CPN Tools
State Space Manual

Last updated: January 2006

University of Aarhus

Department of Computer Science
Aabogade 34
DK-8200 Aarhus N, Denmark
Tel: +45 89 42 56 00
Fax: +45 89 42 56 01

Department of Computer Science

Aabogade 34

DK-8200 Aarhus N, Denmark

Tel: +45 89 42 56 00

Fax: +45 89 42 56 01

e-mail: cpntools-support@daimi.au.dk

Authors: Kurt Jensen, Søren Christensen and Lars M. Kristensen.

CPN Tools
State Space Manual

Table of Contents

Index

O

P

R

S

T

U

Chapter 1

Introduction to State Spaces

The History of the State Space Tool

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

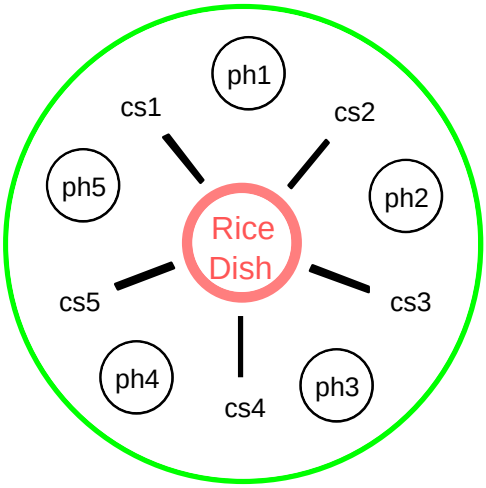
The SS tool is integrated with CPN Tools. This means that you can easily switch between the editor, the simulator, and the SS tool. When a state space node has been found, it can be inspected in the simulator. This means that you can see the marking directly on the graphical representation of the CPN 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 state space or used as the initial marking for a new state space.

The SS 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 require that you write 2-5 lines of quite straightforward ML code.

Example: Dining Philosophers

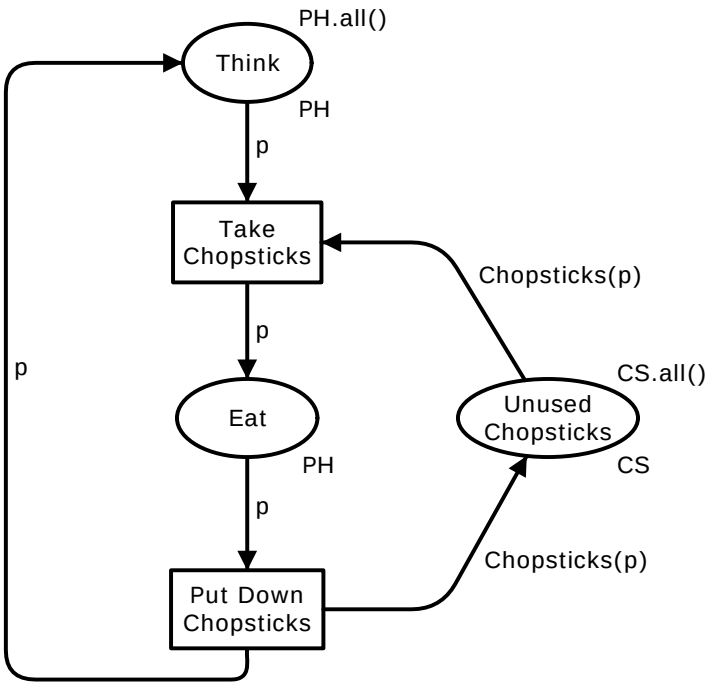
The basic idea behind state spaces is to make a directed graph with a node for each reachable marking and an arc for each occurring binding element. An introduction to state spaces can be found in Sect. 5.1 of [CPN 1] and in Sect. 1.1 of [CPN 2].

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.

