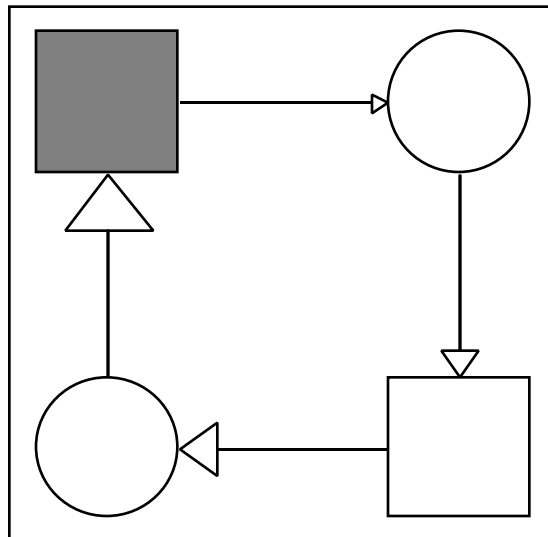


Design/CPN Occurrence Graph Manual

Version 3.0



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

© 1996 University of Aarhus

© 1996 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: Kurt Jensen, Søren Christensen and Lars M. Kristensen.

Design/CPN is a trademark of Meta Software Corporation.

Macintosh is a registered trademark of Apple Computer, Inc.

Design/CPN Occurrence Graph Manual

Version 3.0

Table of Contents

Chapter 1

Introduction to Occurrence Graphs

The History of the Design/CPN Occurrence Graph Tool.....	9
Example: Dining Philosophers	10

Chapter 2

How to Calculate an Occurrence Graph

Generation of Occurrence Graph Code	13
Details and Limitations.....	14
Generation of the Occurrence Graph	14
Standard Report.....	15

Chapter 3

How to Refer to the Items of an Occurrence Graph

Nodes, Arcs and Strongly Connected Components.....	17
Place Instances.....	17
Transition Instances.....	18
Markings.....	19
Binding Elements.....	20
String Representations.....	21
Time Values.....	22

Chapter 4

How to Make Standard Queries

Reachability Properties.....	23
Boundedness Properties.....	25
Home Properties.....	27
Liveness Properties.....	29
Fairness Properties.....	32

Chapter 5

How to Make Your Own Queries

Nodes and Arcs.....	35
Strongly Connected Components.....	36
SearchNodes.....	37
PredNodes and EvalNodes.....	40
Examples of SearchNodes Calls.....	41
SearchArcs.....	43
Examples of SearchArcs Calls.....	44
SearchSccs.....	44
Examples of SearchSccs Calls.....	45

Chapter 6

How to Draw an Occurrence Graph

Object Types	47
Occ Menu	49
Calculate Occ Graph.....	49
Calculate Successors.....	50
Calculate Scs Graph.....	50
Show Statistics.....	50
Save Report.....	51
Occ State to Sim.....	51
Sim State to Occ.....	51
Attributes/Options.....	51
Display Node.....	52
Display Arc.....	52
Display Successors.....	52
Display Predecessors.....	53
Display Scs Graph.....	53
Toggle Descriptor.....	53
Update Node.....	53

Chapter 7

How to Change Attributes and Options

Attributes/Options Command	55
Attributes	57
String Representation Options	58
Node and Arc Descriptor Options	60
Successor/Predecessor Options	61
Stop Options	61
Branching Options	62
Inspection Options	63
 Reference List	 65

Index

- AllReachable 24
- arc 17
- arc descriptor option 60
- Arcs 35
- ArcsInPath 35
- ArcToBE 20
- ArcToScc 36
- ArcToTI 20
- attribute 55
- attribute default 57
- Attributes/Options command 51, 55

- BESDead 30
- BESFairness 32
- BESLive 31
- BESStrictlyLive 31
- BEToTI 20
- Bind 19
- binding element 19
- boundedness property 16, 25
- branching option 62

- Calculate Occ Graph command 15, 49
- Calculate Scc Graph command 15, 50
- Calculate Successors command 50
- CalculateOccGraph 15
- CalculateSccGraph 15
- chatty version 23
- combination function 38
- CreationTime 21

- DeadMarking 29
- DestNode 35
- diagram default 55
- dining philosopher 10
- Display Arc command 52
- Display Node command 52
- Display Predecessors command 53
- Display Scc Graph command 53
- Display Successors command 52
- DisplayArcPath 52
- DisplayArcs 52
- DisplayNodePath 52
- DisplayNodes 52

- EqualUntimed 22
- EqualsUntimed 22

- EntireGraph 38
- EntireGraphCalculated 50
- EvalAllArcs 43
- EvalAllNodes 41
- EvalAllScCs 45
- EvalArcs 43
- EvalNodes 41
- EvalScCs 45
- evaluation function 37

- fairness property 32
- FairnessProperty 32
- FullyProcessed 36

- generation of occurrence graph 14
- generation of occurrence graph code 13

- home property 16, 27
- HomeMarking 28
- HomeMarkingExists 28
- HomeSpace 27

- InArcs 35
- Initial HomeMarking 28
- InitNode 17
- InitScc 17
- InNodes 35
- inspection option 63
- Inst 17

- ListDeadMarkings 29
- ListDeadTIs 30
- ListFairTIs 33
- ListHomeMarkings 28
- ListHomeScc 28
- ListImpartialTIs 33
- ListJustTIs 33
- ListLiveTIs 31
- liveness property 16, 29
- LowerInteger 25
- LowerMultiSet 26

- Mark 19
- marking 19
- MinimalHomeSpace 27
- ML Evaluate 23
- ML Evaluate command 15, 56

Occurrence Graph Manual

- node 17
- node descriptor option 60
- NodesInPath 35
- NodeToScc 36
- NoLimit 39
- NoOfArcs 50
- NoOfNodes 50
- NoOfSecs 50

- object type 47
- Occ menu 49
- Occ State to Sim command 51
- occurrence graph code 13
- OccurrenceTime 21
- OG arc 47
- OG arc descriptor 47
- OG node 47
- OG node descriptor 47
- option 55
- OutArcs 35
- OutNodes 35

- PI 17
- PI.All 18
- place instance 17
- PredAllArcs 43
- PredAllNodes 41
- PredAllSccs 45
- PredArcs 43
- predicate function 37
- PredNodes 40
- PredSccs 45
- Processed 36

- reachability property 23
- Reachable 24

- Save Report command 15, 51
- Scc 17
- SccArcs 36
- SccArcsInPath 37
- SccDestNode 36
- SccGraphCalculated 51
- SccInArcs 36
- SccInNodes 36
- SccListDeadMarkings 30
- SccNodesInPath 37
- SccNoOfArcs 51
- SccNoOfNodes 51
- SccNoOfSecs 51
- SccOutArcs 37
- SccOutNodes 36
- SccReachable 24
- SccSourceNode 36
- SccTerminal 37
- SccToArcs 36
- SccToNodes 36
- SccTrivial 37
- search area 37
- search limit 37
- SearchAllArcs 43
- SearchAllNodes 41
- SearchAllSccs 45
- SearchArcs 43
- SearchNodes 37
- SearchReachableArcs 43
- SearchReachableNodes 41
- SearchReachableSccs 45
- SearchSccs 44
- Show Statistics command 50
- Sim State to Occ command 51
- SourceNode 35
- standard report 15
- start value 38
- statistics 15
- stop option 61
- string representation 21
- string representation option 58
- StripTime 22
- strongly connected component 17
- st_Arc 20
- st_BE 20
- st_Mark 21
- st_Node 20
- st_PI 20
- st_TI 20
- successor/predecessor option 61
- system default 55

- Terminal 35
- TI 18
- TI.All 18
- timed occurrence graph 21
- TIsDead 30
- TIsFairness 32
- TIsLive 31
- Toggle Descriptor command 47, 53
- transition instance 18

- Update Node command 53
- UpperInteger 25
- UpperMultiSet 26

Chapter 1

Introduction to Occurrence Graphs

The History of the Design/CPN Occurrence Graph Tool

This manual describes a tool to calculate, analyse and draw occurrence graphs (also called state spaces, reachability graphs or reachability trees).

The original version of the Design/CPN occurrence graph tool (OG tool) was designed and implemented in 1991-92. It was developed by Meta Software, Cambridge MA, USA, in close cooperation with researchers from the Computer Science Department of University of Aarhus, Denmark.

The present version of the OG tool has been developed at University of Aarhus in 1995. It improves and extends the original occurrence graph tool in several ways.

The OG tool is now fully integrated with Design/CPN. This means that you can switch between the editor/simulator and the OG tool. When an occurrence graph node has been found, it 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 occurrence graph or used as the initial marking for a new occurrence graph.

The new version of the OG tool has a large number of built-in standard queries. They can be used to investigate all 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 you to formulate your own, non-standard queries. The standard queries require no programming at all. The non-standard queries usually requires that you write 2-5 lines of quite straightforward ML code.

Occurrence Graph Manual

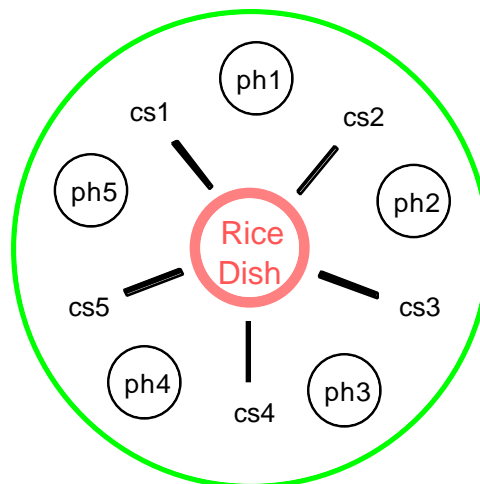
The ML interface has been totally rewritten. The naming system and the typing system have been redesigned and simplified. This means that the tool now is easier and more straightforward to use. The manual has been totally rewritten, and a lot of examples have been added.

