapped into the other. If the transition has two variables which both will be of colour set DBM we require that if they are bound to identical values in one of the bindings then the same must hold for the second. The **EquivBE** functions is shown below.

```
fun EquivBE (
  Bind.DataBase'Update (1,_),
  Bind.DataBase'Update (1,_)) = true
| EquivBE (
  Bind.DataBase'AResceive (1,_),
  Bind.DataBase'AResceive (1,_)) = true
| EquivBE (
  Bind.DataBase'MResceive
(1,{s=s1,r=r1}),
  Bind.DataBase'MResceive
(1,{s=s2,r=21}))
= (s1=r1) = (s2=r2)
| EquivBE (
  Bind.DataBase'Send (1,{s=s1,r=r1}),
  Bind.DataBase'Send (1,{s=s2,r=21}))
= (s1=r1) = (s2=r2)
| EquivBE (_,_) = false;
```

Above we have introduced a number of utility functions which support restriction sets and permutations . To make the description of the utility functions related to restriction sets complete we will list the remaining ones.

```
fun TestPermutations `a Perm list *
    (`a ms * `a ms) list ->
bool
```

TestPermutations takes a list of permutations and two multi-sets m1 and m2 and determines whether there exists a permutation in the supplied list which maps m1 into m2.

```
fun FilterPermutations `a Perm list *
```

```
('a ms * 'a ms) list -> 'a Perm list
```

This function works like `TestPermutations` above but instead the list of permutations that work is returned.

```
fun FilterPermutationsPair 'a Perm  
list * (('a * 'a) ms * ('a * 'a ms) list  
->  
      'a Perm list
```

FilterPermutationsPair works like `FilterPermutations` above except that it is a multi-set of of a product colour.

```
fun ApplyPermutation      'a Perm -> 'a ms  
                           -> 'a ms  
  
fun ApplyPermutationPair 'a Perm *  
      ('a * 'a) ms -> ('a * 'a) ms
```

These two functions both takes a permutation and a multi-set and applies the permutation on to the supplied multi-set.

ApplyPermutaionsPair works on multi-sets over some product colour whereas **ApplyPermutations** works directly on the multi-set, i.e, it does not go into the structure elements in the multi-set.

Chapter 5

How to Make Standard Queries

This chapter explains how to perform standard queries to investigate the properties of a CPN model. It is, e.g., possible to investigate the reachability, boundedness, home, liveness and fairness properties using the OE/OS-graph.

The standard query functions available depends upon the class of the specification, i.e, whether it is an equivalence specification (OE-specification), a symmetry specification (OS-specification), or a permutation symmetry specification (OSP-specification). This is because OSP-specification have stronger proof rules than OS-specifications which again has stronger proof rules than OE-specification. This relationship is also reflected in the following sections. For each of the properties (reachability, boundedness, home, liveness and fairness) we first consider the standard query functions available for OE-specifications, then the additional standard queries for OS-specifications and finally the additional standard query functions available for OSP-specifications. If a standard query functions which is not supported for the given class of equivalence specification is invoked, the exception `std_query_not_avail` will be raised.

It is also important to notice, that all proof rules depend on the consistency of the supplied equivalence specification.

The query functions are typically used in auxiliary boxes – alone or as part of a larger ML expression. The box is evaluated by means of the **ML Evaluate** command. If you select a non-empty part of the text, ML Evaluate only deals with that part.

Reachability Properties

OE-specifications

The query functions for reachability properties are based on Prop 2.6 in [CPN 2].

fun Reachable	Node * Node -> bool
fun SccReachable	Node * Node -> bool
fun AllReachable	unit -> bool

Reachable determines whether there exists occurrence sequences from all markings of the first node leading to some marking in the second node. For the resource allocation system:

```
Reachable(5,10)
```

returns true. This tells us that there exist occurrence sequences from any marking equivalent with the representative of node 5 to some marking equivalent with the representative of node 10.

SccReachable return the same result as **Reachable**, but it uses the Scc-graph.

AllReachable implements Prop 2.6 (iv) in [CPN 2]. The function returns true if for all pairs of markings equivalent with a reachable marking there exist an occurrence sequence from the first marking to a marking equivalent with the second marking. For the resource allocation system:

```
AllReachable ()
```

return true. It should be noted that Prop 2.6 (iv) only expresses a sufficient condition. Hence if the function return false the negation of the above assertion cannot be concluded.

