Course Business Information Systems
Exercises

Wil van der Aalst  Massimiliano de Leoni  Boudewijn van Dongen
Christian Stahl

March 5, 2015

Contents

1 Transition systems  2
2 Petri nets  11
3 Colored Petri nets  38
4 Colored Petri nets: Hierarchy  49
5 Functions  70
6 Petri nets standard properties  77
7 Coverability graphs and fairness  85
8 Place- and transition invariants  93
1 Transition systems

Exercise 1.1 The crew of an airplane consists of three members: a captain, a pilot, and a navigator. Anyone is either working or resting and can change from either state into the other at any moment, provided the following security measures are met:

- At least one crew member is working at any time;
- Not all crew members are working at the same time; and
- If the captain is resting, the pilot must be working.

Model the described situation and draw the state-transition diagram.

Exercise 1.2 When a scientific paper has been sent to a journal, the editor has to decide if the paper is good enough to be published. To this end, the editor asks external reviewers to judge the paper. First, he sends the paper to two reviewers who rate the paper acceptable or not acceptable. If they both judge the paper the same way, the editor takes over their judgment. Otherwise, if the reviewers have different opinions, the editor sends the paper to a third reviewer and uses his judgment. Model this reviewing process as a transition system and draw the state-transition diagram.

Exercise 1.3 A Dutch traffic light is an example of a system with three possible states: $R$ (red), $G$ (green), and $O$ (orange). Model a T-junction with three traffic lights (see Figure 1) as a transition system and draw the state-transition diagram. The traffic lights are programmed in such a way that at least two lights are red at the same time; that is, for at most one direction of the traffic, the traffic light can be green or orange. (Hint: Represent each state by a combination of three colors.) Of course, as in any T-junction, there is a fair alternation in the traffic lights to become green: a traffic light can become green a second time only after every other traffic light has become green and exactly once.

![Figure 1: A T-junction with three traffic lights.](image)

Exercise 1.4 To improve the traffic flow, the traffic light system of the previous exercise is upgraded to a situation with five lights; see Figure 2. The goal is to program the traffic light system such that crossing cars coming from different directions cannot have a green light at the same time. Furthermore, if at any point lights turn orange, they must first turn red before other lights can change.

![Figure 2: A T-junction with five traffic lights.](image)
Exercise 1.5 The score in a game of tennis is calculated as follows. The first player who wins four rallies in total and at least two rallies more than the opponent wins the game. If both players have won three rallies (40−40), then the player who wins the next rally gets the advantage. If this player wins another rally, she wins the game. If she loses the next rally, she loses the advantage, and the score is equal to the situation in which both players have won three rallies (40−40). Model this system as a transition system and draw the state-transition diagram for the complete game.

Exercise 1.6 Consider the process in a restaurant. After customers enter the restaurant, a waiter assigns them a table and gives them the menu. The customers then order, and the waiter writes down this order, takes the menu, and delivers the order to the kitchen. A cook prepares the order, and the waiter brings it to the customers who then consume it. If the appetite of the customers is not yet satisfied, they can call the waiter and ask for the menu again (after which the entire process repeats itself). If the customers are satisfied, they call the waiter and ask for the check. When that check arrives, they pay and leave.

1. Give the transition system that models the behavior of a customer and the transition system that models the behavior of a waiter. Draw both state-transition diagrams.

2. If you want to make one state-transition diagram in which you describe the behaviors of a customer and a waiter, how many states do you need?

Exercise 1.7 A hospital has an outdated information system to support the financial administration. To replace the current information system, the hospital starts a large automation project. At the start of the project, the functional requirements are collected. Next, the required equipment (hardware) and, at the same time, the required software is purchased. When the hardware and the software have been acquired, the information system can be integrated. After testing the information system, the project is finished. Model this process as a transition system and draw the state-transition diagram.

Exercise 1.8 We can consider the building of a house as a project. The main activities are: start project, arrange building license, make foundation, make frames, make roof, build walls, install roof, and end of project. Model this building project as a transition system and draw the state-transition diagram.

Exercise 1.9 Draw the state-transition diagram for a washing machine with state space \( S = \{\text{off}, \text{defective}, \text{pre-wash}, \text{main\_wash}, \text{rinse}, \text{whiz}\} \) with state 0 = off, and transition relation \( TR = \{(\text{pre-wash}, \text{defective}), (\text{main\_wash}, \text{defective}), (\text{rinse}, \text{defective}), (\text{whiz}, \text{defective}), (\text{off}, \text{pre-wash}), (\text{pre-wash}, \text{rinse}), (\text{rinse}, \text{main\_wash}), (\text{off}, \text{main\_wash}), (\text{main\_wash}, \text{rinse}), (\text{rinse}, \text{off}), (\text{rinse, whiz}), (\text{whiz, off})\} \).

Exercise 1.10 The behavior of the washing machine in Exercise 1.9 is not completely as what we would expect of a washing machine in reality, because transition sequences \( (\text{off, pre-wash, rinse, off}) \) and \( (\text{off, main\_wash, rinse, main\_wash, rinse, main\_wash, rinse, ...}) \) are possible. Adjust the state space and the transition relation such that these transition sequences are not possible any more. Draw the improved state-transition diagram.