The new version of the OG tool is much easier to use than the old one – and it should present few problems to people who are familiar with CP-nets and Design/CPN. To use the OG tool, the user simply enters the simulator and invokes the **Enter Occ Graph** command (in the **File** menu). This has a similar effect as **Enter Simulator**. It creates the occurrence graph code, i.e., the ML code necessary to calculate, analyse and draw occurrence graphs. Moreover, it creates a new menu, called **Occ**. This menu contains all the commands which are used to perform the calculation and drawing of occurrence graphs

Example: Dining Philosophers

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 of [CPN 2].

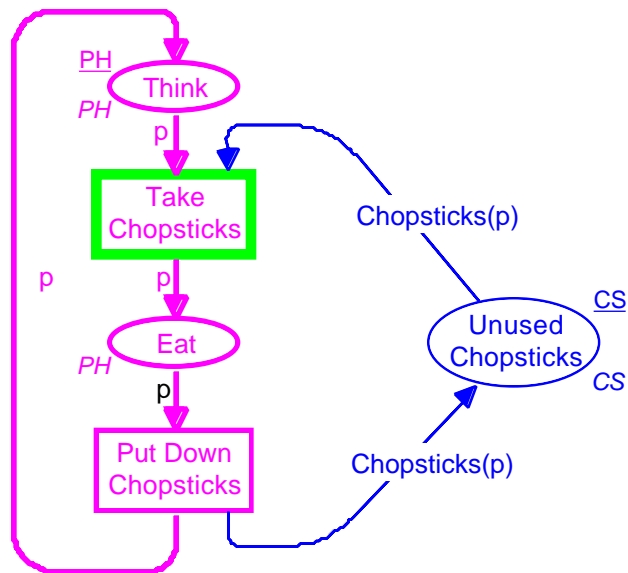
In this manual we use the dining philosopher system as our main example. Five Chinese philosophers are sitting around a circular table. In the middle of the table there is a delicious dish of rice, and between each pair of philosophers there is a single chopstick. Each philosopher alternates between 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.



Introduction to Occurrence Graphs

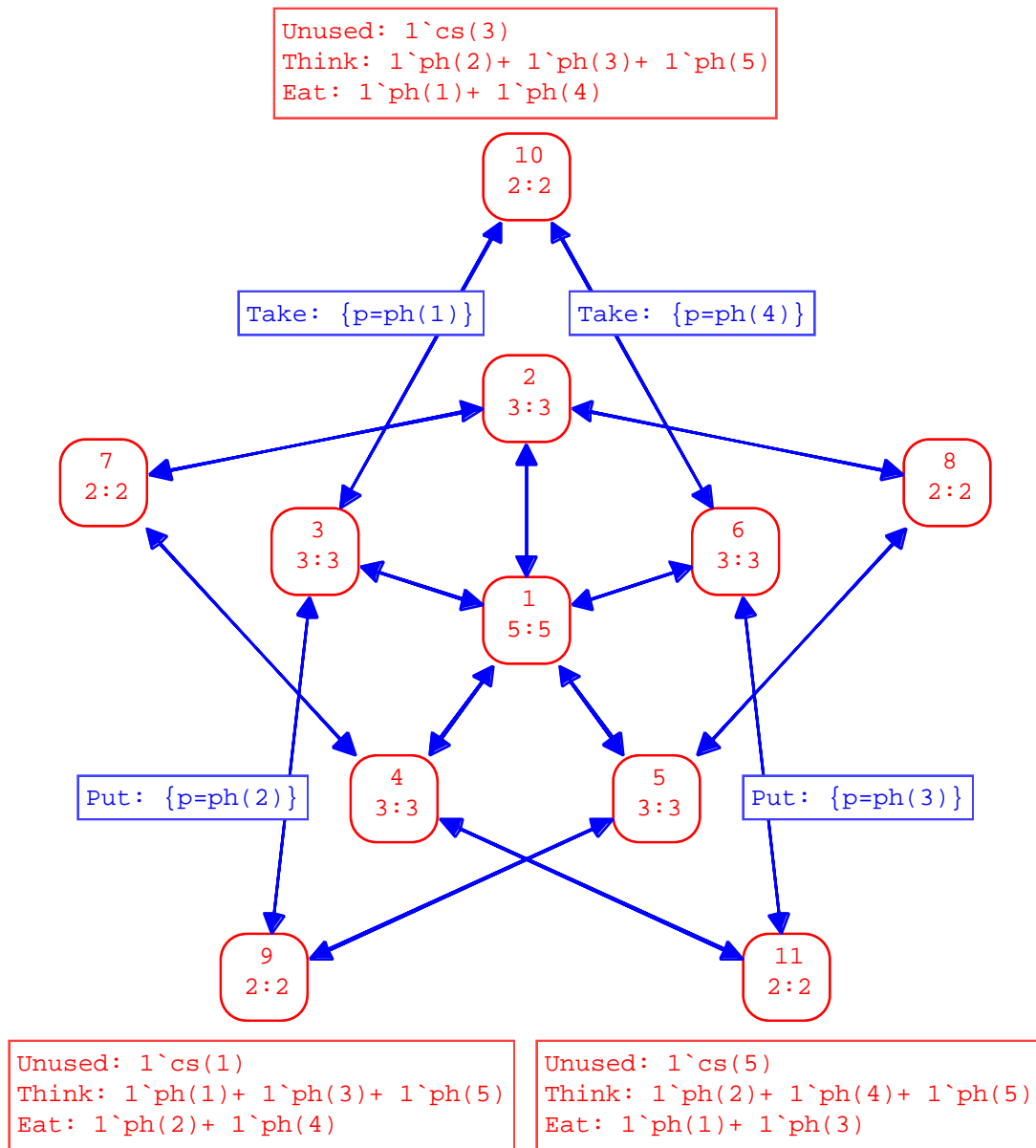
The philosopher system is modelled by the CP-net shown below. The PH colour set represents the philosophers, while the CS colour set represents the chopsticks. The function *Chopsticks* maps each philosopher into the two chopsticks next to him.

```
val n = 5;  
color PH = index ph with 1..n declare ms;  
color CS = index cs with 1..n 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).

Occurrence Graph Manual



The occurrence graph for the dining philosophers is made by means of the OG tool. In the following we shall show how this is done. However, it should be noted that this occurrence graph is rather atypical. Most occurrence graphs are much larger. The present version is able to handle graphs with 20,000-200,000 nodes and 50,000-2,000,000 arcs – provided that you have sufficient RAM in your machine. Future versions are expected to be able to handle much larger occurrence graphs.

Chapter 2

How to Calculate an Occurrence Graph

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

Generation of Occurrence Graph Code

To generate the occurrence graph code the following steps must be performed (in the specified order):

- a) Make sure that you are using Design/CPN version 3.0 (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 tell whether you want your occurrence graph to be with or without time. To choose the setting which you want 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 occurrence graph. – If you want to use the initial marking of CPN model as the initial marking of your occurrence graph, nothing needs to be done.
- f) Invoke **Enter Occ Graph** (in the File menu). This will create the occurrence graph code. For large nets it takes a while – comparable to the time for a full simulator switch.

When **Enter Occ Graph** terminates, a new **Occ 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 occurrence graphs. A detailed description of the menu and the commands can be found in Chap. 6. Most of the commands are used to draw occurrence graphs (and hence you don't need to read about them at this stage).

The generation of new nodes progresses in a *width first* fashion. This means that the nodes are being processed in the order in which they were created. To a certain extent, a depth first generation can be obtained by using "narrow" *Branching Options* (described in Chapter 7).

For a timed occurrence graph the processing order is determined by the creation time (i.e., the model time at which the individual markings start to exist).

We propose that you now try to generate the occurrence graph code for the dining philosopher system. To do this use the CPN model called "DiningPhilosophers". It can be downloaded from the Design/CPN WWW pages.

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 occurrence graph code from scratch. This also means that the occurrence graph (if any) is lost. When the modification is made in the simulator it is sufficient to invoke **Reswitch** and **Enter Occ Graph**.

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

The settings for timed simulations are also observed. Hence, it is possible to generate occurrence graphs, for timed CP-nets, with or without the time mechanism in effect. Use **General Simulation Options** to determine whether you want your occurrence graph to be timed or not.

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

Generation of the Occurrence Graph

When you have generated the occurrence graph code (by following the steps described above), you are ready to calculate the occurrence graph.

How to Calculate an Occurrence Graph

If the occurrence graph is expected to be small (e.g., with a few hundred nodes and arcs), you can simply invoke **Calculate Occ Graph**. Otherwise you may need to change the *Stop Options* and/or the *Branching Options* described in Chap. 7.

Many of the query functions in Chap. 4 use the Scc-graph (i.e., the strongly connected components of the occurrence graph). To calculate the Scc-graph you invoke **Calculate Scc Graph**.

The occurrence graph and the Scc-graph can also be calculated by using the **ML Evaluate** command to evaluate the following ML functions, which work exactly as the menu commands. This can, e.g., be useful if you want to handle exceptions raised by the CPN model or the Stop options (in Chap. 7):

