4 Application of Petri Nets for the Modelling

4.1 Methodology for the Modelling with Petri Nets

Construction of a Petri Net as model for a concurrently working system which already exists or which is yet to build is mostly an iterative process. With the help of simulation and particularly with analysis we can detect properties of our Petri Net model and compare with required system properties. Differences may be have causes in incorrect or incomplete modelling or give hints that the system do not work properly respectively that the system is not correct designed. Modifications of the Petri Net are the consequences. This procedure is outlined in figure 4.1.1.

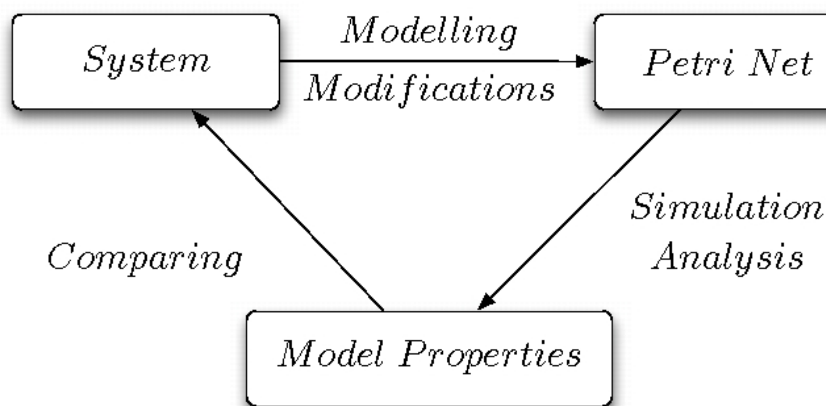

Fig. 4.1.1: Iterative proceeding modelling Petri Nets

The first-time modelling of a Petri Net needs a **stepwise proceeding**:

1. Step:

a) Interpretation for places and transitions,
i.e. which elements of a real system correspond to passive elements and which to active elements?
(possible interpretations can be found in table 4.1),

b) formulation of local causal connections of places and transitions,
i.e. definition of pre- and post-places of transitions resp.
pre- and post-transitions of a places,

2. Step: Design of a connected process net using subnets,

3. Step: Specifying an initial marking and a transition rule
to analyze the dynamic system behaviour.

Possible interpretations for places and transitions shows table 4.1:

Places (passive elemens)	**Transitions** (active elements)
Memory	Processors
Propositions	Proofs
Reagents	Chemical Reactions
Languages	Translators
Materials	Production activities
States	Actions

table 4.1: Interpretations for places and transitions

4.2 Examples for the Modelling with Petri Nets

The method for modelling Petri Nets like shown in the previous chapter would be demonstrated now by some examples.

4.2.1 Sender-Receiver-Model

As a modification of the consumer-producer-system we consider now a message system consisting of a sender a receiver, and a transmission channel (a real example for this is a pneumatic delivery system):

1st variant

1) A sender goes from a stand by state in to a ready state an vice versa.

2) In the ready state the sender can send *one* message or go in its final state.

3) The sent messages are outputted from the sender to a channel which can contain maximal *two* messages.

4) A receiver changes from stand by state to the ready state and vice versa.

5) In the ready state the receiver either receives *one* message or goes to its final state

6) In the initial situation the sender and the receiver are in its ready state, and the message channel is *empty*