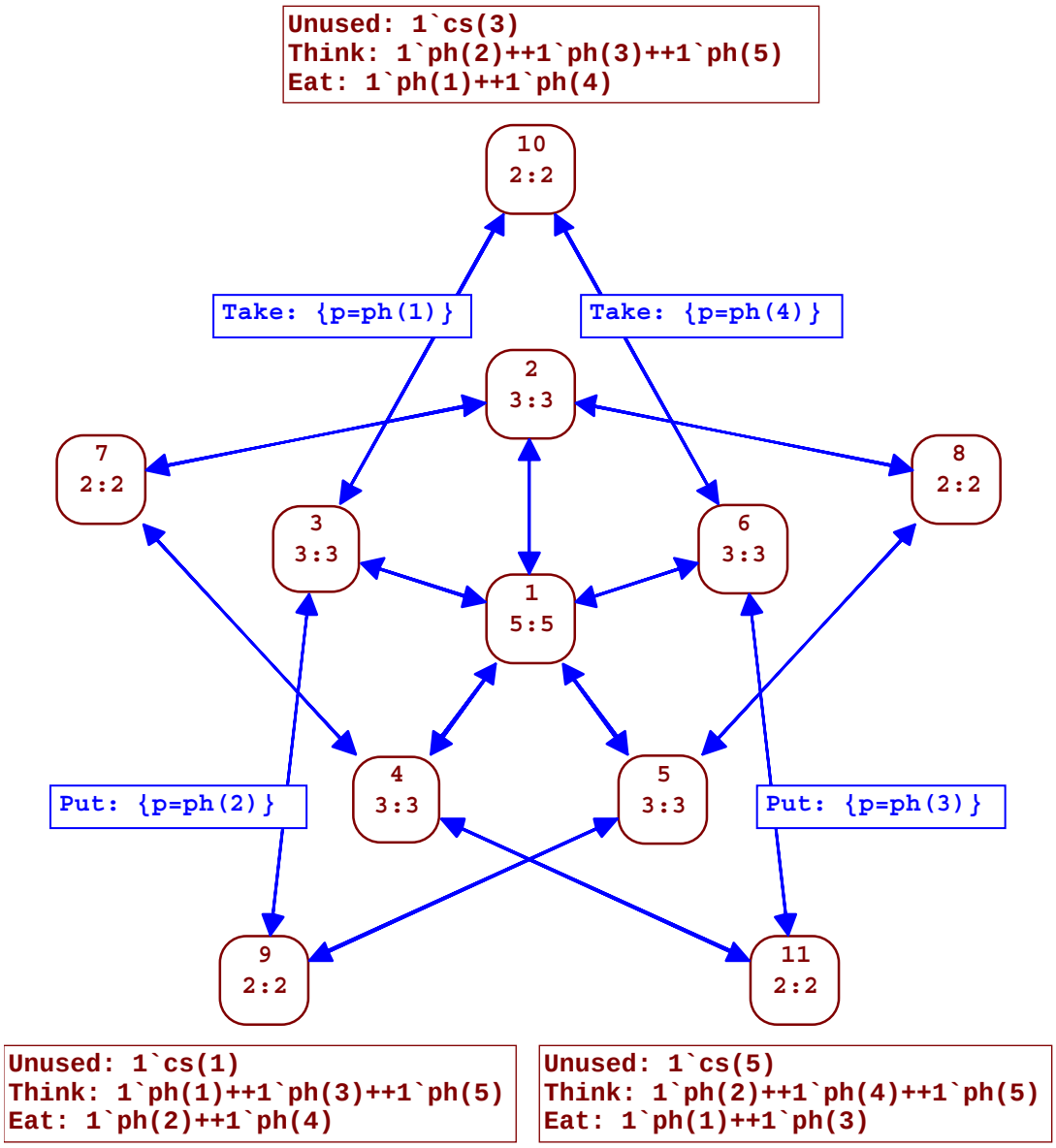


The philosopher system is modelled by the CP-net shown below. The PH colour set represents the philosophers, while the CS colour set represents the chopsticks. The function *Chopsticks* maps each philosopher into the two chopsticks next to him.

```
val n = 5;
colset PH = index ph with 1..n;
colset CS = index cs with 1..n;
var p: PH;
fun Chopsticks(ph(i)) =
1`cs(i) ++ 1`cs(if i=n then 1 else i+1);
```



A state space 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., they represent two individual arcs.

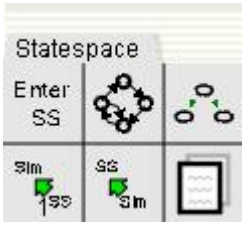


The state space for the dining philosophers is generated in the SS tool in CPN Tools. Currently, there is no support for drawing state spaces in CPN Tools. It should be noted that this state space is rather atypical. Most state spaces are much larger. The present version of the SS tool is able to handle 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 state spaces.