```
fun CalculateOccGraph    unit -> unit
fun CalculateSccGraph    unit -> unit
```

Additional information about the menu commands mentioned above can be found in Chap. 6.

Standard Report

When you have generated the occurrence graph for a CP-net, you can use **Save Report** to generate a text file which contains a **standard report** providing information about:

- Statistics (size of occurrence graph and Scc-graph).
- Boundedness Properties (integer and multi-set bounds for place instances).
- Home Properties (home markings).
- Liveness Properties (dead markings, dead/live transition instances).
- Fairness Properties (impartial/fair/just transition instances).

The command invokes a dialogue box allowing the user to specify the kind of information which he wants to obtain. This is done by choosing one or more of the possibilities mentioned above (home and fairness properties can only be chosen if the Scc-graph has been calculated). For the dining philosopher system the full standard report looks as follows:

Statistics

```
-----
Occurrence Graph
Nodes:   11
Arcs:    30
Secs:    1
Status:  Full
```

Occurrence Graph Manual

Scc Graph
Nodes: 1
Arcs: 0
Secs: 0

Boundedness Properties

Best Integer Bounds	Upper	Lower
System'Eat 1	2	0
System'Think 1	5	3
System'Unused 1	5	1

Best Upper Multi-set Bounds

System'Eat 1	$1\text{'ph}(1) + 1\text{'ph}(2) + 1\text{'ph}(3) + 1\text{'ph}(4) + 1\text{'ph}(5)$
System'Think 1	$1\text{'ph}(1) + 1\text{'ph}(2) + 1\text{'ph}(3) + 1\text{'ph}(4) + 1\text{'ph}(5)$
System'Unused 1	$1\text{'cs}(1) + 1\text{'cs}(2) + 1\text{'cs}(3) + 1\text{'cs}(4) + 1\text{'cs}(5)$

Best Lower Multi-set Bounds

System'Eat 1	empty
System'Think 1	empty
System'Unused 1	empty

Home Properties

Home Markings: All

Liveness Properties

Dead Markings: None
Dead Transitions Instances: None
Live Transitions Instances: All

Fairness Properties

System'Put 1	Impartial
System'Take 1	Impartial

It is possible to customise the way the system displays place instances and transition instances (e.g. to replace "System'Eat 1" by "Eat"). This is done by means of the *String Representation Options* described in Chap. 7.

Chapter 3

How to Refer to the Items of an Occurrence Graph

This chapter describes how you can refer to the items of an occurrence graph, such as nodes, place instances, binding elements and markings.

Nodes, Arcs and Strongly Connected Components

We denote **nodes** and **arcs** by positive integers while we denote **strongly connected components** (Sccs) by *negative* integers:

```
type Node = int      (* positive *)  
type Arc = int       (* positive *)  
type Scc = int       (* negative *)
```

By convention 1 denotes the initial marking of the occurrence graph:

```
val InitNode = 1:Node
```

while ~1 (minus one) denotes the Scc to which node 1 belongs:

```
val InitScc = ~1:Scc
```

Place Instances

To denote **place instances** the following ML structure is available:

```
type Inst = int  
con PI.<PageName>'<PlaceName>  
                                Inst -> PI.PlaceInst
```

For the dining philosophers we use:

```
PI.System'Think 1
```

to refer to place *Think* on the *first* instance of page *System*. For the ring network from Sect. 3.1 of [CPN 1] we use:

```
PI.Site'PackNo 3
```

to refer to place *PackNo* on the *third* instance of the page *Site*.

You may want to make an alias for place instances frequently referred to, e.g.:

```
val Eat = PI.System'Eat 1
```

To denote the set of all place instances, the following notation is available:

```
PI.All           PI.PlaceInst list
```

Transition Instances

To denote **transition instances** the following ML structure is available. It is totally analogous to PI above:

```
con TI.<PageName>'<TransName>  
                                Inst -> TI.TransInst
```

For the dining philosophers we use:

```
TI.System'Take 1
```

to refer to transition *Take* on the *first* instance of page *System*. For the ring network we use:

```
TI.Site'Send 3
```

to refer to transition *Send* on the *third* instance of page *Site*.

To denote the set of all transition instances, the following notation is available:

```
TI.All           TI.TransInst list
```

Markings

To inspect the **markings** of the different place instances the following ML structure is available:

```
fun Mark.<PageName>'<PlaceName>  
                                Inst -> (Node -> CS ms)
```

where CS is the colour set of the place instance. For the dining philosophers we use:

```
Mark.System'Think 1 10
```

to refer to the multi-set of tokens on place *Think* on the *first* instance of page *System* in the marking M_{10} (by convention we use M_i to refer to the marking of node i). For the ring network we use:

```
Mark.Site'PackNo 3 217
```

to refer to the marking of place *PackNo* on the *third* instance of the page *Site* in M_{217} . It should be noted that the **Mark** function returns the internal ML representation of the multi-set. To obtain a more readable string representation the `st_Mark` function should be used (see *String Representations* below).

For a timed occurrence graph the above functions return a timed multi-set (for places with a timed colour set). For an untimed occurrence graph of a timed CPN model places with timed multi-sets have "dummy" time stamps (as in the simulator) and hence the **Mark** functions return a timed multi-set.

Binding Elements

To denote **binding elements** the following ML structure is available:

```
con Bind.<PageName>'<TransName>  
                                Inst * record -> Bind.Elem
```

where the second argument is a record specifying the binding of the variables of the transition. The type of this argument depends upon the transition. For the dining philosophers we use:

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

Occurrence Graph Manual

to refer to the binding element where transition *Take* on the *first* instance of page *System* has the variable *p* bound to *ph(3)*. For the ring network we use:

```
Bind.Site'NewPack (3,{n=2,r=S(1),s=S(3)})
```

to refer to the binding element where transition *NewPack* on the *third* instance of page *Site* has the variables *n*, *r* and *s* bound to 2, *S(1)* and *S(3)*, respectively.

The two following functions map an arc into its binding element/transition instance. The third function maps a binding element into its transition instance.

```
fun ArcToBE      Arc -> Bind.Elem
fun ArcToTI      Arc -> TI.TransInst
fun BEToTI       Bind.Elem -> TI.TransInst
```

It should be noted that

```
Bind.<PageName>'<TransName>
```

is a constructor. This means that it can be used in pattern matches. Examples can be found in *Examples of SearchArcs Calls* in Chap. 5.

String Representations

The following functions are used to obtain string representations of nodes, arcs, place instances, transition instances and binding elements:

```
fun st_Node      Node -> string
fun st_Arc       Arc -> string
fun st_PI        PI.PlaceInst -> string
fun st_TI        TI.TransInst -> string
fun st_BE        Bind.Elem -> string
```

Examples:

```
st_Node (3)                "3"
st_Arc (18)                 "18:6->10"
st_PI (PI.System'Eat 1)    "System'Eat 1"
st_TI (ArcToTI(18))        "System'Take 1"
st_BE (ArcToBE(18))        "System'Take 1: {p=ph(4)}"
```

How to Refer to the Items of an Occurrence Graph

To produce string representations of the markings of place instances the following ML structure is provided:

```
fun st_Mark.<PageName>'<PlaceName>
                                Inst -> (Node -> string)
```

Example (the produced string ends with a carriage return):

```
st_Mark.System'Eat 1 10
                                "System'Eat 1: 1`ph(1)+1`ph(4)"
```

The string representations produced by the st-functions can be modified by means of the *String Representation Options* in Chap. 7. It is, e.g., possible to get the following more compact representations (in which place instances and page names are omitted):

```
st_PI (PI.System'Eat 1)      "Eat "
st_TI (ArcToTI(18))          "Take "
st_BE (ArcToBE(18))          "Take: {p=ph(4)} "
```

Analogously, it is possible to get a compact version of st_Mark (for empty markings the result is the empty string):

```
st_Mark.System'Eat 1 10
                                "Eat: 1`ph(1)+1`ph(4)"
st_Mark.System'Eat 1 1      ""
```

Time Values

The following functions can only be used for **timed occurrence graphs**.

Each node has a time value – denoting the model time at which the marking started to exist:

```
fun CreationTime      Node -> TIME
```

Analogously, each arc has a time value – denoting the model time at which the binding element occurred:

```
fun OccurrenceTime    Arc -> TIME
```

Occurrence Graph Manual

The following function maps a timed multi-set into an untimed multi-set:

```
fun StripTime           'a tms -> 'a ms
```

The following function tells whether the markings of the two specified nodes are identical when time stamps are ignored:

```
fun EqualUntimed       Node * Node -> bool
```

The following function maps a node into all those nodes which have the same marking when time stamps are ignored:

```
fun EqualsUntimed      Node -> Node list
```

Most occurrence graphs with time will be non-cyclic (since all nodes in a cycle must have the same creation time).

Chapter 4

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 defined in [CPN 1]. Many of the query functions return results which already are included in the standard report described in Chap. 2.

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.

Some of the functions also have a **chatty version** which returns the same result as the ordinary query function. The difference is that the chatty version (sometimes) prints a text string with a more elaborated explanation of the result. Each chatty query function has the same name as the corresponding ordinary query function, with a single quote appended to the end (e.g., `Reachable'`).

Reachability Properties

The query functions for reachability properties are based on Prop 1.12 in [CPN 2].

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

Reachable determines whether there exists an occurrence sequence from the marking of the first node to the marking of the second. This is done by investigating whether the occurrence graph contains a directed path from the first node to the second. For the dining philosopher system:

```
Reachable (5,3)
```

returns true. This tells us that there exists an occurrence sequence from the marking M_5 (of node 5) to the marking M_3 (of node 3). The function also has a chatty version:

```
Reachable' (5,3)
```

which returns the same result together with the explanation:

```
"A path from node 5 to node 3 is: [5, 9, 3]"
```

This tells us that there exists an occurrence sequence containing the markings M_5 , M_9 and M_3 (in that order). The path is of *minimal* length.

SccReachable returns the same result as **Reachable**, but it uses the Scc-graph, i.e., the strongly connected components. This means that it is faster than **Reachable** (at least for occurrence graphs which contain cycles). The function also has a chatty version:

```
SccReachable' (5,3)
```

which returns the same result together with the explanation:

```
"A path from the SCC of node 5 to the  
SCC of node 3 is: [~1]"
```

This tells us that both M_5 and M_3 belong to the strongly connected component ~ 1 (i.e. the strongly connected component of the initial marking).

AllReachable determines whether all the reachable markings are reachable from each other. This is the case iff there is exactly one strongly connected component. For the dining philosopher system:

```
AllReachable ()
```

returns true.

Boundedness Properties

The query functions for boundedness properties are based on Prop 1.13 in [CPN 2].