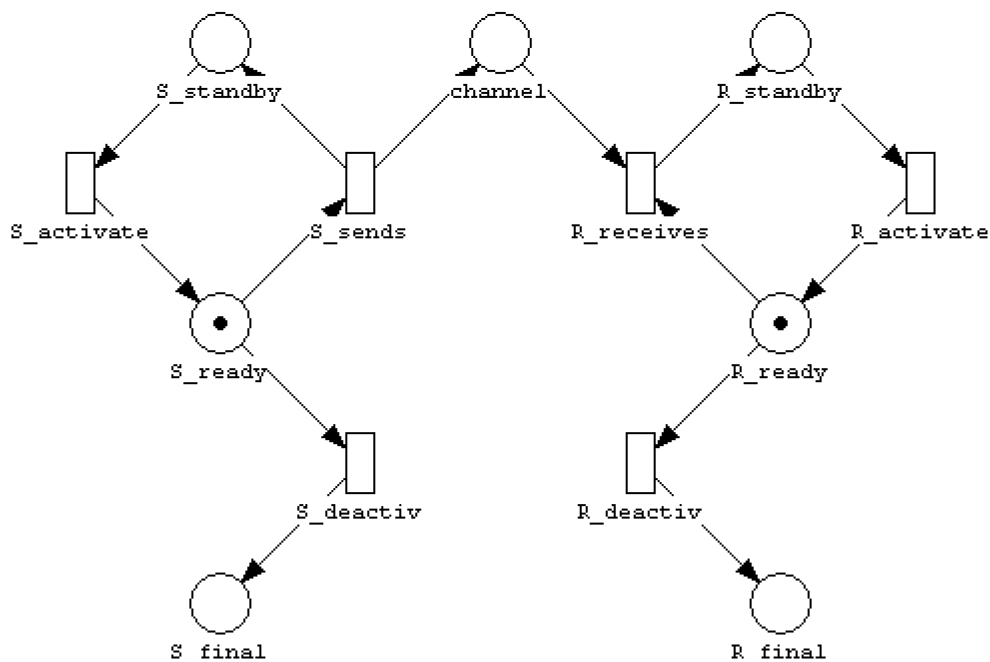

Fig. 4.2.1: Sender-Receiver-Model - 1st variant

Figure 4.2.1 shows a realization of 1st variant as Place-Transition-Net. However this variant is **disappointing** because the receiver can go to its final state even if

(i) the sender is not yet in its final state
(i.e. it is possible that he sends later) resp.

(ii) the channel is not yet empty.

To exclude these state transitions we expand our model as follows:

2nd variant

1) to 6) like variant 1,

7) We add a complementary channel between the sender and the receiver. With this channel we can count the number of *free* capacities of the message channel (e.g. the number of free boxes for the pneumatic delivery system). The sender reduces the free capacities by *one* sending a message while the receiver raises the free capacities by *one* receiving a message. Otherwise the receiver should only be deactivated if *all* capacities are *free* it means that in the message channel there is no message. At the beginning the maximal number of free capacities is *two*.

Comment:

The test of a place if it has no tokens is called the **null test of a place**. The described method introducing a co-place for the null test works only if the capacity of tested place is finite because the co-place obtains as an initial marking so many tokens how the difference between the maximal capacity of tested place and its initial marking. Such a restriction of the capacity is given for the most practical applications.

8) We introduce a next new channel which informs the receiver that the sender is inactive. The receiver should only change to its final state if he is ready and the sender is inactive.

Figure 4.2.2 shows the Sender-Receiver-Model with the additional channels chann free and S inactive and the needed arcs.

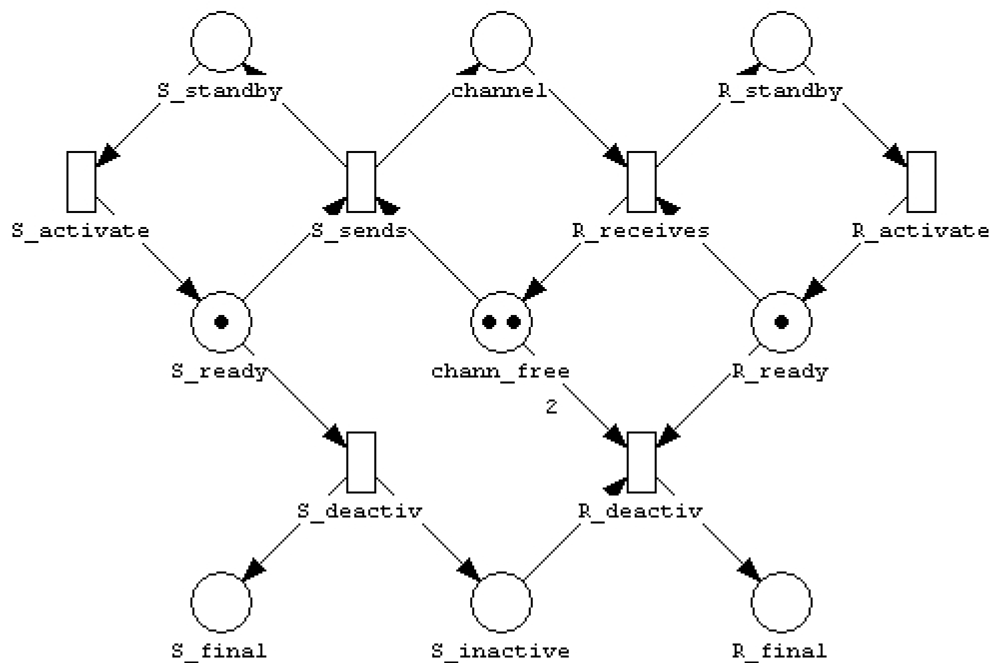

Fig. 4.2.2: Sender-Receiver-Model - 2nd variant

In the end the sender-receiver model is embedded in an environment which ensures its restart.