Exercise 1.11 Consider a transition system with state space \( S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\} \) and with transition relation \( TR = \{(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (5, 0), (5, 4), (4, 4), (6, 7), (7, 6), (7, 8)\} \).

1. Draw the state-transition diagram.

2. Which states are reachable from the initial state 0?

3. List three different transition sequences that start at state 0.

4. Does the system have a terminal state?
Exercise 1.12 We consider the problem of the dining philosophers as introduced by Dijkstra: “Five philosophers, numbered from 0 through 4 are living in a house where the table laid for them, each philosopher having his own place at the table: Their only problem besides those of philosophy is that the dish served is a very difficult kind of spaghetti, that has to be eaten with two forks. There are two forks next to each plate, so that presents no difficulty: as a consequence, however, no two neighbours may be eating simultaneously.” Figure 3 illustrates this example.

Figure 3: Five philosophers $ph_1, \ldots, ph_5$ eating with forks $f_1, \ldots, f_5$.

1. Model the philosophers as a transition system. It is assumed that each philosopher simultaneously (and indivisibly) picks up his pair of forks. Analogously, he puts them down in a single indivisible action. Each philosopher may be in two states think and eat.

2. Does the model have terminal states (i.e., deadlocks)?

Exercise 1.13 Figure 4 depicts a simplified remote control of a TV. It has six buttons to choose a channel, one button to regulate the volume, one button to mute the sound (and to turn it on again), and one button to switch the TV off. We consider the remote control and the corresponding TV as a system and assume that the possible states of this system are controlled by the buttons on the remote control.

1. Describe all possible states of this system.

2. The transition relation is too large to be depicted as a state-transition diagram. Give therefore examples of possible and impossible transitions. You should especially pay attention to switching the TV on and to the use of the volume button in combination with the mute button.

Figure 4: The remote control of a TV.
Solutions

Solution 1.1

This solution is just an example. There are many possible solutions to any given modeling exercise.

We denote a state as a triple \((x, y, z)\) where \(x\) specifies the state of the captain, \(y\) the state of the pilot, and \(z\) the state of the navigator. Formally, we have \(S = \{(x, y, z) \mid x, y, z \in \{r, w\} \land (x, y, z) \notin \{(r, r, r), (w, w, w), (r, r, w)\}\}\) and \(r\) denotes resting and \(w\) waiting. The three security measures guarantee that states \((r, r, r)\), \((w, w, w)\), and \((r, r, w)\) are not reachable. The state-transition diagram is depicted in Figure 5; we assume that only one crew member can change its state at any time. Note that there is no initial marking specified.

\[
\begin{align*}
(r, w, w) & \rightarrow (r, w, r) \\
(w, w, r) & \rightarrow (w, w, r) \\
(w, r, r) & \rightarrow (w, r, w) \\
\end{align*}
\]

Figure 5: State-transition diagram of Exercise 1.1.

Solution 1.2

This solution is just an example. There are many possible solutions to any given modeling exercise.

The state-transition diagram is depicted in Figure 6. We make the assumption that the review requests are sent concurrently. In addition, not the judgement of the individual reviewer is important, but the number of rejects and accepts. For this reason, we do not link a judgement to a reviewer.

\[
\begin{align*}
\text{submitted} & \rightarrow \text{sntToR1} \\
\text{sntToR2} & \rightarrow \text{sntToBoth} \\
\text{acc} & \rightarrow \text{accPaper} \\
\text{acc+acc} & \rightarrow \text{acc+acc} \\
\text{acc+rej} & \rightarrow \text{acc+rej} \\
\text{rej} & \rightarrow \text{rejectPaper} \\
\text{rej+rej} & \rightarrow \text{rej+rej} \\
\end{align*}
\]

Figure 6: State-transition diagram of Exercise 1.2.

Solution 1.3

This solution is just an example. There are many possible solutions to any given modeling exercise.

We can describe each state of the T-junction by a triple \((1, 2, 3)\) where each of the three numbers denotes the respective traffic light of Figure 1. The set of states can be described by

\[
S = \{(Rn, R, R), (R, R, R), (O, R, R), (R, Rn, R), (R, G, R),
(R, O, R), (R, R, Rn), (R, R, G), (R, R, O)\}\.
\]

State \(Rn\) denotes which traffic light turns green next. That way, we can distinguish the three states where all lights are red. Figure 7 depicts the transition relation.
Solution 1.4

This solution is just an example. There are many possible solutions to any given modeling exercise.

We can describe each state of the T-junction by a five-tuple \((1S, 1R, 2, 3S, 3L)\) where each of the five values denotes the respective traffic light of Figure 2. The basic observation is that the traffic lights 1S, 1R, and 3S can be green at the same time. The same holds for 1R and 2 together, and for 3L alone. The set of states can be described by

\[
\]

State Rn denotes which traffic light turns green next. That way, we can distinguish state \((R, R, R, R, Rn)\) from state \((Rn, Rn, R, Rn, R)\). The former encodes that in the next state traffic light 3L will turn green and not 1S, 1R, and 3S as in state \((Rn, Rn, R, Rn, R)\). Figure 8 depicts the transition relation.