```
fun UpperInteger      (Node -> 'a ms) -> int
fun LowerInteger      (Node -> 'a ms) -> int
fun UpperMultiSet     (Node -> 'a ms) -> 'a ms
fun LowerMultiSet     (Node -> 'a ms) -> 'a ms
```

UpperInteger uses a specified function F of type:

$\text{Node} \rightarrow 'a \text{ ms}$

to calculate an integer $F(n)$. This is done for each node n in the occurrence graph, and the maximum of the calculated integers is returned. For the dining philosopher system:

`UpperInteger (Mark.System'Eat 1)`

calculates the maximal number of tokens on place *Eat* on the *first* instance of page *System*. The result is 2, and this tells us that at most two philosophers can eat at the same time.

LowerInteger is analogous to **UpperInteger**, but returns the minimal value of the integers $F(n)$. For the dining philosopher system:

`LowerInteger (Mark.System'Think 1)`

calculates the minimal number of tokens on place *Think* on the *first* instance of page *System*. The result is 3, and this tells us that there always are at least three thinking philosophers.

UpperMultiSet is analogous to `UpperInteger`, but it calculates $F(n)$ instead of $\bar{F}(n)$. The result is the smallest multi-set which is larger than or equal to all the calculated multi-sets. For the dining philosopher system:

```
UpperMultiSet (Mark.System'Eat 1)
```

returns:

```
!! ((1,ph 1)!! ((1,ph 2)!! ((1,ph 5)!!  
((1,ph 3) !! ((1,ph 4) empty))))
```

which is the ML representation of the multi-set:

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

This tells us that each of the five philosophers is able to eat. To obtain the second, more readable format of the result, evaluate the following ML code:

```
mkst_ms'PH (UpperMultiSet (Mark.System'Eat 1))
```

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

```
LowerMultiSet (Mark.System'Eat 1)
```

returns the empty multi-set. This tells us that each of the five philosophers is able to think (because there is a marking in which the philosopher is not eating).

When the four query functions for boundedness are used for a timed place instance of a timed CP-net, you can use `StripTime` to get rid of the time stamps, e.g.:

```
LowerMultiSet (StripTime o (Mark.System'Eat 1))
```

For more information on `StripTime`, see *Time Values* at the end of Chap. 3.

Home Properties

The query functions for home properties are based on Prop 1.14 in [CPN 2].

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

HomeSpace determines whether the set of markings (specified in the list of nodes) is a home space, i.e., whether, from each reachable marking, it is possible to reach at least one of the markings. For the dining philosopher system:

```
HomeSpace [2,6]
```

returns true. The function also has a chatty version.

MinimalHomeSpace returns the minimal number of markings which is needed to form a home space. This is identical to the number of terminal strongly connected components. For the dining philosopher system:

```
MinimalHomeSpace ()
```

returns 1. This function cannot be used for a timed CPN model.

HomeMarking determines whether the marking of the specified node is a home marking, i.e., whether it can be reached from all reachable markings. This is the case iff there is exactly one terminal strongly connected component and the specified marking belongs to that component. For the dining philosopher system:

```
HomeMarking (6)
```

returns true. The function also has a chatty version.

ListHomeMarkings returns a list with all those nodes that are home markings. For the dining philosopher system:

```
ListHomeMarkings ()
```

returns a list which contains all 11 nodes of the occurrence graph. This function cannot be used for a timed CPN model.

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 home markings are exactly those markings that belong to the Scc) or the result is zero (and then there are no home markings). For the dining philosophers:

```
ListHomeScc ()
```

returns ~1 (i.e. the Scc to which the initial marking belongs). This tells us that all reachable markings are home markings. This function cannot be used for a timed CPN model.

HomeMarkingExists determines whether the CP-net has any home markings. This is the case iff there is exactly one terminal strongly connected component. For the dining philosopher system:

```
HomeMarkingExists ()
```

returns true. This function cannot be used for a timed CPN model.

Initial HomeMarking determines whether the initial marking of the occurrence graph is a home marking, i.e., whether it can be reached from all reachable markings. This is the case iff there is exactly one strongly connected component. The result of this function is identical to the result of `AllReachable` (defined in *Reachability Properties*). For the dining philosopher system:

```
InitialHomeMarking ()
```

returns true.

Liveness Properties

The query functions for liveness properties are based on Prop 1.15 in [CPN 2].

```
fun DeadMarking           Node -> bool
fun ListDeadMarkings      unit -> Node list
fun SccListDeadMarkings   unit -> Node list

fun TIsDead    TI.TransInst list * Node -> bool
fun BESDead    Bind.Elem list * Node -> bool
fun ListDeadTIs      unit -> TI.TransInst list

fun TIsLive      TI.TransInst list -> bool
fun BESLive      Bind.Elem list -> bool
fun BESStrictlyLive Bind.Elem list -> bool
fun ListLiveTIs      unit -> TI.TransInst list
```

DeadMarking determines whether the marking of the specified node is dead, i.e., has no enabled binding elements. For the dining philosopher system:

```
DeadMarking (8)
```

returns false. This tells us that M₈ has some enabled binding elements.

ListDeadMarkings returns a list with all those nodes that are dead, i.e., have no enabled binding elements. For the dining philosopher system:

```
ListDeadMarkings ()
```

returns the empty list.

SccListDeadMarkings returns the same result as **ListDeadMarkings**, but it uses the Scc-graph, i.e., the strongly connected components. This means that it is faster than **ListDeadMarkings** (at least for occurrence graphs which contain cycles).

TIsDead determines whether the set of transition instances (specified in the list) is dead in the marking of the specified node, i.e., whether it is impossible to find an occurrence sequence which starts in the marking and contains one of the transition instances. For the dining philosopher system:

```
TIsDead ([TI.System'Take 1],4)
```

returns false. This tells us that there exists an occurrence sequence which starts in M_4 and contains an occurrence of transition *Take* on the *first* instance of the page *System*. The function also has a chatty version:

```
TIsDead' ([TI.System'Take 1],4)
```

which returns the same result together with the explanation:

```
"A transition instance from the given  
list is contained in the SCC: ~1  
(which is reachable from the SCC of  
the given node)"
```

BESDead is analogous to **TIsDead** except that the argument is a list of binding elements (instead of transition instances). For the dining philosopher system:

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

returns false. This tells us that there exists an occurrence sequence which starts in M_4 and contains an occurrence of transition *Take* on the *first* instance of page *System*, with the variable *p* bound to *ph(3)*. The function also has a chatty version.

ListDeadTIs returns a list with all those transition instances that are dead, i.e., do not appear in any occurrence sequence starting from the initial marking of the occurrence graph. For the dining philosopher system:

```
ListDeadTIs ()
```

returns the empty list.

TIsLive determines whether the set of transition instances (specified in the list) is live, i.e., whether, from each reachable marking, it is possible to find an occurrence sequence which contains one of the transition instances. For the dining philosopher system:

```
TIsLive [TI.System'Take 1]
```

returns true. This tells us that it is impossible to reach a marking such that transition *Take* on the *first* instance of page *System* never can occur. The function also has a chatty version.

BESLive is analogous to **TIsLive** except that the argument is a list of binding elements (instead of transition instances). For the dining philosopher system:

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

returns true. This tells us that philosopher *ph(3)* always has a chance to *Take* his chopsticks. He cannot do that in all the reachable markings – but it is always possible to choose a sequence of steps so that this may happen. The function also has a chatty version.

BESStrictlyLive determines whether the set of binding elements (specified in the list) is strictly live, i.e., whether each individual element in the list is live. For the dining philosopher system:

```
BESStrictlyLive [
  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)}),
  Bind.System'Take (1, {p=ph(5)})]
```

returns true. This tells us that each philosopher always has a chance to *Take* his chopsticks. He cannot do that in all the reachable markings – but it is always possible to choose a sequence of steps so that this may happen.

ListLiveTIs returns a list with all those transition instances that are live. For the dining philosopher system:

```
ListLiveTIs ()
```

returns:

```
[TI.System'Put 1, TI.System'Take 1]
```

This tells us that it is impossible to reach a marking such that one of the transition instances never can occur.

Fairness Properties

The query functions for fairness properties are based on Prop 1.16 in [CPN 2].