3rd variant:

9) When sender and receiver go into their final state, they send out a turn off signal to the environment.

10) The restart of sender and receiver takes place synchronously, which means that only if both turn off signals are present, then both may restart again.

11) For the next preparation of operations both sender and receiver will turn into the stand by modus. The free channel will be brought up to *full capacity*.

The complete model is described in figure 4.2.3 in detail.

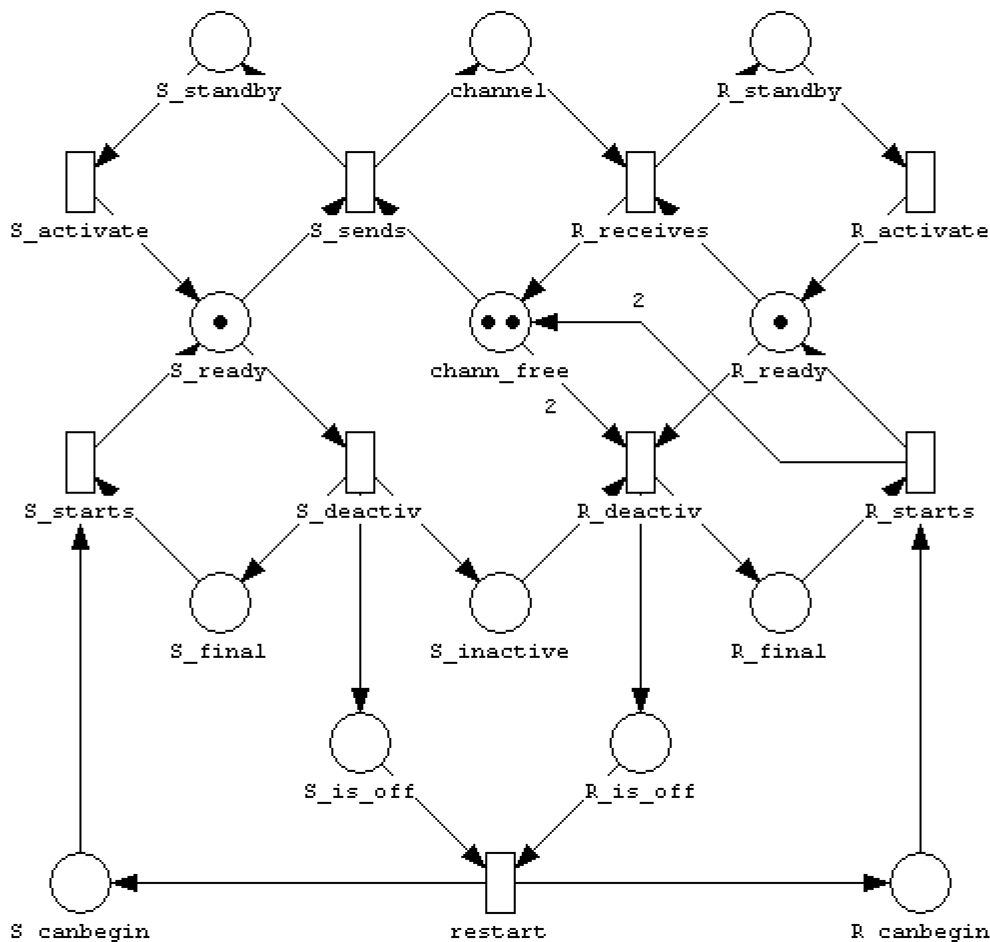

Fig. 4.2.3: Sender-Receiver-Model - 3rd variant

If the sender-receiver-model is modelled correctly, it consists of the following properties:

(P1) Sender and receiver are exactly in one state out of the set {*S_standby, S_ready, S_final*} or {R_*standby, R_ready, R_final*}.

(P2) The place *channel* contains not more than two tokens.

(P3) The sender (resp. receiver) is at rest, when it has given a certain signal to the environment. The final state can only be activated again by a signal of the environment.

(P4) The sender in the final state can only be activated again, if the receiver is also in final state.