Solution 1.5

This solution is just an example. There are many possible solutions to any given modeling exercise.

We can describe the state space by

\[
S = \{(0 - 0), (15 - 0), (30 - 0), (40 - 0), (30 - 15), (40 - 15), (40 - 30), (0 - 15), (0 - 30), (0 - 40), (15 - 30), (15 - 40), (30 - 40), (15 - 15), (30 - 30), (40 - 40), (A - 40), (40 - A), (1 - 0), (0 - 1)\}.
\]

State \((1 - 0)\) denotes that the first player has won the match, and state \((A - 40)\) denotes that the first player has an advantage. Figure 9 depicts the transition relation.
Solution 1.6

This solution is just an example. There are many possible solutions to any given modeling exercise.

1. Customers can be in eight states: Initially, they are about to enter the restaurant (initial). They then enter the restaurant (entered), are assigned to a table (table), get the menu (menu), order (order), receive the food (food), ask for the check (check), pay (payment), and finally leave the restaurant (initial). (We could also model the latter situation by using an explicit state left.) We have the following set of states:

\[ S = \{ \text{initial, entered, table, menu, order, food, check, payment} \} \]

Figure 10(a) depicts the transition relation. Being in state food, customers can return to state menu to order additional food.

A waiter can be in the following nine states: Initially, the waiter is waiting (initial). If customers enter the restaurant, the waiter assigns them a table (assignment), gives them a menu (give_menu), order (order), take the menu (take_menu), take the food (kitchen), serve the check (check), and pay (payment).

Figure 10: State-transition diagrams for the customer and the waiter.
the menu (give_menu), receives the order (order), takes the menu (take_menu), delivers the 
order to the kitchen (kitchen), delivers the food (food), delivers the check (check), receives 
the money (payment), and finally returns to state initial to wait for the next customers. As 
the latter is not specified, other modeling decisions are possible (e.g., the waiter remains in 
state payment). We can therefore describe the set of states by:

\[ S = \{ \text{initial, assignment, give_menu, order, take_menu, } \\ 
\text{kitchen, food, check, payment} \} \]

Figure 10(b) depicts the transition relation.

2. The state changes of the customer and the waiter are synchronized sequentially; that is, 
only one of the two can perform a state change at the same time. As a consequence, the 
state-transition diagram modeling the behavior of the customer and the waiter contains the 
sum of the two single state spaces: 8 + 9 = 17 states. If, in contrast, both actors could 
change state independently, there would be 8 \cdot 9 = 72 states.

Solution 1.7

\[
\text{This solution is just an example. There are many possible solutions to any } \\
\text{given modeling exercise.}
\]

The state-transition diagram is depicted in Figure 11. Note that purchasing software and hardware 
is done concurrently.

\[ \text{Figure 11: State-transition diagram of Exercise 1.7.} \]

Solution 1.8

\[
\text{This solution is just an example. There are many possible solutions to any } \\
\text{given modeling exercise.}
\]

The state-transition diagram is depicted in Figure 12. We have concurrency, because the roof can 
be made while the foundation and the frames are made and the walls are built. In addition, the 
frames can be made while the foundation is made.

\[ \text{Figure 12: State-transition diagram of Exercise 1.8.} \]
Solution 1.9
Figure 13 shows the state-transition diagram of the transition system of the washing machine.

![State-transition diagram for washing machine](image)

Figure 13: A state-transition diagram for the washing machine in Exercise 1.9.

Solution 1.10

This solution is just an example. There are many possible solutions to any given modeling exercise.

In the description of the washing machine, no distinction is made between the rinse after the pre-wash and the rinse after the main wash. The system can, therefore, move from state pre-wash to state rinse to state main_wash to state rinse, or stop directly after the first turn of rinse. We can solve this problem by replacing state rinse by two states, say, prewash-rinse (after the pre-wash) and main_wash-rinse (after the main wash). The state space changes to:

\[ S = \{ \text{off, defective, whiz, main_wash, pre-wash, prewash-rinse, main_wash-rinse}\} \]

Figure 14 depicts the transition relation \( TR \).

![Improved state-transition diagram](image)

Figure 14: Improved state-transition diagram for the washing machine.
Solution 1.11
1. Figure 15 depicts the state-transition diagram.

![State-transition diagram](image)

Figure 15: The state-transition diagram of Exercise 1.11.

2. The states reachable from state 0 are 0, 1, 2, 3, 4, and 5.

3. Assuming the initial state 0, examples of transition sequences are: (0, 1, 2, 3, 4, 4, 4, 4, 4), (0, 1, 2, 3, 5, 4, 4, 4, 4), (0, 1, 2, 3, 5, 0).

4. State 8 is a terminal state, because no arrows start here (state 8 is, however, not reachable). State 4 is not a terminal state, although it is not possible to “escape” from this state.