```
fun TIsFairness    TI.TransInst list ->
                                   FairnessProperty
fun BEsFairness    Bind.Elem list ->
                                   FairnessProperty
fun ListImpartialTIs unit -> TI.TransInst list
fun ListFairTIs      unit -> TI.TransInst list
fun ListJustTIs      unit -> TI.TransInst list
```

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

{Impartial, Fair, Just, No_Fairness}.

A definition and explanation of impartial, fairness and justice can be found in Sect. 4.5 of [CPN 1].

TIsFairness determines whether the set of transition instances (specified in the list) is impartial, fair or just. For the dining philosopher system:

```
TIsFairness [TI.System'Take 1]
```

returns *Impartial*. This tells us that we cannot have an infinite occurrence sequence unless transition *Take* on the *first* instance of page *System* continues to occur.

BEsFairness is analogous to **TIsFairness** except that the argument is a list of binding elements (instead of transition instances). For the dining philosopher system:

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


returns `No_Fairness`. This tells us that it is possible to have an infinite occurrence sequence (starting from a reachable marking) in which philosopher three never takes his chopsticks.

ListImpartialTIs returns a list with all those transition instances that are impartial. For the dining philosopher system:

```
ListImpartialTIs ( )
```

returns the list:

```
[TI.System'Put 1,TI.System'Take 1]
```

This tells us that all infinite occurrence sequences (starting from the initial marking) contains an infinite number of both transition instances.

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

Impartiality implies fairness which in turn implies justice. Hence, we always have:

```
ListImpartialTIs( )  
ListFairTIs( )  
ListJustTIs( )
```


Chapter 5

How to Make Your Own Queries

This chapter describes how you can make your own non-standard queries – by writing some simple ML functions. First we introduce a number of functions to inspect the structure of an occurrence graph and an Scc-graph. Then we describe three search functions by which you can traverse the nodes, arcs and strongly connected components of an occurrence graph.

Nodes and Arcs

The following functions allow you to “move” between adjacent nodes and arcs of the occurrence graph:

```
fun SourceNode      Arc -> Node
fun DestNode       Arc -> Node
fun Arcs           Node * Node -> Arc list
fun InNodes        Node -> Node list
fun OutNodes       Node -> Node list
fun InArcs         Node -> Arc list
fun OutArcs        Node -> Arc list
```

The following function tells whether a node is terminal (i.e., have no outgoing arcs). The result is identical to the result of `DeadMarking` (in Chap. 4).

```
fun Terminal        Node -> bool
```

The following functions return the nodes/arcs in one of the *shortest* paths between the two specified nodes. If no path exists the exception `NoPathExists` is raised:

```
fun NodesInPath     Node * Node -> Node list
fun ArcsInPath      Node * Node -> Arc list
```

By definition we always have:

```
NodesInPath (n,n) = [n]
ArcsInPath  (n,n) = []
```

The following functions determine to which extent a node has been processed. The *Branching Options* (in Chap.7) allow you to specify that a node can be processed without calculating all the immediate successors. The second function checks whether this is the case:

```
fun Processed           Node -> bool
fun FullyProcessed     Node -> bool
```

Strongly Connected Components

Each node of an Scc-graph is a strongly connected component while each arc of an Scc-graph is an ordinary occurrence graph arc. We have an Scc arc for each occurrence graph arc that starts in one Scc and ends in another.

The following functions allow you to “move” between strongly connected components and their nodes/arcs. The first function maps an occurrence graph node into the Scc to which it belongs. The second function maps an occurrence graph arc into the Scc from which it starts (i.e., the Scc to which the source node belongs). The third function maps an Scc node into the occurrence graph nodes that belong to the Scc. Finally, the fourth function maps an Scc node into the occurrence graph arcs that start in the Scc (while we don't care where the arcs end):

```
fun NodeToScc           Node -> Scc
fun ArcToScc            Arc  -> Scc
fun SccToNodes          Scc  -> Node list
fun SccToArcs           Scc  -> Arc list
```

The following functions (for Scc-graphs) are analogous to the functions defined in *Nodes and Arcs* (at the beginning of this chapter). Hence they have the same names – prefixed with “Scc”.

```
fun SccSourceNode       Arc  -> Scc
fun SccDestNode         Arc  -> Scc
fun SccArcs             Scc * Scc -> Arc list
fun SccInNodes          Scc  -> Scc list
fun SccOutNodes         Scc  -> Scc list
fun SccInArcs           Scc  -> Arc list
```

How to Make Your Own Queries

```
fun SccOutArcs          Scc -> Arc list
fun SccTerminal         Scc -> bool
fun SccNodesInPath     Scc * Scc -> Scc list
fun SccArcsInPath      Scc * Scc -> Arc list
```

The following function tells whether a strongly connected component is trivial (i.e., consists of a single node and no arcs):

```
fun SccTrivial          Scc -> bool
```

SearchNodes

The function **SearchNodes** traverses the nodes of the occurrence graph. At each node some specified calculation is performed and the results of these calculations are combined, in some specified way, to form the final result.

SearchNodes takes six different arguments and by varying these arguments it is possible to specify a lot of different queries, e.g., many of the standard queries from Chap. 4. The following description is taken from Sect. 1.7 of [CPN 2]:

Search Area	This argument specifies the part of the occurrence graph which should be searched. It is often all nodes, but it may also be any other subset of nodes, e.g., those belonging to a strongly connected component.
Predicate function	This argument specifies a function. It maps each node into a boolean value. Those nodes which evaluate to false are ignored; the others take part in the further analysis – as described below.
Search Limit	This argument specifies an integer. It tells us how many times the predicate function may evaluate to true before we terminate the search. The search limit may be infinite. This means we always search through the entire search area.
Evaluation function	This argument specifies a function. It maps each node into a value, of some type A. It is important to notice that the evaluation function is only used at those nodes (of the search area) for which the predicate function evaluates to true.

Occurrence Graph Manual

Start value	This argument specifies a constant, of some type B.
Combination function	This argument specifies a function. It maps from $A \times B$ into B, and it describes how each individual result (obtained by the evaluation function) is combined with the prior results.

SearchNodes works as described by the following Pascal-style pseudo-code. When the function terminates it returns the value of *Result*:

```
SearchNodes (Area, Pred, Limit, Eval, Start, Comb)
begin
  Result := Start; Found := 0
  for all n Area do
    if Pred(n) then
      begin
        Result := Comb(Eval(n), Result)
        Found := Found + 1
        if Found = Limit then stop for-loop
      end
    end
  end
end.
```

The arguments have the following types:

area	search area	Node list
pred	predicate function	Node -> bool
limit	search limit	int
eval	evaluation function	Node -> 'a
start	start value	'b
comb	combine function	'a * 'b -> 'b

The ML types 'a and 'b are arbitrary and may be identical. The search area is specified by a list of nodes (if a node is listed twice it will be searched twice). By convention we use:

```
val EntireGraph
```

How to Make Your Own Queries

to denote the set of all nodes in the occurrence graph. The search limit is specified by a positive integer. By convention we use:

```
val NoLimit
```

to specify an infinite limit, i.e., that the search continues until the entire search area has fully been traversed.

The `SearchNodes` function is a bit complicated. But it is also extremely general and powerful. As an example, we can use `SearchNodes` to implement the query function `ListDeadMarkings` from Chap.4, i.e., to find all the dead markings. Then we simply use the following arguments:

Search area	EntireGraph
Predicate function	fun Pred(n) = (length(OutArcs(n)) = 0)
Search limit	10
Evaluation function	fun Eval(n) = n
Start value	[]
Combination function	fun Comb(new,old) = new::old

The predicate function uses the function `OutArcs` (from *Nodes and Arcs* at the beginning of this chapter) to get a list of all the output arcs. If the length of this list is zero there are no successors, and thus we have a dead marking. The evaluation function maps a node into itself, i.e., into the unique node number. The combination function adds each new dead marking to the list of those which we have previously found. With these arguments `SearchNodes` returns a list with at most 10 dead markings. If the list is empty there is no dead marking. If the length is less than 10, the list contains all the dead markings. The exact ML call looks as follows:

```
SearchNodes (
  EntireGraph,
  fn n => (length(OutArcs(n)) = 0),
  10,
  fn n => n,
  [ ],
  op ::)
```

Occurrence Graph Manual

As a second example, we may use `SearchNodes` to implement the query function `UpperInteger` from Chap. 4, i.e., to find the best upper integer bound for a given place instance `p` `PI`. This is done by using the following arguments:

Search area	EntireGraph
Predicate function	fun Pred(n) = true
Search limit	NoLimit
Evaluation function	fun Eval(n) = Mark(p)(n)
Start value	0
Combination function	max

The exact ML call looks as follows (for the place *Eat* on the *first* instance of the page *System*):

```
SearchNodes (
  EntireGraph,
  fn _ => true,
  NoLimit,
  fn n => size (Mark.System'Eat 1 n),
  0,
  max)
```

PredNodes and EvalNodes

For convenience we also define some abbreviated forms of `SearchNodes` where one or more of the arguments are predefined. The first function searches the specified area and returns a list of all those nodes that satisfy the specified predicate. The predeclared function `id` maps an arbitrary ML value into itself:

```
fun PredNodes (area, pred, limit) : Node list
  = SearchNodes (area, pred, limit, id, [], op ::)
```


How to Make Your Own Queries

The second function searches the specified area and returns a list of all the calculated values:

```
fun EvalNodes (area, eval) : 'b list
  = SearchNodes (area, fn _ => true,
                 NoLimit, eval, [], op ::)
```

The next three functions are identical to SearchNodes, PredNodes and EvalNodes, except that they always search the entire occurrence graph:

```
fun SearchAllNodes
  (pred, eval, start, comb) : 'b
  = SearchNodes (EntireGraph, pred,
                 NoLimit, eval, start, comb)

fun PredAllNodes (pred) : Node list
  = PredNodes (EntireGraph, pred, NoLimit)

fun EvalAllNodes (eval) : 'b list
  = EvalNodes (EntireGraph, eval)
```

The final function is identical to SearchNodes, except that the search area consists of those nodes that are reachable from the node in the first argument:

```
fun SearchReachableNodes
  (node, pred, limit, eval, start, comb) : 'b
```

Examples of SearchNodes Calls

Two of the query functions from Chaps. 4 and 6 can be implemented as follows:

```
fun ListDeadMarkings () : Node list
  = PredAllNodes Terminal

fun EntireGraphCalculated () : bool
  = (PredAllNodes (fn n =>
                    not(FullyProcessed n)) = [])
```

Occurrence Graph Manual

If the occurrence graph contains unprocessed nodes, it may be desirable to exclude these from the node list returned by `ListDeadMarkings`. We then get the following function:

```
fun ListDeadMarkingsFP () : Node list
  = PredAllNodes (fn n => (Terminal n)
                    andalso (FullyProcessed n))
```

All nodes in which a particular philosopher is eating can be found as follows (where `cf` returns the coefficient of the specified colour in the specified multi-set):

```
fun Eating (p:PH) : Node list
  = PredAllNodes (fn n =>
                    cf(p,Mark.System'Eat 1 n) > 0)
```

The maximal number of simultaneously enabled transition instances can be found as follows (where `remdupl` removes duplicates from a list, while `map` uses the specified function on all the elements of the specified list):

```
fun MaxTIEabled () : int
  = SearchAllNodes(
    fn _ => true,
    fn n => length(remdupl(
      map ArcToTI(OutArcs n))),
    0,
    max)
```

Checking whether there are reachable markings in which two neighbouring philosophers simultaneously eat, can be done as follows (where `next` is a function mapping each philosopher in its successor, `ext_col` extends a function `'a -> 'b` to a function `'a ms -> 'b ms`, while `<=<=` is the less-than-equal operation on multi-sets):

```
fun EatingNeighbours () : Node list
  = PredAllNodes(fn n =>
    let
      val Eating = Mark.System'Eat 1 n
    in
      not(Eating +
        ext_col next Eating <=<= PH)
    end)
```

Checking whether there are nodes which violate the linear invariant:

$$M(\text{Unused}) + \text{Chopsticks}(M(\text{Eat})) = \text{CS}$$

can be done in the following way (where $\langle \rangle$ is the operator which checks whether two multi-sets are different from each other):

```
fun InvariantViolations () : Node list
= PredAllNodes(
  fn n => Mark.System'Unused 1 n +
    ext_ms Chopsticks (Mark.System'Eat 1 n)
    <><> CS)
```

SearchArcs

The function **SearchArcs** traverses the arcs of the occurrence graph. At each arc some specified calculation is performed and the results of these calculations are combined, in some specified way, to the form the final result.

We define **SearchArcs** in a way which is totally analogous to **SearchNodes**. The arguments have the following types:

area	search area	Arc list
pred	predicate function	Arc -> bool
limit	search limit	int
eval	evaluation function	Arc -> 'a
start	start value	'b
comb	combine function	'a * 'b -> 'b

We define **PredArcs**, **EvalArcs**, **SearchAllArcs**, **PredAllArcs**, **EvalAllArcs**, and **SearchReachableArcs** analogously to **PredNodes**, **EvalNodes**, **SearchAllNodes**, **PredAllNodes**, **EvalAllNodes**, and **SearchReachableNodes**. The latter searches all the arcs which are reachable from the *node* specified in the first argument.

Examples of SearchArcs Calls

The following function returns all the arcs where transition *Take* occurs on the *first* instance of page *System* with the variable *p* bound to a specified philosopher:

```
fun TakeChopsticks (p:PH) : Arc list
= PredAllArcs(
  fn a => case ArcToBE a
  of Bind.System'Take (1,{p=p'}) => p=p'
  | _ => false)
```

For the ring network, the following function returns all the arcs where transition *Send* occurs on *some* instance of page *Site* with variables *s* and *r* bound to the same value:

```
fun SendToMyself () : Arc list
= PredAllArcs(
  fn a => case ArcToBE a
  of Bind.System'Send
    (1,{s=v1,r=v2,...}) => v1=v2
  | _ => false)
```

SearchSccs

The function **SearchSccs** traverses the strongly connected components of the occurrence graph. At each strongly connected component some specified calculation is performed and the results of these calculations are combined, in some specified way, to form the final result.

We define **SearchSccs** in a way which is totally analogous to **SearchNodes** and **SearchArcs**. The arguments have the following types:

How to Make Your Own Queries

area	search area	Scc list
pred	predicate function	Scc -> bool
limit	search limit	int
eval	evaluation function	Scc -> 'a
start	start value	'b
comb	combine function	'a * 'b -> 'b

We define **PredSccs**, **EvalSccs**, **SearchAllSccs**, **PredAllSccs**, **EvalAllSccs**, and **SearchReachableSccs** analogously to **PredNodes**, **EvalNodes**, **SearchAllNodes**, **PredAllNodes**, **EvalAllNodes**, and **SearchReachableNodes**. The latter searches all the strongly connected components which are reachable from the *Scc* specified in the first argument.

Examples of SearchSccs Calls

Two of the query functions from Chap. 4 can be implemented as follows:

```
fun HomeMarkingExists () : bool
  = (length(PredAllSccs SccTerminal) = 1)

fun HomeMarking (n:Node) : bool
  = SccTerminal(NodeToScc(n)) andalso
    HomeMarkingExists()
```


Chapter 6

How to Draw an Occurrence Graph

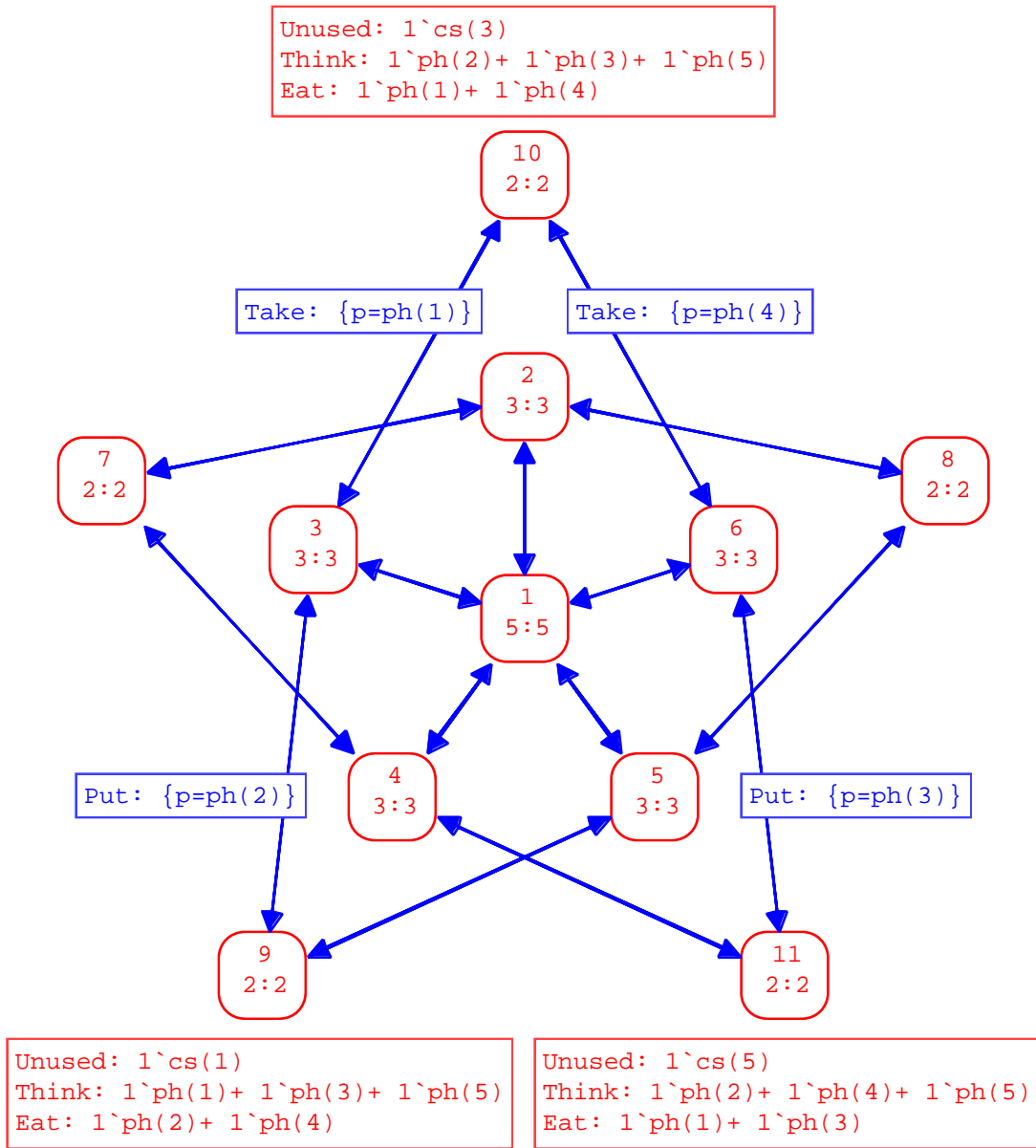
In addition to the standard report described in Chap. 2 and the query functions described in Chaps. 4 and 5, you may investigate the behaviour of a CPN model by making a drawing of the occurrence graph. For large occurrence graphs it only makes sense to draw small parts e.g., the nearest surroundings of some interesting nodes, found by means of the query functions. This chapter describes how to make drawings of the occurrence graph and the Scc-graph.

Object Types

The occurrence graph for the dining philosopher system may be drawn as shown below.

The rounded boxes are **OG nodes** while the arcs are **OG arcs**. The dashed boxes next to the OG nodes are **OG node descriptors** while those next to the OG arcs are **OG arc descriptors**. For readability we have only shown some of the node and arc descriptors.

Each OG node contains the node number and the number of immediate predecessors and successors – in the presently *calculated* occurrence graph. The full occurrence graph may have more predecessors/ successors while the present drawing may have less. In the example above, node number 1 has five predecessors and five successors. Nodes 2-6 have three predecessors and three successors, while the remaining nodes have two predecessors and two successors. You can create/delete a node/arc descriptor by double clicking the corresponding node/arc (or by using the **Toggle Descriptor** command).



For an unprocessed OG node a double quote is appended to the node number, for a partially processed node a single quote is appended, and for a fully processed node a blank.

We have customised the contents of the text strings in the node/arc descriptors. This was done by means of the *String Representation Options* in Chap. 7. In the default settings each node descriptor contains the node number and the marking of the corresponding node. Each arc descriptor contains the arc number, the source and destination node numbers and the binding element of the arc.

If an OG object is copied/cut and then pasted, the resulting object becomes an auxiliary object.

Occ Menu

To draw an occurrence graph the commands of the Occ menu is used:

Occ
Calculate Occ Graph
Calculate Successors
Calculate Scc Graph

Show Statistics...
Save Report...

Occ State to Sim...
Sim State to Occ

Attributes/Options...

Display Node...
Display Arc...
Display Successors
Display Predecessors
Display Scc Graph

Toggle Descriptor
Update Node

Calculate Occ Graph

This command calculates the occurrence graph. It observes the *Stop Options* and *Branching Options* in Chap.7. The command must be used with care since occurrence graphs tend to be quite large. During the calculation of the occurrence graph a progress report is displayed in the status bar (usually for every 60 seconds). The report may look as follows:

#nodes 218/500/623 #arcs 725/1000/1524 #secs 14/300/48
--

The first number tells how many new nodes there have been generated (for the present call of **Calculate Occ Graph**). The next number shows the current *Stop Option* (for the number of nodes). If the stop

Occurrence Graph Manual

option is inactive the number is omitted. The third number gives the total number of nodes in the occurrence graph (up to now).

The next three numbers contain similar information for arcs while the last three numbers contain information for the use of real time (not CPU time).

Calculate Successors

This command calculates the immediate successors of the selected OG node(s). It observes the *Stop Options* and *Branching Options* in Chap.7. When a full occurrence graph has been calculated, there is no need to use **Calculate Successors**. However, for a partial occurrence graph, the command allows you to determine the “direction” in which you want the system to make further developments.

Calculate Scc Graph

This command calculates the Scc-graph of the occurrence graph. The Scc-graph is used by many of the query functions in Chap.4.

Show Statistics

This command gives information about the size of the occurrence graph and the size of the Scc-graph. The information is identical to the statistics in **Save Report** – but it is displayed in a dialogue box instead of a text file.

The occurrence 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 represents the initial marking. If the Scc-graph has not been calculated, the second part of the statistics is missing.

If the occurrence graph is partial you may extend it by invoking **Calculate Occ Graph** once more (perhaps after modifying some of the *Stop Options* or *Branching Options*).

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
```