OS-specifications

The query functions for reachability properties are based on Prop 3.7 in [CPN 2].

fun ReachableSym	Node * Node * (Node -> bool) -> bool
fun SccReachableSym	Node * Node * (Node -> bool) -> bool

Reachable and **SccReachable** now also determines whether there exists occurrence sequences to all marking in the

How to Make Standard Queries

second node starting in some marking of the first node. For dining philosopher system:

```
Reachable (3,2)
```

return true. This tells us that there exist occurrence sequence from all markings of node 3 leading to some marking in node 2. It also tells us that all markings equivalent with the representative of node 1 can be reached from some marking equivalent with the representative of node 2.

ReachableSym implements Prop 3.7 (vi) in [CPN 2]. The function returns true if there exists occurrence sequence between all markings of the first node to all marking of the second node. The function uses a predicate which must be provided by the user. The predicate the user must provide is the predicate **Sym** from page 75 in [CPN 2]. In this manual we will refer to this predicate as a **SymPredicate**. This predicate is true on a node if the size of the equivalence class which the node represent is 1 and false otherwise. An example of this will be given below. If **ReachableSym** return false it cannot be concluded that no occurrence sequence between any two markings in the nodes, since Prop 3.7 (iv) is only a sufficient condition .

SccReachableSym is similar to **ReachableSym** except that it uses the Scc-graph.

AllReachable now determines whether there exist an occurrences sequence between all pairs of reachable markings.

OSP-specifications

There are no additional standard queries functions since the proof rules are identical to the ones for OS-graphs. Instead we will show how to write the **SymPredicate** used by **ReachableSym** for the dining philosopher system. This function is a simple modification of the **EquivMark** function and is shown below.

```
fun SymPhil n1 =
let
  fun alltrue [] = true
    | alltrue (x::xs) =
      (x andalso (alltrue xs))
in
  (alltrue (map
    (fn (x1,x2) =>
      (TestRotation (fst x)
        [(OEMark.System'Think 1 n1,
          OEMark.System'Think 1 n1),
         (OEMark.System'Eat 1 n1,
          OEMark.System'Eat 1 n1)]))
    (snd x1, snd x2)))
```

```

                                andalso
                                (TestRotation (snd x)
                                 [(OEMark.System'Unused 1 n1,
                                   OEMark.System'Unused 1
                                   n1)]))
                                rotations
                                end;

```

The difference compared to `EquivMark` is that we now find all the rotations which map the representative of the given node to itself. If this is all the rotations then this representative can only be equivalent to itself and hence the equivalence class to which the representative belongs will only consist a single marking . The auxiliary function `alltrue` tests whether all elements in a boolean list is true. For the dining philosporer system:

```
ReachableSym( 3, 5, SymPhil)
```

returns true. This tells us that for all pairs of markings where the first marking belongs to the equivalence class of node 3 and the second marking belongs to the equivalence class of node 5 there exist an occurrence equence from the first marking to the second marking.

Boundedness Properties

OE-specifications

The query functions for boundedness properties are based on Prop 2.7 in [CPN 2].

<code>fun UpperInteger</code>	<code>(Node -> 'a ms) -> int</code>
<code>fun LowerInteger</code>	<code>(Node -> 'a ms) -> int</code>
<code>fun UpperMultiset</code>	<code>(Node -> 'a ms) -> 'a ms</code>
<code>fun LowerMultiset</code>	<code>(Node -> 'a ms) -> 'a ms</code>

UpperInteger uses a specified function `F` of type:

```
Node -> 'a ms
```

to calculate an integer $|F(n)|$. This is done for each node `n` in the OE/OS-graph, and the maximum of the calculated integers is returned. Since only the representatives for the equivalence classes is traversed the `UpperInteger` will be less than or equal to the best upper integer bound.

How to Make Standard Queries

LowerInteger is analogous to `UpperInteger`, but returns the minimal value of the integers $|F(n)|$.

UpperMultiSet is analogous to `UpperInteger`, but it calculates $F(n)$ instead of $|F(n)|$. The result is the smallest multi-set which is larger than or equal to all the calculated multi-sets. Similar to `UpperInteger` the returned multi-set will be less than or equal to then best upper multi-set bound.