Solution 1.12
1. We formalize the state space as

\[ S = \{ (x_0, \ldots, x_4) \mid \forall i = 0, \ldots, 4 : x_i \in \{ t, e \} \land (x_i = e \implies x_{i-1 \mod 4} = x_{i+1 \mod 4} = t) \} \]

and a transition relation

\[ TS = \{ ((x_0, \ldots, x_4), (y_0, \ldots, y_4)) \in S \times S \mid \forall i = 0, \ldots, 4 : (x_i = t \land y_i = e) \implies x_{i-1 \mod 4} = x_{i+1 \mod 4} = t \} \]

A possible initial marking is (t, t, t, t, t); that is, all philosophers are thinking and all forks are on the table.

2. There are no deadlocks. In any state, there is at least one philosopher who can either pick up his pair of forks or put them down.

Solution 1.13
This solution is just an example. There are many possible solutions to any given modeling exercise.

When you have used your own TV as an example, your answers may be different from ours.

1. The first possible state is off. When the TV is turned on, we can describe a state by the triple \((\text{channel}, \text{mute}, \text{volume})\) where channel is a number between 1 and 6, mute has value on or off, and volume is represented by a number—for example, on a scale from 0 to 20. We must represent mute separately, because off is not the same as turning the volume down to 0. When we switch the sound on again, it starts with an initial volume setting. Other assumptions are possible. We can, for example, think of a TV that, when switched on again, returns to the state it was in when it was switched off the last time. This TV must “remember” its state when turned off. We can model that as a four-tuple \((\text{power}, \text{channel}, \text{mute}, \text{volume})\).

2. The TV is typically turned on by choosing a channel. The values of mute and volume then become standard values—for example, on and 8. We therefore obtain six transitions of the form \((\text{off}, n, \text{on}, 8)\). Pushing the volume button, when the TV is off, has no effect. So we have a transition \((\text{off}, \text{off})\). The volume button is the most interesting button. Pushing the volume control on the side labeled “+” increases the TV’s volume, except when the volume is already maximal. This gives the transitions: \(((\text{on}, n, \text{on}, x), (\text{on}, n, \text{on}, x+1))\), for \(x < 20\) and transition \(((\text{on}, n, \text{on}, 20), (\text{on}, n, \text{on}, 20))\). With the mute button, we may turn on the sound again. This results in transitions, such as \(((\text{on}, 3, \text{off}, 8), (\text{on}, 3, \text{on}, 8))\). As a consequence, changing the volume while the sound is muted is not possible.
2 Petri nets

A Petri net $N = (P, T, F)$ consists of a finite set $P$ of places, a finite set $T$ of transitions such that $P$ and $T$ are disjoint, and a flow relation $F \subseteq (P \times T) \cup (T \times P)$.

The set $t^* = \{ p \mid (t, p) \in F \}$ defines all output places of a transition $t$. We refer to $t^*$ as the preset of $t$ and to $t^*$ as the postset of $t$.

A marking of a Petri net $(P, T, F)$ is a function $m : P \rightarrow \mathbb{N}$, assigning to each place $p \in P$ the number $m(p)$ of tokens at this place.

A Petri net system $(P, T, F, m_0)$ consists of a Petri net $(P, T, F)$ and a distinguished marking $m_0$, the initial marking.

Exercise 2.1 Draw the Petri net defined by $P = \{p_1, p_2\}$, $T = \{t_1, t_2\}$, and $F = \{(p_1, t_1), (t_1, p_2), (p_2, t_1), (p_2, t_2), (t_2, p_2)\}$.

Exercise 2.2 Draw the Petri net system $(P, T, F, m_0)$, which is defined as $P = \{p_1, \ldots, p_7\}$, $T = \{t_1, \ldots, t_6\}$, $F = \{(p_1, t_1), (p_2, t_2), (p_3, t_3), (p_4, t_1), (p_4, t_4), (p_5, t_5), (p_6, t_6), (p_7, t_4), (t_1, p_2), (t_2, p_3), (t_3, p_1), (t_3, p_4), (t_4, p_5), (t_5, p_6), (t_6, p_4), (t_6, p_7)\}$, and $m_0 = [p_1, p_4, p_7]$.

Exercise 2.3 Consider the Petri net in Figure 16.

1. Define the net formally as a triple $(P, T, F)$.
2. List for each transition what its preset and postset are.

Exercise 2.4 Consider the Petri net in Figure 16 and assume a marking $m$ where every place contains exactly one token.

1. Which transitions are enabled at $m$?
2. What is the result of the firing of transition $t_1$ in $m$?
3. What is the result of the firing of transition $t_2$ in $m$?
4. What is the result of the firing of transition $t_3$ in $m$?
5. What are the reachable markings from $m$, and which of these markings are terminal markings?

Exercise 2.5 Consider the Petri net system in Figure 17.

1. Formalize this net as a four-tuple $(P, T, F, m_0)$.
2. Give the preset and the postset of each transition.
3. Which transitions are enabled at $m_0$?
4. Give all reachable markings.
5. What are the reachable terminal markings?
6. Is there a reachable marking where we have a nondeterministic choice?