How to Draw an Occurrence Graph

```
fun SccNoOfNodes                unit -> int
fun SccNoOfArcs                  unit -> int
fun SccNoOfSecs                  unit -> int
fun SccGraphCalculated          unit -> bool
```

Save Report

This command generates a text file containing a **standard report** with information about:

- Statistics (size of occurrence graph and Scc-graph).
- Boundedness Properties (integer and multi-set bounds for place instances).
- Home Properties (home markings).
- Liveness Properties (dead markings, dead/live transition instances).
- Fairness Properties (impartial/fair/just transition instances).

For more details see the end of Chap. 2.

Occ State to Sim

This command allows you to “move” an occurrence graph state to the simulator. This is very useful. You can inspect the marking directly on the graphical representation of the CP-net. You can see the enabled transition instances, investigate their bindings and make simulations. The OG node (to be moved) is either selected (prior to the invocation of the command) or specified in a dialogue box.

Sim State to Occ

This command allows you to “move” a simulator state to the occurrence graph (where it becomes a node). A dialogue box reports the node number and whether the state already belonged to the occurrence graph. The new node can (as all other nodes) be drawn by means of **Display Node**, but this does not happen automatically.

Attributes/Options

This command allows you to change the diagram defaults for OG attributes and the values of OG options. For more information see Chap. 7.

Display Node

This command draws a new OG node – providing a graphical representation of the specified node. The graphical appearance of the node is determined by the *Node Attributes* in Chap. 7. The node is drawn at the centre of the current page

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

OG 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 OG 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**. The graphical appearance of the arcs is determined by the *Arc Attributes* in Chap. 7.

OG 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). The positioning of the new OG nodes/arcs are determined by the *Succ/Pred Options* in Chap. 7, and they also determine the maximal number of successors which are drawn by one call of **Display Successors**. Additional calls will draw addi-

How to Draw an Occurrence Graph

tional successors (if any). It should be noted that the command only draws nodes and arcs which already have been calculated. Hence, it may be necessary/desirable to call **Calculate Occ Graph** or **Calculate Successors** before the use of **Display Successors**.

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. It only works well when there are reasonably few Scc nodes. In a later version, we will allow the components of an Scc-graph to be drawn by the same functions as the components of the occurrence graph.

Toggle Descriptor

This command toggles the existence of the OG node/arc descriptor of the selected OG node/arc(s). If the descriptor does not exist, it is created. If it exists, it is deleted. The graphical appearance of the descriptors is determined by the *Node Descriptor Attributes* and the *Arc Descriptor Attributes* in Chap. 7 while the contents is determined by the *Node Descriptor Options* and the *Arc Descriptor Options* in Chap. 7.

Update Node

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

Chapter 7

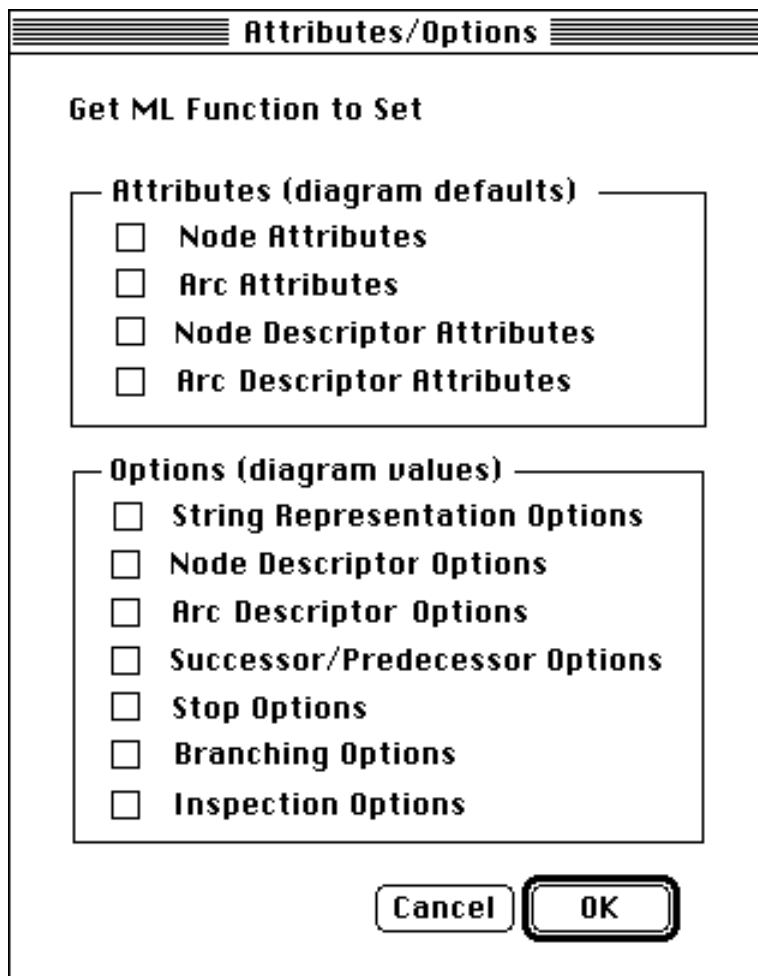
How to Change Attributes and Options

The OG tool has a large number of attributes and options. **Attributes** determine the graphical appearance of the individual OG objects. **Options** determine the way in which the commands of the OG tool works. **Diagram defaults** determine the initial values of the attributes of new objects. **System defaults** determine the diagram defaults of new CPN diagrams.

In the current version of the OG tool, attributes and options are handled in a provisional way. The **Attributes/Options** command allows you to change the diagram defaults of OG attributes and the values of OG options. Attribute values of individual OG objects are changed via the usual attribute commands in the **Set** menu. System defaults cannot be changed.

Attributes/Options Command

When the command is invoked it displays the following dialogue box:



The image shows a dialog box titled "Attributes/Options". Inside, there is a section titled "Get ML Function to Set". Below this, there are two main categories of settings, each enclosed in a rounded rectangle. The first category is "Attributes (diagram defaults)" and contains four checkboxes: "Node Attributes", "Arc Attributes", "Node Descriptor Attributes", and "Arc Descriptor Attributes". The second category is "Options (diagram values)" and contains seven checkboxes: "String Representation Options", "Node Descriptor Options", "Arc Descriptor Options", "Successor/Predecessor Options", "Stop Options", "Branching Options", and "Inspection Options". At the bottom right of the dialog box are two buttons: "Cancel" and "OK".

Attributes/Options

Get ML Function to Set

Attributes (diagram defaults)

- ☐ Node Attributes
- ☐ Arc Attributes
- ☐ Node Descriptor Attributes
- ☐ Arc Descriptor Attributes

Options (diagram values)

- ☐ String Representation Options
- ☐ Node Descriptor Options
- ☐ Arc Descriptor Options
- ☐ Successor/Predecessor Options
- ☐ Stop Options
- ☐ Branching Options
- ☐ Inspection Options

Cancel OK

When the OK button is pressed, a new box is created (on the current page). The box contains ML functions to change the diagram defaults/values of those attributes/options which you have selected (via the check boxes). The current values of the diagram defaults/options are shown in the box – except in those cases where the value is an ML function (for these we show the system defaults). To change the options, you replace the values with those, which you want, and then evaluate the ML functions by means of the **ML Evaluate** command.