(P5) The receiver is totally depending on the sender in its decision to receive or to go in its final state. In the case of *R_ready* there will never be a conflict because *R_receives* and *R_deactiv* can never be enabled simultaneously.

(E6) The receiver can only turn to its final state, if the channel is empty and the sender is in its final state.

These properties can be verified using **invariants** (see chapter 5.9).
In order to model a sender-receiver-model with unbounded channel capacity correctly, the class of place-transition-net is not appropriate. For this higher Petri Net classes are needed (see chapter 7).

4.2.2 Five Philosophers Dining Problem

Another, very well-known problem is the **Five Philosophers Dining Problem** which goes back to *Dijkstra.*

Five philosophers live together in a house. Their lives consist of thinking, eating and sleeping alternating. While the result of their thinking and sleeping is of lower interest, a real problem arises during their meals: The philosophers dine together at a round table. Each of them has his own fixed place at the dinner table and his own plate. One day they eat a certain kind of fish which can only be consumed by using two forks. The problem is: Only one fork lies between each plate (see figure 4.2.4):

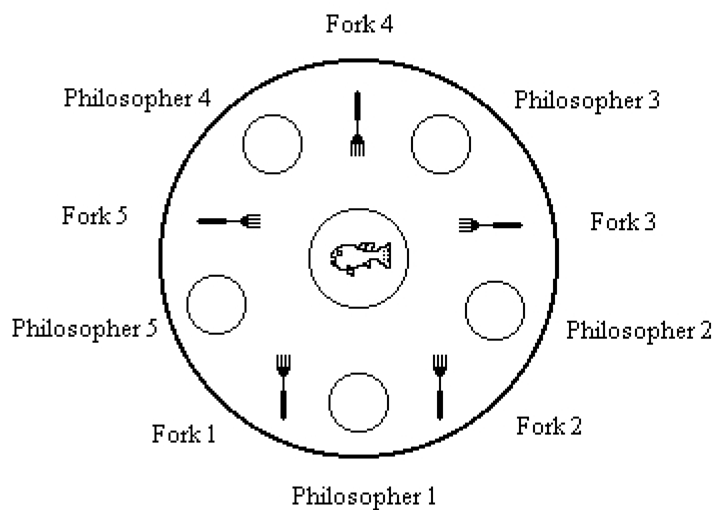

Fig. 4.2.4: Five Philosophers eating Fish

Task: Develop an optimal algorithm - without proposing individual lunch hours - which ensures that no philosopher dies of starvation. Of course it is assumed the each of the eating philosophers has eaten enough, cleans his forks, and puts them back on the table eventually.

Variant 1 a):

A hungry philosopher picks up the fork on his left side. As soon as he has this fork in his hand, he grabs the fork on his right side. Only when he holds both of the forks in his hands, he is allowed to start eating. If not he must wait.

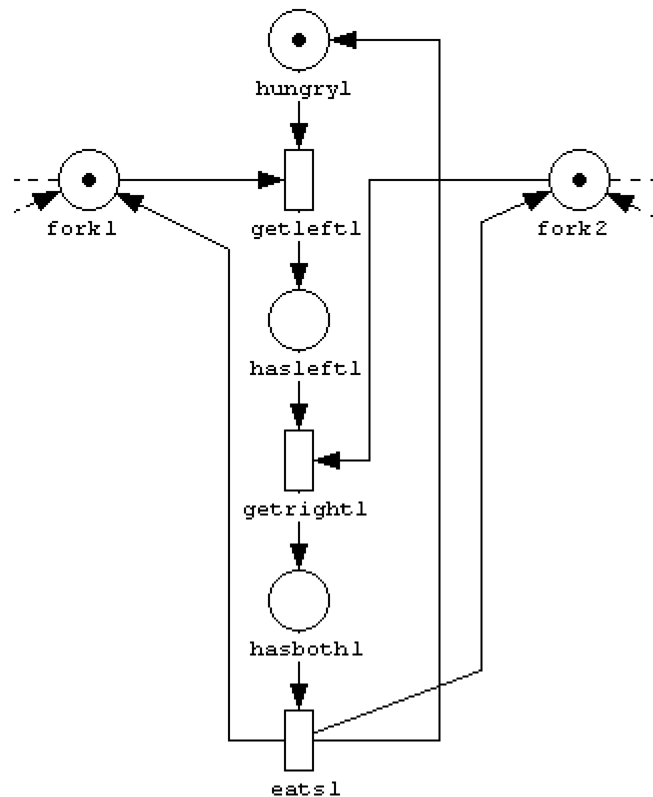

Fig. 4.2.5: Life cycle of Philosopher 1 for variant 1 a)

We get the whole solution for variant 1 by adding the single life cycles of the philosophers according to figure 4.2.5, whereas the same labelled conditions fork1 to fork5 are merged (see dashed lines). This yields that two neighbouring philosophers cannot eat simultaneously.