7. Does the number of reachable markings increase or decrease if we remove (1) place $p_1$ and its adjacent arcs and (2) place $p_3$ and its adjacent arcs?

**Exercise 2.6** Consider the Petri net system in Figure 18.

1. Formalize this net as a four-tuple $(P, T, F, m_0)$.
2. Give the preset and the postset of each transition.
3. Which transitions are enabled at $m_0$?
4. Give all reachable markings.
5. What are the reachable terminal markings?
6. Is there a reachable marking where we have a nondeterministic choice?
7. Does the number of reachable markings increase or decrease if we remove place $p_1$ and its adjacent arcs?

**Exercise 2.7** The firing of a transition $t$ at marking $m$ yields a successor marking $m'$. Function $m'(p)$ in the definition of firing defines the effect of transition $t$ on a place $p$. It is any one of $m(p) + 1$, $m(p) - 1$, or $m(p)$. Explain when each of the three effects occurs and formalize this by completing the following equation:

$$m'(p) = \begin{cases} 
m(p) - 1, & \text{if } \ldots, \\
m(p) + 1, & \text{if } \ldots, \\
m(p), & \text{if } \ldots.
\end{cases}$$

**Exercise 2.8** Represent the Petri net system in Figure 19 as a transition system assuming that

1. Each place cannot contain more than one token in any marking;
2. Each place may contain any natural number of tokens in any marking.

Hint: Describe a marking of the net as a triple $(x, y, z)$ with $x$ specifies the number of tokens in place free, $y$ in place busy, and $z$ in place docu.
Exercise 2.9 A plant produces an item to order by a grinding step (event “g”), followed by a milling step (event “m”). After the milling step, the item is tested. If the test is positive (event “positive”), the item is sent to the customer (event “send”) and the production of the next item can start; otherwise (event “negative”), the item is discarded and the production starts again with a grinding step. Items can be in any of the states “wait”, “grinded”, “milled”, “to_be_sent”, “sent”, and “discarded”. Suppose that there are initially two items in state “wait”. Model this business process as a Petri net system.

Exercise 2.10 Model the following treatment of a patient at a dentist as a Petri net system. If we look at the patient separately, then there are five states: the patient is at home (state “p_home”), the patient is sitting in the waiting room (state “p_wait”), the patient is treated (state “p_treat”), the treatment is finished (state “p_done”), and the patient has left the practice (state “p_left”). There are four events: the patient enters the practice (“enter”); the treatment starts (“start”); the treatment is documented (“docu”); and the treatment ended (“end”). The dentist can be in three states: “d_free”, “d_busy”, and “d_docu”. There are three respective events: “start”, “docu”, and “end”. The nurse can be either in state “n_free” or in state “n_busy”. There are two events where the nurse is involved: “start” and “docu”. The nurse starts being busy after event “start” and becomes free again after event “docu”, namely when the patient is not treated anymore. Finally, there is also a secretary involved. Either she sits at the reception (state “s_reception”) or she helps the dentist documenting the treatment and writing a prescription (state “s_docu”). For the documentation, there are two events relevant: “docu” and “end”. Furthermore, when a patient enters the practice, the secretary is taking care of the reception. Suppose that initially there are two patients at home, the dentist and the nurse are in state “free”, and the secretary is sitting at the reception. Hint: Model first the patient and add then the dentist, the secretary, and the nurse.

Exercise 2.11 From a bridge, frogs jump into a stream, nondeterministically choose one of the two banks to swim to, and then hop to the bridge to start over again. A lovely girl picks up every third frog from the stream, kisses the frog, and puts the frog back on the bridge. Model this fairy tale as a Petri net. Suppose that three frogs are initially on the bridge.

Exercise 2.12 The Brisbane CityCat system is an urban transportation system using catamarans to quickly move people along the Brisbane River. Let us assume that there are four stops named A, B, C, and D. CityCats move from one stop to the other, first upstream (A,B,C,D) and then downstream (D,C,B,A). There are ten CityCats. Initially, all CityCats are in a dedicated harbor denoted by X. Depending on the workload, CityCats are put into service (moved from harbor X to stop A) or taken out of service (moved from stop A to harbor X). The number of CityCats in service may, therefore, vary between zero and ten. Possible moves of a CityCat are X,A,B,C,D,C,B,A,X; X,A,B,C,D,C,B,A,B,C,D, C,B,A,X; or X,A,B,C,D,C,B,A,B,C,D,C,B,A,D,C,B,A,X. A CityCat can only turn at stop D or A, depending whether it is respectively going towards D or, backwards, towards A; that is, a move X,A,B,A,X or X,A,B,C,B,A,X is not possible. The stops have a capacity of one; that is, only one CityCat can dock at a particular stop at a time. The capacity of the river is large enough to fit all CityCats in-between any two stops. Model the Brisbane CityCat system as a Petri net (including its initial marking). There is no need to distinguish individual CityCats.
Exercise 2.13 Every Petri net with place capacities can be transformed into a Petri net in which each place contains at most one token. The idea of the transformation is to model each place \( p \) with capacity \( k > 1 \) by \( k + 1 \) places \( p_0, \ldots, p_k \). A token in each of these places simulates a marking where place \( p \) in the original net contains the number of tokens according to the subscript. As an example, a token in place \( p_3 \) simulates three tokens in place \( p \). This transformation is known as unfolding. In addition to unfolding places with capacities, a transition having place \( p \) in its preset or postset must be unfolded. Unfold the Petri net system shown in Figure 20.