LowerMultiSet is analogous to `UpperInteger`, but returns the largest multi-set which is smaller than or equal to all the calculated multi-sets.

OS-specifications

There are no additional standard query functions

OSP-specifications

<code>fun BestUpperMultiSet</code>	<code>(Node -> 'a ms) 'a ms list -> 'a ms</code>
<code>fun BestLowerMultiSet</code>	<code>(Node -> 'a ms) 'a ms list -> 'a ms</code>

BestUpperMultiSet determines the best upper multi-set bound for a given place. It implements Prop. 3.18 (ii). The function takes the equivalence classes for the colour set associated with the place as argument. For the dining philosophers system and the place Think:

```
mkst_ms'PH (BestUpperMultiSet (  
                                OEmark.System'Think  
1,[PH]));
```

returns the multi-set:

```
ph(1)+ph(2)+ph(3)+ph(4)+ph(5)
```

Since the colour set PH consists of only a single equivalence class (ph(i) can always be mapped to ph(j) by a rotation) the list specifying the equivalence classes consist only of a single element.

BestLowerMultiSet is analogous to `BestUpperMultiSet` except that the best lower multi-set bound is returned.

For OSP-graphs, the query functions **UpperInteger** and **LowerInteger** return the best upper and best lower integer bound respectively. This is because permutation symmetry specifications preserve the number of tokens on places.

Home Properties

OE-specifications

The query functions for home properties are based on Prop 2.8 in [CPN 2]. It should be noted that most items in Prop 2.8 only express necessary conditions. Hence if a function returns true this should be interpreted as *possible*. False is interpreted in the usual way.

fun HomeSpace	Node list -> bool
fun MinimalHomeSpace	unit -> int
fun HomeMarking	Node -> bool
fun ListHomeMarkings	unit -> Node list
fun ListHomeScc	unit -> Scc
fun HomeMarkingExists	unit -> bool
fun InitialHomeMarking	unit -> bool

HomeSpace determines whether the markings of the specified list of nodes is a home space. For the resource allocation system:

```
HomeSpace [1,2]
```

returns true. This tells us that the union of the set of markings of the two equivalence classes of node 1 and node 2 constitutes a home space.

MinimalHomeSpace returns the minimal number of equivalence classes which is needed to form a home space. This is identical to the number of terminal strongly connected components.

HomeMarking determines whether it is possible for each marking in the node to be a home marking. For the resource allocation system:

```
HomeMarking 1
```

How to Make Standard Queries

returns true. This tells us that each marking in the equivalence class of node 1 is a potential home markings

ListHomeMarkings returns a list with all those nodes whose markings are potential home markings. For the resource allocation system:

```
ListHomeMarking ( )
```

returns the list [1,2,3,4,5,6,7,8,9,10,11,12,13]. This tells us that all markings equivalent with a reachable marking are candidates for being home markings.

ListHomeScc is similar to **ListHomeMarkings**, but the result is given in a more compact way. The result is either a single Scc (and then the possible home markings are exactly those markings that belong to the Scc) or the result is zero (and then there are no home markings).

HomeMarkingExists determines whether the CP-net possibly has any home markings. This is the case if there is exactly one terminal strongly connected component.

InitialHomeMarking determines whether the initial marking of the CP-net is a possible home marking. This is the case if there is exactly one strongly connected component.

OS-specifications

The query functions for home properties are based on Prop. 3.9 in [CPN 2]

<code>fun HomeSpaceSym</code>	<code>Node list * (Node -> bool) -> bool</code>
<code>fun HomeMarkingSym</code>	<code>Node -> (Node -> bool) -> bool</code>

HomeSpaceSym returns true if any set of markings in which a marking from each of the nodes in the list constitutes a home space. The functions corresponds to Prop. 3.9 (v). The functions uses a **SymPredicate** which must be provided by the user.

HomeMarkingSym returns true if all markings specified by the node are home markings. Like the function **HomeSpaceSym** the user must provide a **SymPredicate**.

InitialHomeMarking now determines whether the initial marking is a home marking or not.