Analysis:

If all philosophers are hungry at the same time and all of them take their left fork, then nobody get its right fork. ==> All philosophers starve.
This situation is called a (total) ***Deadlock***.

Variant 1 b):

This is like variant 1 a) but with an additional rule:
If a philosopher who has the left fork didn't get the right fork, then he can put back the left fork. This solution is shown in figure 4.2.6.

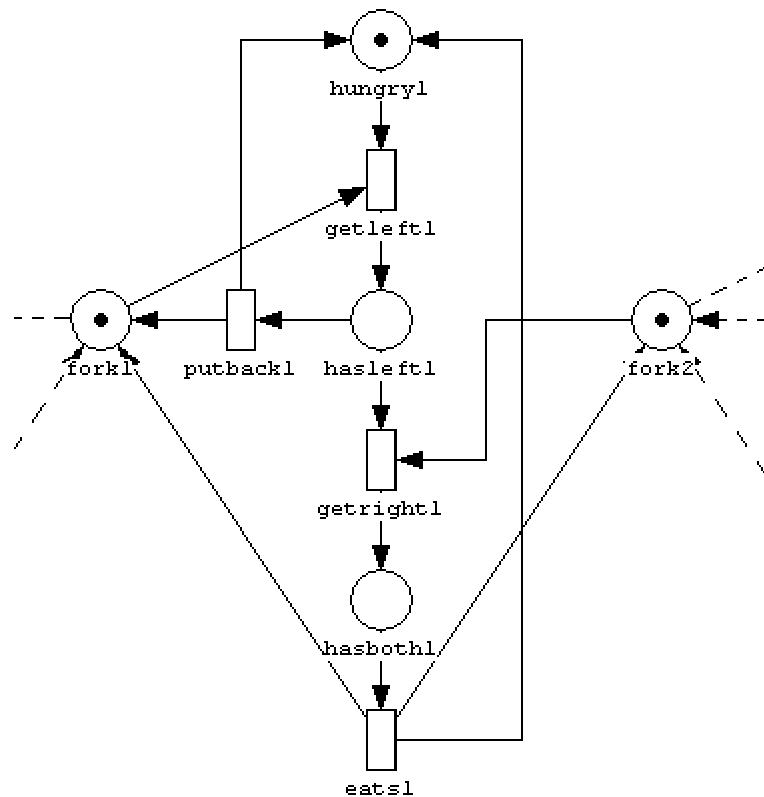

Fig. 4.2.6: Life cycle of Philosopher 1 for variant 1 b)

Analysis:

Now there is no deadlock. But if all philosophers take the left fork in the same moment and put them back also in the same moment, then all philosophers must starve. The system is busy with itself and the fireable transitions "eats*i*" are not involved. We are calling this situation a ***livelock***.

Variant 2:

It is like variant 1a) but with the additional rule that the access to the forks is exclusively allowed only one philosopher, i.e. if a philosopher has both forks than he gives back the exclusive right, and another philosopher can take the forks.

This variant is demonstrated for philosopher 1 in figure 4.2.7.

