Dining Philosophers

Abstract
This is a small toy example which is well-suited as an introduction to occurrence graphs. The analysis of the occurrence graph is described in great detail and a large number of different queries are illustrated.

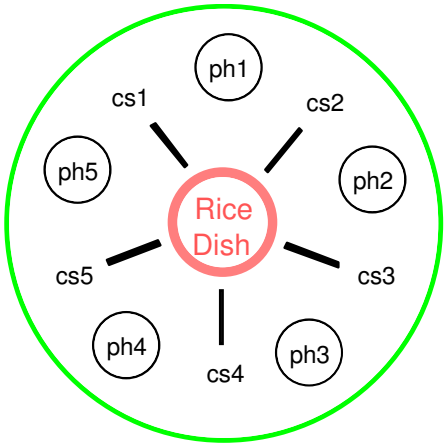
The CPN model describes how a number of processes (philosophers) share common resources (chopsticks). The Dining Philosophers is one of the traditional examples used by computer scientists to illustrate new concepts in the area of synchronisation and concurrency.

The example is taken from Sect. 1.6 of Vol. 2 of the CPN book. Most of the detailed explanation is taken from the Occurrence Graph Manual, which uses the Dining Philosophers as its main example.

Developed and Maintained by:
Kurt Jensen, Aarhus University, Denmark (kjensen@daimi.au.dk).

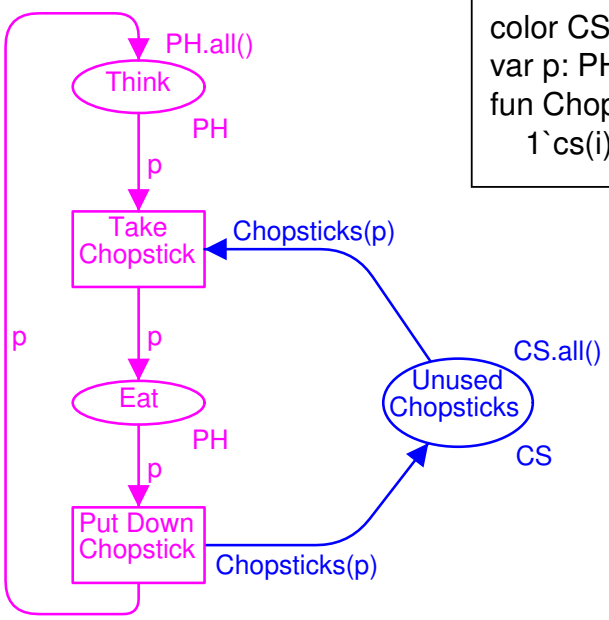
CPN Model

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.