Attributes

The following ML function changes the diagram defaults of OG nodes (the values indicate the system defaults):

```
OGSet.NodeAttributes{  
  Size = {width = 35, height = 35},  
  Graphics = {fill = None, line = Solid,  
              thick = 1},  
  Text = {size = 10, font = Courier,  
           style = PlainText,  
           just = Centered}}
```

The following ML function changes the diagram defaults of OG arcs (the values indicate the system defaults):

```
OGSet.ArcAttributes{  
  Graphics = {fill = Black, line = Solid,  
              thick = 1}}
```

The following ML function changes the diagram defaults of OG node descriptors (the values indicate the system defaults):

```
OGSet.NodeDescriptorAttributes{  
  Size = {width = 250, height = 80},  
  Graphics = {fill = White, line =  
              Dashed/Diagonal, thick = 1},  
  Text = {size = 10, font = Courier,  
           style = PlainText,  
           just = LeftJustification}}
```

The following ML function changes the diagram defaults of OG arc descriptors (the values indicate the system defaults):

```
OGSet.ArcDescriptorAttributes{  
  Size = {width = 100, height = 40},  
  Graphics = {fill = White, line =  
              Dashed/Diagonal, thick = 1},  
  Text = {size = 10, font = Courier,  
          style = PlainText,  
          just = LeftJustification}}
```

To specify fill and line patterns the following constants can be used:

Fill patterns on the Unix platform:
None, Black, White

Line patterns on the Unix platform:
Solid, Dashed, LongDashed, Dotted,
DottedDashed

Fill and line patterns on the Mac platform:
None, Black, White, Horizontal,
Vertical, Diagonal

To specify text styles the following constants can be used (combinations are obtained by adding the corresponding constants):

Bold, Condense, Extended, Italic, Outline,
PlainText, Shadow, Underline

To specify text justification the following constants can be used:

Centered, LeftJustification,
RightJustification

String Representation Options

The *String Representation Options* allow the user to specify how he wants the st-functions in Chap. 3 to work. As an example, he may determine whether he wants a transition instance to be represented as:

System'Take 1 or Take

The first representation is convenient for a CP-net with many different pages/page instances, while the second representation is convenient for a system with only one or a few pages (and only one instance of each page).

How to Change Attributes and Options

The st-functions are used for the standard reports generated by **Save Report** and for the contents of OG node/arc descriptors. Hence, the options also influence these things. However, it should be noted that the string representation options do not influence the input format of the different ML functions in Chaps. 3-5. This means, e.g., that the user always has to specify the page and instance of a transition instance – even though he may have decided to omit this information from the text strings created by `st_TI`.

For each of the st-functions we provide an ML function which specifies how the individual substrings are put together (e.g., the order and the separators). The options are changed by the following set of ML functions (the values indicate the system defaults):

```
OGSet.StringRepOptions'Node(  
  fn (node) => node)  
  
OGSet.StringRepOptions'Arc(  
  fn (arc,source,dest) =>  
    arc ^ ":" ^ source ^ "->" ^ dest)  
  
OGSet.StringRepOptions'PI(  
  fn (page,place,inst) =>  
    page ^ " " ^ place ^ " " ^ inst)  
  
OGSet.StringRepOptions'TI(  
  fn (page,trans,inst) =>  
    page ^ " " ^ trans ^ " " ^ inst)  
  
OGSet.StringRepOptions'BE(  
  fn (TI,bind) => TI ^ ": " ^ bind)  
  
OGSet.StringRepOptions'Mark(  
  fn (PI,mark) => PI ^ ": " ^ mark ^ NEWLINE)
```

The more compact string representations mentioned in Chap. 3 are obtained by using the following options:

```
OGSet.StringRepOptions'PI(  
  fn (page,place,inst) => place)
```

```
OGSet.StringRepOptions'TI(  
  fn (page,trans,inst) => trans)  
  
OGSet.StringRepOptions'BE(  
  fn (TI,bind) => TI ^ ": " ^ bind)" )  
  
OGSet.StringRepOptions'Mark(  
  fn (PI,mark) =>  
    if mark="empty" orelse mark="tempty"  
    then ""  
    else PI ^ ": " ^ mark ^ NEWLINE)
```

Node and Arc Descriptor Options

The *Node Descriptor Options* determine the contents of the OG node descriptors. They are changed by the following ML function (the value indicate the system defaults):

```
OGSet.NodeDescriptorOptions(  
  fn n =>  
    (st_Node n)^NEWLINE^  
    (st_Mark.System'Unused 1 n)^  
    (st_Mark.System'Think 1 n)^  
    (st_Mark.System'Eat 1 n))
```

You may replace the default by any other ML function of type:

```
Node -> string
```

In this way it is possible to obtain a compact representation of a complex marking. As an example it is possible to omit the marking of some place instances or only show the number of tokens (ignoring the token colours).

The *Arc Descriptor Options* determine the contents of the OG arc descriptors. They are changed by the following ML function (the value indicate the system defaults):

```
OGSet.ArcDescriptorOptions(  
  fn (a:Arc):string =>  
    (st_Arc a)^NEWLINE^  
    (st_BE(ArcToBE a)))
```

How to Change Attributes and Options

You may replace the default by any other ML function of type:

```
Arc -> string
```

In this way it is possible to obtain a compact representation of a complex binding element.

Successor/Predecessor Options

The *Successor/Predecessor Options* determine how **Draw Successors** and **Draw Predecessors** position the new nodes. They also determine the maximum number of successors/predecessors to be drawn. The options are changed by the following ML function (the values indicate the system defaults):

```
OGSet.SuccPredOptions{  
  FirstSucc = {horz = 0,vert = 80},  
  NextSucc = {horz = 50,vert = 10},  
  MaxSucc = 10,  
  FirstPred = {horz = 0,vert = ~80},  
  NextPred = {horz = ~50,vert = ~10}  
  MaxPred = 10}
```

The first argument determines the offset of the first successor (relative to the original node). Positive values means “to the right” and “downwards”. The second argument determines the offset of the second successor (relative to the first successor). It also determines the offset of the third successor (relative to the second) – and so on. The third argument determines the maximal number of successors drawn by one call of **Display Successors**. The last three arguments are analogous to the first three, but determines the offset and maximal number of predecessors (instead of successors).

Stop Options

The *Stop Options* allow you to determine when the calculation of an occurrence graph stops. This happens when all nodes have been processed or when one of the stop options becomes satisfied. The options are changed by the following ML function (the values indicate the system defaults):

```
OGSet.StopOptions{  
  Nodes = NoLimit,  
  Arcs = NoLimit,  
  Secs = 300,  
  Predicate = fn _ => false}
```

The first three arguments specify the maximal number of nodes, arcs and seconds. By convention, zero indicates `NoLimit` (i.e., that the corresponding stop option is inactive). All counts are reset to zero whenever you call **Calculate Occ Graph**. This means that you can extend an occurrence graph without changing the *Stop Options*. The fourth argument specifies a predicate function which is evaluated *after* the calculation of the successors:

```
Node -> bool
```

If the predicate evaluates to true the calculation of the occurrence graph will be stopped. This can, e.g., be used to stop when a dead marking is found:

```
fn n => Terminal n
```

When a *Stop Option* has been met, the exception `StopOptionSatisfied` is raised (this can, e.g., be seen if the occurrence graph is generated by means of the **CalculateOccGraph** function described in Chap. 2). Furthermore a message is printed with details about the activated stop option.

Warning: It is impossible to stop a “run-away” occurrence graph generation in a graceful way. Hence, it is important that the *Stop Options* in have some sensible values

Branching Options

The *Branching Options* allow you to specify that, under certain circumstances, the OG tool need not calculate all the successors of a node. The node is then said to be only *partially processed*. The options are changed by the following ML function the values indicate the system defaults):

```
OGSet.BranchingOptions{  
  TransInsts = NoLimit,  
  Bindings = NoLimit,  
  Predicate = fn _ => true}
```

How to Change Attributes and Options

The first argument specifies the maximal number of enabled transition instances to be used to find successor markings (for each node). Analogously, the second argument specifies the maximal number of enabled bindings to be used (for each enabled transition instance). By convention zero indicates NoLimit. All counts are reset to zero whenever you invoke **Calculate Occ Graph**. This means that you can extend the number of calculated successor markings without changing the *Branching Options*. The third argument specifies a predicate function which is evaluated *before* the calculation of the successors:

```
Node -> bool
```

If the predicate evaluates to false no successors are calculated.

Nodes which are processed, without calculating all successors, are marked as *partially processed*. When you add to an existing occurrence graph, some of the partially processed nodes may become *fully processed*.

The generation of new nodes progresses in a *width first* fashion. This means that the nodes are being processed in the order in which they were created. To a certain extent, a depth first generation can be obtained by using "narrow" *Branching Options*.

For a timed occurrence graph the processing order is determined by the creation time (i.e., the model time at which the individual markings start to exist).

Inspection Options

The *Inspection Options* allow you to specify how often you want to get a progress report in the status bar – during an execution of **Calculate Occ Graph**. The options are changed by the following ML function (the values indicate the system defaults):

```
OGSet.InspectionOptions{  
    Frequency = 60,  
    Action = !OGSet.StandardInspectionReport}
```

The frequency is measured in seconds. The contents of the standard report is explained in Chap. 6. You may replace the standard report by another ML function of type:

```
unit -> unit
```


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