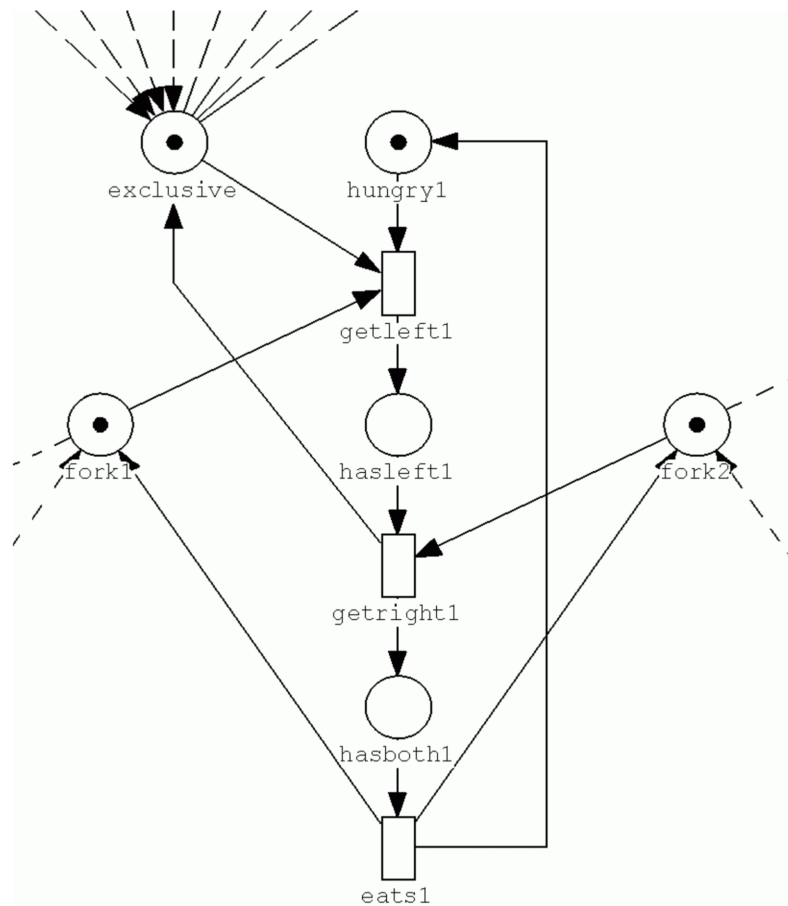

Fig. 4.2.7: Life cycle of Philosopher 1 for variant 2

Analysis:

The exclusiveness of the access ensures that no deadlock can happen. The algorithm, however, is **not optimal**:

Because a philosopher can be prevented from eating even though his both forks are not used by someone else. For example: Philosopher 1 is eating. Then both the forks 1 and 2 are being used. Now philosopher 5 becomes hungry and wants to use the fork 5 on his left side. Unfortunately, he is not allowed to use the fork on his right side, because this fork is being used by philosopher 1. If philosopher 3 comes to the table, he must wait - even though both of his forks 3 and 4 are not being used by someone else, because philosopher 5 has got exclusive rights. Besides hindering philosopher 3, philosopher 5 prevents also philosopher 4 from eating, because he takes away fork 5 - although he has no use for this fork.

Conclusions:

Every philosopher should only have access to a fork, if he is also able to get access to the other fork. This strategy is discussed as in variant 3.

Variant 3:

Every philosopher takes both forks simultaneously - as long as they are accessible.
This solution is illustrated as a complete Petri Net in figure 4.2.8.