Chapter 2

How to Use the State Space Tools

Tools for calculating a state space, saving a state space report, and transferring states between the simulator and state space tool are found in the state space tool palette:



More information about using the state space tools can be found in the help pages, for example in the online help:
http://wiki.daimi.au.dk/cpntools-help/use_state_space_tool.wiki

Generation of State Space Code

Before a state space can be calculated, it is necessary to generate the **state space code**, i.e., the ML code which is used to calculate and analyse state spaces. To generate the state space code the following steps must be performed (in the specified order):

a) Make sure that all transitions, places, and pages in the net have names. The names are required to be unique and to be alphanumeric ML-identifiers: any sequence of letters, digits, primes (') and underscores (_) starting with a letter.

b) Apply the **Enter State Space** tool to the net, i.e., select "Enter SS" in the state space tool palette, and apply it to one of the pages in the net.

The generation of new nodes progresses in a *breadth- first* fashion. This means that the nodes are 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 6).

For a timed state space 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 state space code for the dining philosopher system. To do this use the CPN model called “DiningPhilosophers”. It is located in the “Sample Nets” directory of the CPN Tools distribution.

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 state space code from scratch. This also means that the state space (if any) is lost. When the modification is made in the simulator it is sufficient to apply the **Enter State Space** tool again.

The state space is calculated for those parts of the net which would participate in a simulation. Please note that, in state spaces, it only makes sense to use code segments in a very limited fashion, e.g., to initialise a CPN model.

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

Generation of the State Space and Scc Graph

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

If the state space is expected to be small (e.g., with a few hundred nodes and arcs), you can simply apply the **Calculate State Space** tool, i.e. select the tool from the state space tool palette and apply it to one of the pages of the net. Otherwise you may need to change the *Stop Options* and/or the *Branching Options* described in Chap. 6.

Many of the query functions in Chap. 4 use the Scc graph (i.e., the strongly connected components of the state space). To calculate the Scc graph you must apply the **Calculate Scc Graph** tool, i.e. select the tool from the state space tool palette and apply it to one of the pages in the net.

The state space and the Scc graph can also be calculated by using the **Evaluate ML** tool to evaluate the following ML functions, which work exactly as the tools in the state space tool palette. This can, e.g., be useful if you want to handle exceptions raised by the CPN model or the Stop options (in Chap. 6):

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

Options for the **Calculate State Space** tool can be found in Chapter 6.

Standard Report

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

- Statistics (size of state space 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).

Using the options for the **Save Report** tool (see Chapter 6), one can specify how much information is to be put in the report. This is done by choosing one or more of the possibilities mentioned above. Home and fairness properties can be included in the report only if the Scc graph has been calculated. For the dining philosopher system the full standard report looks similar to the report below:

```
Statistics
------------------------------------------
  State Space
    Nodes:  11
    Arcs:   30
    Secs:   1
    Status: Full

  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_Chopsticks 1   5         1

  Best Upper Multi-set Bounds
  System'Eat 1          1`ph(1)++ 1`ph(2)++ 1`ph(3)++
                        1`ph(4)++ 1`ph(5)
  System'Think 1        1`ph(1)++ 1`ph(2)++ 1`ph(3)++
                        1`ph(4)++ 1`ph(5)
  System'Unused_Chopsticks 1
                        1`cs(1)++ 1`cs(2)++ 1`cs(3)++
                        1`cs(4)++ 1`cs(5)

  Best Lower Multi-set Bounds
  System'Eat 1                   empty
  System'Think 1                 empty
  System'Unused_Chopsticks 1     empty


 Home Properties
------------------------------------------
  Home Markings:  All
```

```
 Liveness Properties
----------------------------------------
  Dead Markings:  None
  Dead Transitions Instances: None
  Live Transitions Instances: All

 Fairness Properties
----------------------------------------
  System'Put_Down_Chopsticks 1      Impartial
  System'Take_Chopsticks 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. 6.

Statistics Functions

A number of functions can be used to get information about the size of the state space and the size of the Scc graph.

The state space will always have at least one node (even if the **Calculate State Space** tool has not been used). By convention node number 1 represents the initial marking.

If the state space is partial you may extend it by applying the **Calculate State Space** tool once more (perhaps after modifying some of the *Stop Options* or *Branching Options* (see Chapter 6)).

Statistics about the calculation of the state space and Scc graph can 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
```

State Space to Simulator

The **State Space to Simulator** tool (SStoSim) allows you to transfer a state from the state space 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 SS node (to be moved) is specified in the tool option for the **State Space to Simulator** tool.