Figure 20: Place \( \text{in\_stock} \) has a capacity of two tokens.

Exercise 2.14 Consider the handling of insurance claims at Sunny Side Corp., Australia. Sunny Side distinguishes simple claims and complex claims. The type of the claim is determined in the first step.

For simple claims, Sunny Side carries out two steps independently: it checks the insurance policy of the insured party for validity and retrieves the statement of a local authority. When both results are available, in a next step Sunny Side checks the statement against the policy. If the result is positive, Sunny Side makes a payment to the insured party; if the result is negative, Sunny Side sends a rejection letter.

For complex claims, Sunny Side carries out three steps independently: it checks the insurance policy of the insured party for validity, retrieves the statement of a local authority, and asks for two witness statements. The business process can only proceed if both witness statements are available. Again, when both results are available, in a next step Sunny Side checks the statements of the local authority and witnesses against the policy. If the result is positive, Sunny Side makes a payment to the insured party; if the result is negative, Sunny Side sends a rejection letter.

1. Model this business process as a Petri net.

2. From time to time, Australia has to cope with large fires. As a result, Sunny Side is flooded with claims concerning the fires and the water damage from extinguishing the fires. If more than 150 claims are in the system of Sunny Side (i.e., claims that have been classified but not yet fully processed), the throughput decreases dramatically. Therefore, the management of Sunny Side decided, in such cases, to skip the extended procedure. Instead, only the check against validity of the insurance policy is made for new cases.

Draw a Petri net in which you clearly model the described situation; that is, in the case of more than 150 claims, the short procedure is followed and, in case of fewer cases, the initial procedure is followed. State the initial marking explicitly in your model. Show how the original model needs to be changed.

3. Suppose we want to change the business process of (1) as follows: If there are more than 150 claims in the system, then every new claim is determined to be simple; and if there are less than or equal to 150 claims in the system, a new claim is determined to be complex. This choice and hence the whole business process cannot be modeled as a Petri net.

Explain why it is not possible to model this choice by means of place capacities and arc multiplicities.

Exercise 2.15 Consider the following business process for handling traffic offenses. Every offense is registered after arrival. After registration, procedures “judge the traffic offense” and “investigate
the history” are started concurrently. In procedure “judge the traffic offense”, the traffic offense is classified as either “severe” or “normal”. Severe traffic offenses are then temporary judged, and, in a second step, a final judgment is delivered. Normal traffic offenses are judged in one step. Procedure “investigate the history” contains two steps that can be completed in arbitrary order: collect information about earlier traffic offenses and collect information about other offenses committed by the offender. The fine is determined after both procedures are finalized. If the traffic offense is not fined, it will be archived right away; otherwise, a transfer form is sent to the offender and subsequently the traffic offense will be archived.

1. Model this business process as a Petri net.

2. To avoid long processing times, the business process is changed as follows: If more than 100 traffic offenses are processed at the same time (i.e., registered offenses that are not archived yet), every new traffic offense will be processed by an alternative procedure. This procedure contains only one step: the offense is registered and archived. If fewer than 100 offenses are processed at the same time, the original procedure will be followed. Show how the original model needs to be changed.

Exercise 2.16 A factory produces one type of bicycles. The parts (frame, pedal, wheel, and brake) are purchased from various suppliers. The factory needs three assemblies steps to make a bicycle. First, a machine of type B assembles a frame and two pedals into semi-product_1. In the second assembly step, a machine of type A assembles semi-product_1 and two wheels into semi-product_2. In assembly step three, a machine of type B mounts two brakes (front and back) to semi-product_2. After this third assembly step, the bicycle is ready. Currently, the factory has three type-A machines and seven type-B machines available. Every machine has a capacity of one.

1. Model this business process as a Petri net system. Distinguish between the two types of machines and whether the machines are available or not. Assume that the company has initially four copies of each part.

2. Give a run from the initial marking to a marking where a bicycle has been produced.

Exercise 2.17 Consider the business process of a language institute that offers three language courses: English, German, and French. Students need to register before they can join a course. Registered students can choose one of the three courses. Every course has a specific duration, and the number of participating students is limited. At most 20 students can join the English course; the other courses have a capacity of 10 persons each. If a course is open, students may choose the course. After a course has been started, students who have not yet chosen this course cannot take part any more. Students are not allowed to drop a course once they started. After the course has been finished, a student can decide either to follow another course (without registering again and with the possibility to do the same course again) or to deregister and to leave. To give other students a chance to take part in a language course, students are not allowed to remain in a course. Consequently, if in the new semester a course is opened, the number of participants is zero initially.