OSP-specifications

There are no additional standard queries. Instead we will show how to write the predicate used by the functions `HomeSpaceSym` and `HomeMarkingSym` for the database example. To ease the implementation of the predicate some useful functions working on **component sets** are provided by the tool (see [CPN 2] page 96). The function `SymDataBase` implementing the predicate are as follows and will be explained below.

```
fun SymDataBase n1 =
  let

    val ms_inac = OEMark.DataBase'Inactive 1
    n1;
    val ms_wait = OEMark.DataBase'Waiting 1
    n1;
    val ms_perf =
      OEMark.DataBase'Performing 1
    n1;
    val ms_unus = OEMark.DataBase'Unused 1
    n1;
    val ms_sent = OEMark.DataBase'Sent 1 n1;
    val ms_rec = OEMark.DataBase'Received 1
    n1;
    val ms_ack =
      OEMark.DataBase'Acknowledged 1
    n1;

    val cands =
      ListPermutations (CompSetToResSet
        IntersectCompSets
        [CreateResSet ms_inac,
         CreateResSet ms_wait,
         CreateResSet ms_perf,
         DBM])

  in
    ((length (FilterPermutationsPair
      (cands,
       [(ms_unus,ms_unus),
        (ms_sent,ms_sent),
        (ms_rec,ms_rec),
        (ms_ack,ms_ack)]))) = (fac
n))
  end;
```

The function is a modification of the `EquivMark` function for the database system. The first part of the body defines a number of values for easy reference. For each place with colour set DBM a

How to Make Standard Queries

component set is created. This is done using the function `CreateCompSet`:

```
fun CreateCompSet    `a ms -> `a CompSet
```

CreateCompSet takes a multi-set and creates the corresponding component set. The intersection of the component set obtained is then calculated using the function `IntersectCompSets`:

```
fun IntersectCompSets `a CompSet list  
-> `a CompSet
```

IntersectCompSets takes a list of component sets and calculates the intersection. To check the remaining places with colour set MES, the component set obtained is turned into a restriction set using the function `CompSetToResSet`:

```
fun CompSetToResSet `a CompSet -> `a  
ResSet
```

The set of permutations determined by the restriction set is then obtained using `ListPermutations`.

Finally it is tested whether the set of permutations thus obtained consists of all colour symmetries for the colour set DBM (the function `fac` implements the factorial function). If this is the case, then the equivalence class determined by the node is of size one.

For the database system:

```
HomeSpace(1,2,3,PhilSym)
```

returns true. This tells us that if we pick a marking from each of the nodes 1,2, and 3, then this set of markings will constitute a home space.

```
HomeMarkingSym(3,SymDataBase)
```

returns true. This tells us that any marking in node 3 is a home marking.

Liveness Properties

OE-specifications

The query functions for liveness properties are based on Prop 2.9 in [CPN 2].

<code>fun DeadMarking</code>	<code>Node -> bool</code>
<code>fun ListDeadMarkings</code>	<code>unit -> Node list</code>
<code>fun SccListDeadMarkings</code>	<code>unit -> Node list</code>
<code>fun TIsLive</code>	<code>TI.TransInst list -> bool</code>

DeadMarking determines whether all markings in the specified node are dead. For the resource allocation system:

```
DeadMarking 1
```

returns false. This tells us that none of the markings in the equivalence class of node 1 are dead. Either all markings in an equivalence class are dead or none of them are.

ListDeadMarkings returns a list with all those nodes whose markings are dead. For the resource allocation system;

```
ListDeadMarking ()
```

return the empty list. This tells us that no marking equivalent with a reachable marking is dead.

SccListDeadMarkings returns the same result as **ListDeadMarkings**, but it uses the Scc-graph.

TIsLive determines whether a transition instance is live. The function implements Prop. 2.9 (v) in [CPN 2] and can therefore only be used for with equivalence specifications in which equivalent binding elements are guaranteed to belong to the same transition instance. This is the case with for instance the resource allocation system for which:

```
TIsLive(TI.System'Take 1)
```

returns true.

OS-specifications

The query functions for liveness properties are based on Prop 3.10 in [CPN 2].

How to Make Standard Queries

<code>fun BESLiveSym</code>	<code>Bind.Elem list * (Node -> bool) -> bool</code>
<code>fun BESLiveScc</code>	<code>Bind.Elem list -> bool</code>

BESLiveSym returns true if the set of binding elements provided list is live. Similar to `HomeSpaceSym` the user must provide a `SymPredicate`. For the dining philosopher system:

```
BESLiveSym([Bind.System'Take  
{1,{p=ph(1)}}], SymPhil)
```

returns true. This tells us that this binding element is live.