Simulator to State Space

The **Simulator to State Space** tool (SimToSS) allows you to transfer a simulator state to the state space (where it becomes a node). A green status bubble with a message reports the node number and whether the state already belonged to the state space.

Chapter 3

How to Refer to the Items of a State Space

This chapter describes how you can refer to the items of a state space, 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 state space:

```
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_Chopsticks 1
```

to refer to transition *Take_Chopsticks* 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 a list representation of the multi-set. To obtain a string representation of the marking of a place the st_Mark functions should be used (see *String Representations* below).

For a timed state space the above functions return a timed multi-set (for places with a timed colour 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_Chopsticks (1,{p=ph(3)})
```

to refer to the binding element where transition *Take_Chopsticks* 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 first two of the 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_Chopsticks 1"
st_BE (ArcToBE(18)) "System'Take_Chopsticks 1:{p=ph(4)}"
```

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. 6. 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_Chopsticks"
st_BE (ArcToBE(18))       "Take_Chopsticks: {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          ""
```

A **node descriptor** is a string representation of the information associated with a node in a state space. By default, a node descriptor is a string representation of the node number and the entire net marking that corresponds to the node in the state space. A node descriptor can be obtained using the following function:

```
fun NodeDescriptor              Node -> string
```

The **NodeDescriptor** function uses the st_Mark functions for each of the place instances in the net. Example (the print function prints the string argument):

```
print(NodeDescriptor 7)


7:
System'Think 1: 1`ph(2)++1`ph(3)++1`ph(5)
System'Eat 1: 1`ph(1)++1`ph(4)
System'Unused_Chopsticks 1: 1`cs(3)
```

An **arc descriptor** is a string representation of the information associated with an arc in a state space. An arc descriptor can be obtained using the following function:

```
fun ArcDescriptor               Arc -> string
```

By default, an arc descriptor is a string showing the source and destination nodes of the arc together with the corresponding binding element. Example

```
ArcDescriptor 11
"11:3->10 System'Take_Chopsticks 1: {p=ph(2)}"
```

Here the arc descriptor for arc *11* indicates that the arc goes from node *3* to node *10*, and the binding element is the *Take_Chopsticks* transition on the *first* instance of the page *System* where the variable p is bound to ph(2).

Time Values

The following functions can only be used for **timed state spaces**.

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

```
fun CreationTime                    Node -> IntInf.int
```

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

```
fun OccurrenceTime                  Arc -> IntInf.int
```

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 state spaces 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 written in auxiliary texts – alone or as part of a larger ML expression. The text is evaluated by means of the **Evaluate ML** tool.

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 state space 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 state spaces 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:

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

to calculate an integer |F(n)|. This is done for each node n in the state space, 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 |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:

```
[ph 1,ph 2,ph 3,ph 4,ph 5] : PH ms
```

which is the CPN-ML representation of the multi-set (the elements of the list are not necessarily sorted):

```
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 above representation of the result as a string, evaluate the following ML code:

```
PH.mkstr_ms (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].

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 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 state space. 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.

InitialHomeMarking determines whether the initial marking of the state space 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 M_8 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 state spaces 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_Chopsticks 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_Chopsticks* on the *first* instance of the page *System*. The function also has a chatty version:

```
TIsDead' ([TI.System'Take_Chopsticks 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_Chopsticks (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_Chopsticks* 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 state space. 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_Chopsticks 1]
```

returns true. This tells us that it is impossible to reach a marking such that transition *Take_Chopsticks* 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_Chopsticks (1,{p=ph(3)})]
```

returns true. This tells us that philosopher ph(3) always has a chance to *Take_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_Chopsticks (1,{p=ph(1)}),
Bind.System'Take_Chopsticks (1,{p=ph(2)}),
Bind.System'Take_Chopsticks (1,{p=ph(3)}),
Bind.System'Take_Chopsticks (1,{p=ph(4)}),
Bind.System'Take_Chopsticks (1,{p=ph(5)})]
```

returns true. This tells us that each philosopher always has a chance to *Take_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_Down_Chopsticks 1,
TI.System'Take_Chopsticks 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_Chopsticks 1]
```

returns `Impartial`. This tells us that we cannot have an infinite occurrence sequence unless transition *Take_Chopsticks* 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_Chopsticks (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_Down_Chopsticks 1,
TI.System'Take_Chopsticks 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 a state space and an Scc graph. Then we describe three search functions by which you can traverse the nodes, arcs and strongly connected components of a state space.

Nodes and Arcs

The following functions allow you to "move" between adjacent nodes and arcs of the state space:

```
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.6) 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 state space arc. We have an Scc arc for each state space 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 a state space node into the Scc to which it belongs. The second function maps a state space 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 state space nodes that belong to the Scc. Finally, the fourth function maps an Scc node into the state space 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
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 state space. 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 state space 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.
Start value	This argument specifies a constant, of some type B.
Combination function	This argument specifies a function. It maps from A×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
```

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
```

to denote the set of all nodes in the state space. 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 ::)
```

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 ::)
```

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 state space:

```
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))= [])
```

If the state space 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 MaxTIEnabled () : 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.all())
      end)
```

Checking whether there are nodes which violate the linear invariant:

M(Unused_Chopsticks) ++ Chopsticks(M(Eat)) = CS.all()

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

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

SearchArcs

The function **SearchArcs** traverses the arcs of the state space. 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_Chopsticks* 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_Chopsticks (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 state space. At each strongly connected component 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 SearchSccs in a way which is totally analogous to SearchNodes and SearchArcs. The arguments have the following types:

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 Change Options

The SS tool has a number of **options**. Some options determine how the string representation functions work, other options determine the way in which the tools in the state space tool palette work.

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_Chopsticks 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).

The st-functions are used for the standard reports generated by the **Save Report** tool and for the contents of SS 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 ^ "\n")
```

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 ^ "\n")
```

Node and Arc Descriptor Options

The *Node Descriptor Options* determine the contents of the SS node descriptors (see Chapter 3). They are changed by the following ML function (the value indicates the system defaults for the dining philosopher system):

```
OGSet.NodeDescriptorOptions(
            fn n =>
            (st_Node n)^"\n"^
            (st_Mark.System'Unused_Chopsticks 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 SS arc descriptors. They are changed by the following ML function (the value indicates the system defaults for the dining philosopher system):

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

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.

Options for Calculating a State Space

There are a number of options for determining how a state space is calculated. There are stop options and branching options, both of which are described below. Each kind of option can be changed by modifying the appropriate option for the **Calculate State Space (CalcSS)** tool in the index of CPN Tools. The figure below shows the stop options and the branching options for calculating a state space. The first four options are stop options, and the last three options are branching options.

Information about changing options for the **Calculate State Space** tool can be found in the help page for the tool, for example in the online help: http://wiki.daimi.au.dk/cpntools-help/calculate_state_space.wiki

Stop Options

The *Stop Options* allow you to determine when the calculation of a state space stops. This happens when all nodes have been processed or when one of the stop options becomes satisfied. The stop options can be changed by modifying tool options for the **Calculate State Space** tool in the index of CPN Tools. The stop options can also be 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 apply the **Calculate State Space** tool. This means that you can extend a state space 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 state space 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 state space 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” state space 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 SS tool need not calculate all the successors of a node. The node is then said to be only *partially processed*. The options can be changed by modifying the appropriate options for

the **Calculate State Space** tool. They can also be changed by evaluating the following ML function (the values indicate the system defaults):

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

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 apply the **Calculate State Space** tool. 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 state space, some of the partially processed nodes may become *fully processed*.

The generation of new nodes progresses in a *breadth- 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 state space the processing order is determined by the creation time (i.e., the model time at which the individual markings start to exist).

Save Report Options

As mentioned in Chapter 2, the standard state space report provides information about:

- Statistics (size of state space 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 tool options for the **Save Report** tool determine which kind of information will be included in a state space report. The figure below shows the tool options for the **Save Report** tool:

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