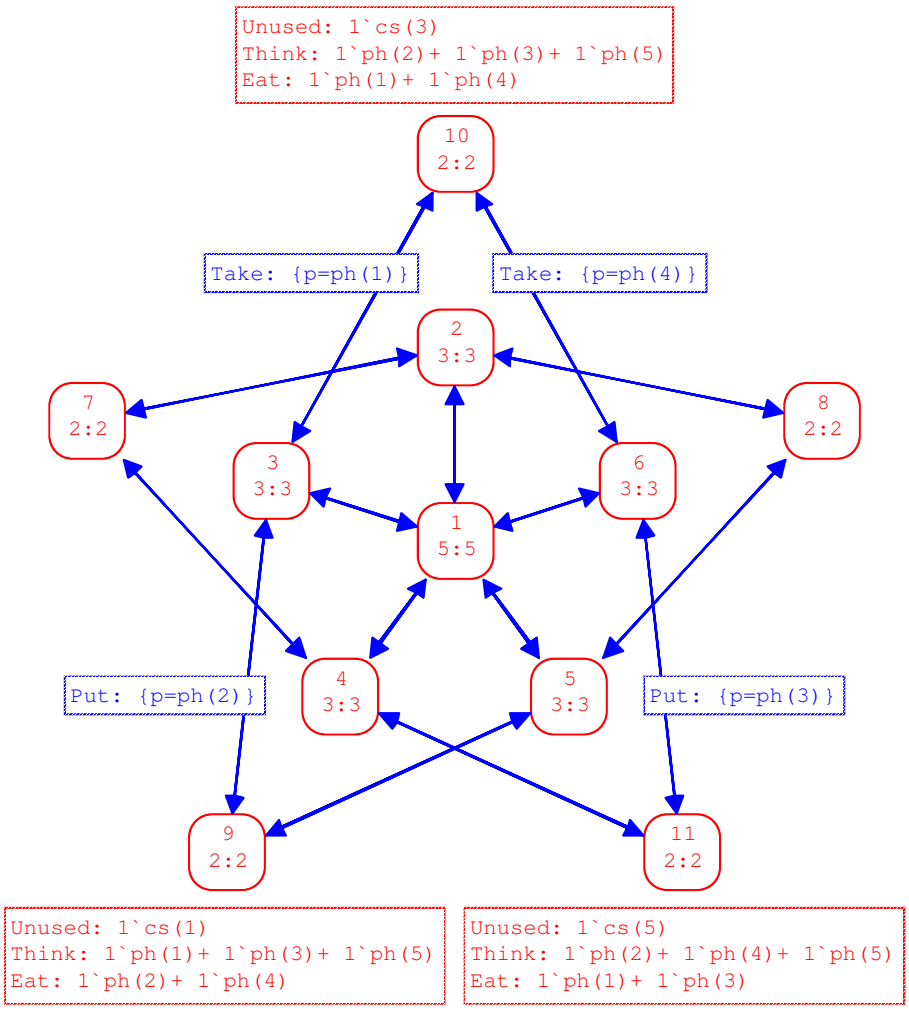


The philosopher system is modelled by the CP-net shown below. The PH colour set represents the philosophers, while the CS colour set represents the chopsticks. The function *Chopsticks* maps each philosopher into the two chopsticks next to him.

```
val n = 5;
color PH = index ph with 1..n;
color 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);
```



An occurrence graph for the dining philosophers is shown below. The current version of CPN Tools does not include facilities for drawing O-graphs. Each node represents a reachable marking, while each arc represents the occurrence of a 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 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).



The standard report looks as shown below. To improve the readability of the multi-set bounds we have substituted *PH* for 1`ph(1)+ 1`ph(2)+ 1`ph(3)+ 1`ph(4)+1`ph(5) and *CS* for 1`cs(1)+1`cs(2)+1`cs(3)+1`cs(4)+1`cs(5).

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

  Scc Graph
    Nodes:  1
    Arcs:   0
    Secs:   0

 Boundedness Properties
--------------------------------
  Best Integer Bounds
         Upper  Lower
  Eat      2      0
  Think    5      3
  Unused   5      1

  Best Upper Multi-set Bounds
  Eat      PH
  Think    PH
  Unused   CS

  Best Lower Multi-set Bounds
  Eat      empty
  Think    empty
  Unused   empty

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

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

```
Fairness Properties
-------------------------------
  Put     Impartial
  Take    Impartial
```

Below we show a large number of standard queries. They are all taken from the Occurrence Graph Manual:

`Reachable` determines whether there exists an occurrence sequence from the marking of the first node to the marking of the second.

```
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.

```
AllReachable ()
```

returns true.

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 occurrence graph, and the maximum of the calculated integers is returned. The query:

```
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)|.

```
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.

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

returns:

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

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.

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

returns the empty multi-set, represented as an empty list. 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).

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.

```
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.

```
MinimalHomeSpace ()
```

returns 1.

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.

```
HomeMarking (6)
```

returns true. The function also has a chatty version.

ListHomeMarkings returns a list with all those nodes that are home markings.

```
ListHomeMarkings ()
```

returns a list which contains all 11 nodes of the occurrence graph.

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.

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

```
HomeMarkingExists ()
```

returns true.

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`.

```
InitialHomeMarking ()
```

returns true.

DeadMarking determines whether the marking of the specified node is dead, i.e., has no enabled binding elements.

```
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.

```
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.

```
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).

```
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.

```
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.

```
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).

```
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.

```
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.

```
ListLiveTIs ()
```

returns:

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

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

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

```
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).

```
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.

```
ListImpartialTIs ()
```

returns the list:

```
[System'Put 1,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 those transition instances that are fair and just, respectively. Impartiality implies fairness which in turn implies justice. Hence, it is obvious that :

```
ListFairTIs ()
```

```
ListJustTIs ()
```

both return the list:

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

Below we show some model dependent queries. They are all taken from the Occurrence Graph Manual:

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 =>
         List.length(remdupl(List.map
                             ArcToTI(OutArcs n))),
     0,
     Int.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 next (ph i: PH) : PH
  = if i<n then ph(i+1) else ph 1;
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 that violate the linear invariant:

M(Unused) + Chopsticks(M(Eat)) = CS

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 1 n +
     ext_ms Chopsticks (Mark.System'Eat 1 n) <><>
CS)
```

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 dining philosophers system the O-graph grows relatively slow – when we increase the number of philosophers:

|PH|	Nodes	Arcs
2	3	4
3	4	6
4	7	16
5	11	30
6	18	60
7	29	112
8	47	208
9	76	378
10	123	680
15	1,364	11,310