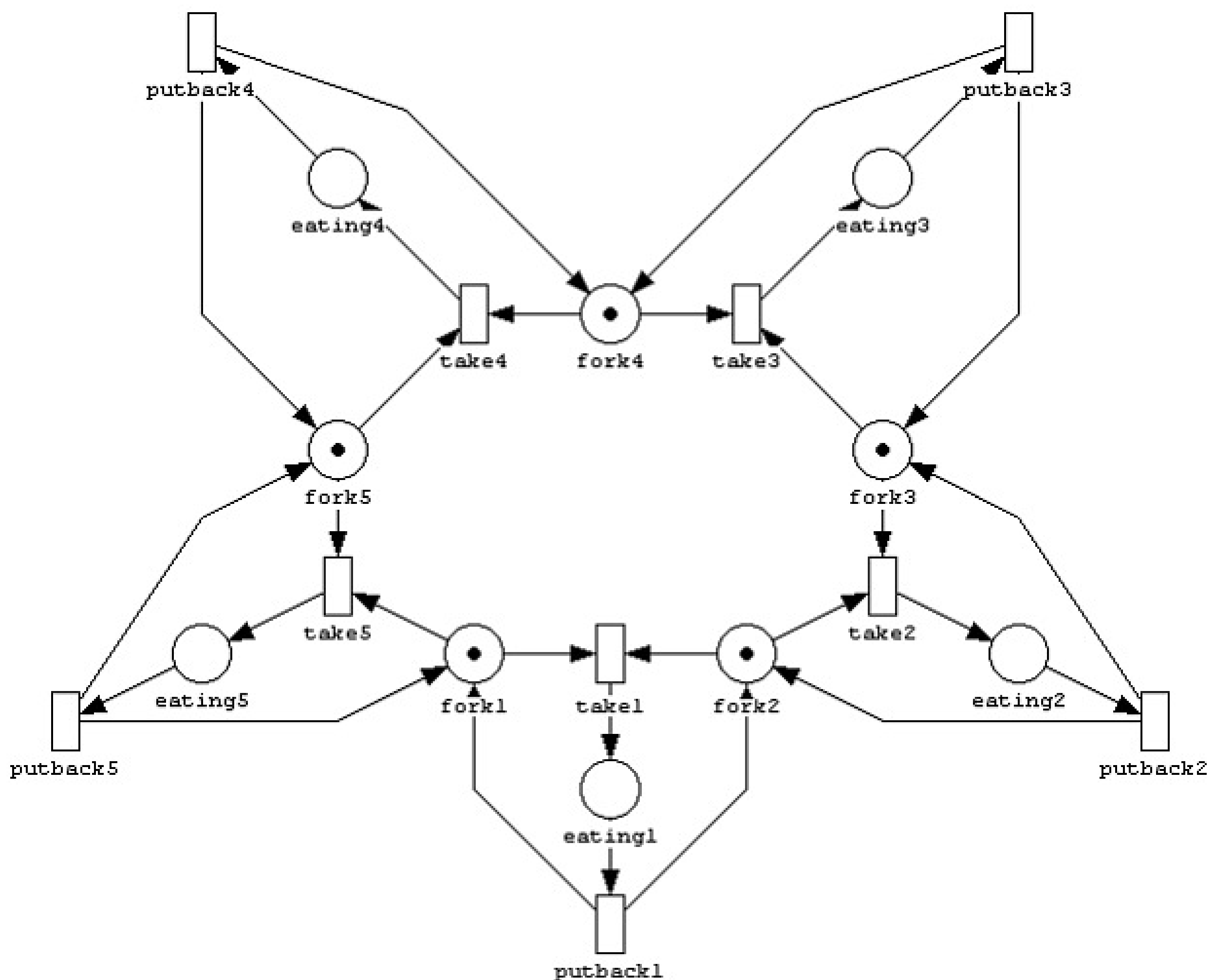

Fig. 4.2.8: Life cycle of Philosophers for variant 3

Analysis:

This modelling facilitates a better utilization of resources than variant 2. Now up to two philosophers are able to eat at the same time. But there is no guarantee for every life cycle of philosophers (problem of fairness). Every philosopher should be allowed to eat

eventually, so that he doesn’t starve. But if four philosophers cooperate “viciously”, one philosopher might starve. For example philosopher 3 comes to the table too late, the following process may happen:
Philosopher 2 eats, philosopher 4 eats, philosopher 2 puts both forks back, philosopher 1 eats, philosopher 1 puts forks back, philosopher 2 eats, philosopher 4 puts forks back, philosopher 5 eats, philosopher 5 puts forks back and so on - meaning philosopher 3 is starving.

Variant 4 (Erlanger Solution, HOFMANN 1974):

In order to avoid the starvation of anyone, definitely, one needs a condition defines that one philosopher had been waiting longer for he food than his neighbour. When both neighbours of a hungry philosopher have eaten, the hungry philosopher gets very hungry. Now one has to make sure that no neighbour of a very hungry philosopher is allowed to eat. For the modelling of this solution certain priorities must be used, which are only available using the so called ***Priority-Nets*** (see chapter 7.4).

Comment:

Finally it should be stated that the philosophers’ problem finds his analogy in connected computers in a ring network. Every processor contains memory and can access to the memory of a connected computer - as long as no other processor is accessing to the memory of this computer.

4.2.3 Farmer Wolf Goat Cabbage Problem

A farmer stands at the south bank of a river. He has a wolf, a goat, and a cabbage. He wants to cross the river to the north bank by boat. The farmer can only go by himself or together with either the wolf or the goat or the cabbage.
If the goat and the wolf stay behind alone, the wolf will eat the goat.
If the goat stays behind with the cabbage, the goat will eat the cabbage.
How must the farmer organize the crossing of the river if he wants to keep goat and cabbage in sound condition?

a) At first we model this transfer problem as a Condition-Event-Net - without taking additional conditions into consideration.

b) Then we take the additional conditions into consideration.