1. Model this business process as a Petri net. (Hint: Model the states of a student first).

2. How can we remodel the Petri net in (1) such that at most 25 students are registered at the same time?

3. The language institute wants for efficiency reasons to rotate the German and the French course. How must we adjust the Petri net in (1)? (Suppose that initially the German course is offered.)

Exercise 2.18 Construct a Petri net system that has the reachability graph in Figure 21.
Exercise 2.19 In this exercise, we consider a simplified transportation system; see Figure 22.

From a harbor, trucks transport coal to a coal-fired power plant. More precisely, the trucks drive to an unloading point where they unload their coal and then drive back. The road has one lane from the harbor to the unloading point and another lane from the unloading point to the harbor. There is, however, a small part of the road where - because of reconstruction work - only one truck can drive (i.e., this part has a capacity of one truck). While a truck is passing this part, all other trucks - no matter which direction they go - have to wait. Initially, there are five trucks at the harbor. Each truck has a capacity of ten units of coal. The unloading point has a capacity of 100 units of coal. As long as this capacity is not reached, arriving trucks can immediately unload and drive back. If the capacity has been reached or unloading would exceed the capacity, the truck has to wait. There is no partial unload of trucks!

At the unloading point, there are two wheel loaders, transporting coal from the unloading point to the power plant. Each wheel loader has a capacity of one unit of coal.

Model this transportation system as a classical Petri net (i.e., a Petri net with indistinguishable black tokens without time). The model must clearly reflect the capacities and that each truck always drives from the harbor to the unloading point, and vice versa.

Exercise 2.20 Let us consider a simple production system (see Figure 23 for an illustration) where raw parts are preprocessed by a machine $M_1$, stored in a temporary buffer, and finally assembled by a second machine $M_2$. There is a single robot $R$ that moves the parts between the input line, $M_1$, the buffer, $M_2$ and the output line. The buffer can hold at most seven preprocessed items.

Model the production system as a classical Petri net. Include in this classical Petri net model
a simple environment that is producing raw parts and consuming finished parts. There is no need to distinguish particular buffer places and parts. The actions by the two machines are not atomic; that is, the start and the completion of these actions should be distinguishable. The actions of the robot may be considered to be atomic.

**Exercise 2.21** Let us consider a small airport with two runways. A runway is a strip of land on an airport, on which aircrafts can take off and land. Aircrafts that enter the airspace of the airport need to wait until they get permission to land. Similarly, aircrafts need permission to take-off. At any point in time only one aircraft can use a runway but both runways can be used at the same time. The total number of aircrafts the airport can hold is ten, so an aircraft cannot land if there are already ten aircrafts at the airport (including runways). After an aircraft has landed, it unloads its passengers, loads new passengers, and then is ready for take-off.

- Model the airport as a classical Petri net. Include in this classical Petri net model a simple environment that models the arrival of aircrafts into the airspace of the airport and that models the disappearance of departing aircrafts.
- Extend the previous model to include the constraint that it is not allowed to use one runway for landing and the other for take-off at the same time. So both runways may be used for landing, or both runways may be used take-off, or only one of them is used. Model the new situation as a classical Petri net.

**Exercise 2.22** Let us consider a tunnel connecting Limburg to the rest of the Netherlands. This (fictive) tunnel can only be used in one direction at a time (i.e., from Limburg to the rest of the Netherlands or from the rest of the Netherlands to Limburg); see Figure 24.

![Figure 24: Illustration of the tunnel.](image)

Depending on the traffic, the control-system of the tunnel can switch direction (of course only when the tunnel is empty). The tunnel cannot hold more than ten cars at a time; that is, the capacity is limited and not more than 10 cars are allowed to be in the tunnel at the same time. There should be no unnecessary waiting; for example, it cannot be the case that the tunnel remains empty while cars are queuing.

Model the tunnel and its environment as a marked classical Petri net by extending the partial model shown above. Note that cars should flow from \( p2 \) to \( p3 \) and from \( p6 \) to \( p7 \). The four transitions model the environment that generates cars and absorbs cars. Make sure that the model respects safety (no cars driving in opposite directions and not more than ten cars in the tunnel) without unnecessary waiting. However, the model does not need to be “fair”.

**Exercise 2.23** Consider a take-away bar that sells three fixed menus. Menu \( A \) consists of three beers and two servings of chips, menu \( B \) consists of two wines and one serving of chips, and menu \( C \) consists of two beers, two wines and five servings of chips. Customers arrive in an unpredictable manner; that is, sometimes several customers arrive in a short-time frame while at other times no customers visit the take-away bar for an extended period. An arriving customer tells one of the ten sales persons of the bar which kind of menu is desired (one per customer). Besides the
ten sales persons there are three employees that can prepare drinks. These can prepare the beers and wines ordered as part of the menu. Similarly, there are two dedicated employees to prepare food (i.e., servings of chips). Preparing drinks and food takes time and each of the employees can only work on only one item at a time; that is, at most three drinks and two servings of chips can be prepared in parallel. After all items have been prepared, they are handed over the customer who subsequently leaves to consume the selected menu at home. While a menu is prepared for a customer, the sales person that accepted the order cannot accept any new orders. Hence, at most ten customers can be served at the same time. The system does not need to be “fair”; that is, one customer order can overtake another one and a beer prepared for one customer can be used for another. The only requirement is that there is no unnecessary waiting and that only the ordered items are being prepared.