BESLiveScc returns true if the list of binding elements is live. The query function can only be used in the special case in which the Scc-graph consist of only a single node. If the query function is used when the Scc-graph for the OE/OS-graph has more than one node the exception `more_than_one_scc` will be raised.

OSP-specifications

The query functions for liveness properties are based on Prop 3.19 in [CPN 2].

<code>fun TISStrictlyLiveSym</code>	<code>TI.TransInst * Bind.Elem list * (Node -> bool) -> bool</code>
<code>fun TISStrictlyLive</code>	<code>TI.TransInst * (Node -> bool) -> bool</code>
<code>fun TISStrictlyLiveScc</code>	<code>TI.TransInst * (Node -> bool) -> bool</code>

TISLive determines whether the transition instance is live. The requirement stated above for `TISLive` when used for OE-specifications is automatically fulfilled for OSP-specifications.

TISStrictlyLiveSym returns true if the transition instance is strictly live. The set of binding elements for the transition must be provided. For the dining philosopher system:

```
TISStrictlyLiveSym (TI.System'Take,  
[Bind.System'Take (1,{p=ph(0)}),  
Bind.System'Take (1,{p=ph(1)}),  
Bind.System'Take (1,{p=ph(2)}),
```

```
Bind.System'Take (1, {p=ph(3)}),  
Bind.System'Take (1, {p=ph(4)})],  
PhilSym)
```

returns false. However since Prop 3.19 only is a sufficient condition we cannot conclude that the transition instance is not strictly live.

TIsStrictlyLive determines whether the transition instance is strictly live. The function can only be used in situations in which all binding elements for the transition are equivalent. This is for instance the case with all transitions in the dining philosophers system. The functions requires a SymPredicate. For the dining philosophers system:

```
TIsStrictlyLiveBE ((TI.System'Take,  
[Bind.System'Take (1, {p=ph(0)}),  
Bind.System'Take (1, {p=ph(1)}),  
Bind.System'Take (1, {p=ph(2)}),  
Bind.System'Take (1, {p=ph(3)}),  
Bind.System'Take (1, {p=ph(4)})],  
PhilSym)
```

returns true. This tells us that instance one of the transition Take is strictly live.

TIsStrictlyLiveScc determines whether the transition instance is strictly live in the case in which all binding elements for the transition are equivalent and the Scc-graph for the OSP-graph consist of only a single node. If the query function is used in a situation in which the Scc-graph has more than one node the exception `more_than_one_scc` will be raised.

Fairness Properties

OE-specifications

No standard query functions are available.

OS-specifications

No standard query functions are available.

OSP-specifications

The query functions for fairness properties are based on Prop 3.20 in [CPN 2].

<code>fun TIsFairness</code>	<code>TI.TransInst -> FairnessProperty</code>
-------------------------------------	------------------------------------------------------

How to Make Standard Queries

<code>fun ListImpartialTIs</code>	<code>unit -> FairnessProperty</code>
<code>fun ListFairTIs</code>	<code>unit -> FairnessProperty</code>
<code>fun ListJustTIs</code>	<code>unit -> FairnessProperty</code>

The type **FairnessProperty** has the following four elements:

`{Impartial, Fair, Just, No_Fairness}`.

TIsFairness determines whether the transitions instance is impartial, fair or just.

ListImpartialTIs returns a list with those transition instances that are impartial.

ListFairTIs and **ListJustTIs** are analogous to **ListImpartialTIs** except that they list those transition instances that are fair and just, respectively.

How to Make Your Own Queries

Like in the OG tool it is possible for the user to implement his own queries. For this purpose the same functions are available as for the OG tool. The reader is encouraged to consult Chap. 5 in [OG] for more information and examples of how to write such queries.

Reference List

- [CPN 1] K. Jensen: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science, Springer-Verlag, 1992. ISBN: 3-540-60943-1.
- [CPN 2] K. Jensen: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods*. Monographs in Theoretical Computer Science, Springer-Verlag, 1994. ISBN: 3-540-58276-2
- [OG] K. Jensen, S. Christensen 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/>