Solution for a):
The following conditions are defined:

fs - Farmer in the south
ws - Wolf in the south
gs - goat in the south
cs - cabbage in the south
fn - farmer in the north
wn - Wolf in the north
gn - goat in the north
cn - cabbage in the north

and the following events are defined:

tfn - farmer goes to the north
tfwn - farmer goes with the wolf to the north
tfgn - farmer goes with the goat to the north
tfcn - farmer goes with the cabbage to the north
tfs - farmer goes to the south
tfws - farmer goes with the wolf to the south
tfgs - farmer goes with the goat to the south
tfcs - farmer goes with the cabbage to the south

This solution is illustrated in figure 4.2.9.

Fig. 4.2.9: Farmer-wolf-goat-cabbage-problem without side conditions

Solution for b):

Now the side conditions are realized with the following conditions:

wzns - Wolf and goat not alone in the south
wznn - Wolf and goat not alone in the north
zkns - Goat and cabbage not alone in the south
zknn - Goat and cabbage not alone in the north

The Figure 4.2.10 illustrates this solution.

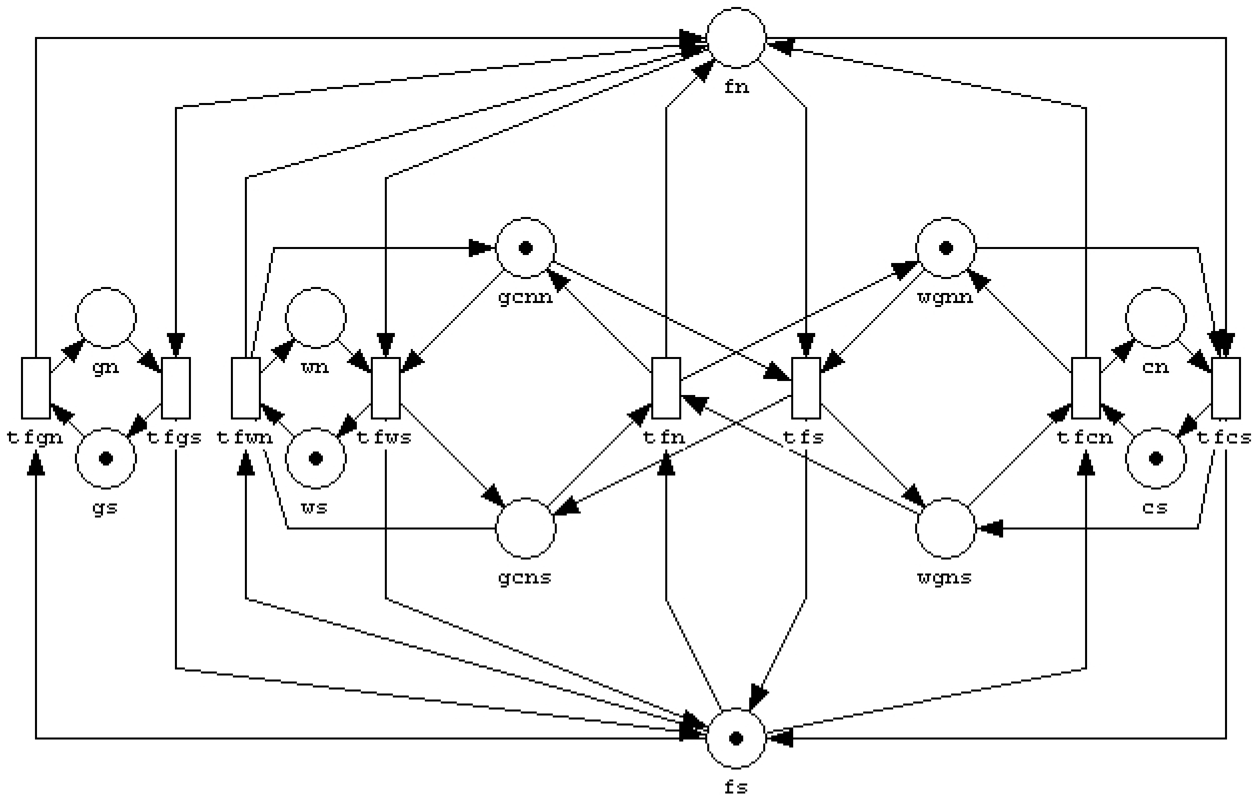

Fig. 4.2.10: Farmer-wolf-goat-cabbage-problem with side conditions

There are two different solutions with a minimal number of transfers:

1. Solution: tfgn, tfs, *tfwn*, tfgs, *tfcn*, tfs, tfgn,

2. Solution: tfgn, tfs, *tfcn,* tfgs, *tfwn*, tfs, tfgn.