Model the take-away bar in terms of a classical Petri net. The modeled systems should be live and reversible. Clearly show the model and also include its initial state.

Exercise 2.24 Consider the process of shipping containers from one side of the canal to the other. There are two container terminals (A and B) and four container ships moving containers from Terminal A to Terminal B. Terminal A has two cranes, and Terminal B has only one crane. The cranes at Terminal A are used to load containers onto ships whereas the crane at Terminal B is used to unload containers. There are two types of containers: 20ft (20 foot long) and 40ft (40 foot long). A truck can transport one 40ft container or two 20 ft containers.

![Figure 25: Illustration of the shipment process.](image)

Figure 25 shows the interface places of the process to be modeled. Trucks containing one 40ft, or one or two 20ft containers arrive on the left, leave their container(s) at Terminal A and leave. The two cranes of Terminal A can put these containers onto one of the four container ships. Loading a ship takes time and during this period one of the cranes is dedicated to the ship. Hence, not more than two ships can be loaded at the same time. Each ship has a capacity of 80ft; that is, a ship may hold two 40ft containers, four 20ft containers, or one 40ft container and two 20ft containers. A ship can only leave once it is fully loaded (80ft of containers). The crane at Terminal B is dedicated to one container ship at a time; that is, the crane is unloading the containers for some time until it can serve the next ship that has arrived. After the ship has been unloaded at Terminal B, it returns to Terminal A on the other side of the canal. This also takes time. We assume that no containers need to be shipped from Terminal B to Terminal A. The containers that have been unloaded are picked up by empty trucks visiting Terminal B (see places on the right-hand side). Again trucks can hold one 40ft container, or one or two 20ft containers. Note that two 20ft containers loaded onto the same truck when arriving at Terminal A may leave Terminal B separately, and vice versa.

Model the above process in terms of a classical Petri net. The model should be readable and clearly show the initial marking. Also list design decisions and assumptions which are not obvious from the description above.
Solutions

Solution 2.1

This solution is just an example. There are many possible solutions to any given modeling exercise.

Figure 26 depicts the Petri net.

Solution 2.2

This solution is just an example. There are many possible solutions to any given modeling exercise.

Figure 27 depicts the Petri net.

Solution 2.3

1. We obtain $P = \{p1, p2\}$, $T = \{t1, t2, t3\}$, and $F = \{(p1, t1), (t1, p2), (p2, t1), (p2, t3), (p2, t2), (t2, p1), (p1, t2)\}$.

2. We obtain the following presets and postsets: $t1^* = \{p1, p2\}$, $t1^* = \{p2\}$, $t2^* = \{p1\}$, $t3^* = \{p2\}$, $t3^* = \emptyset$.

Solution 2.4

1. All three transitions are enabled. If more than one transition is enabled, it is not clear which of these transitions will fire. A nondeterministic choice needs to be made.

2. If transition $t1$ fires, then place $p1$ is empty and place $p2$ still contains one token.

3. If transition $t2$ fires, then place $p2$ is empty and place $p1$ contains one token.

4. If transition $t3$ fires, then place $p2$ is empty and place $p1$ contains one token.
5. The following markings are reachable: \([p1, p2]\) (i.e., the initial marking), \([p2]\) (i.e., the marking after the firing of transition \(t1\)), \([p1]\) (i.e., the marking after the firing of transition \(t2\)), and \([p1, p2]\) (i.e., the marking after the firing of transition \(t3\) in the initial marking \([p1, p2]\)). Markings \([p1]\) and \([p1, p2]\) are the reachable terminal markings.

**Solution 2.5**

1. We obtain \(P = \{p1, p2, p3, p4\}\), \(T = \{t1, t2, t3\}\), \(F = \{(p1, t1), (t1, p1), (t1, p2), (t2, p2), (t2, p3), (t3, p3), (t3, p4), (p4, t4), (p4, t1)\}\), and \(m0 = [p1, p3, 2 \cdot p4]\).

2. We obtain \(t1 = \{p1, p4\}\), \(t1^* = \{p2\}\), \(t2 = \{p2\}\), \(t2^* = \{p3, p4\}\), \(t3 = \{p3, p4\}\), \(t3^* = \{p4\}\).

3. Transitions \(t1\) and \(t3\) are enabled at \(m0\), because each of their respective input places contains at least one token. Transition \(t2\) is not enabled at \(m0\), because its input place \(p2\) does not contain a token.

4. The initial marking \(m0 = [p1, p3, 2 \cdot p4]\). The firing of transition \(t3\) yields marking \([p1, 2 \cdot p4]\). Afterward, transitions \(t1\), \(t2\), and \(t3\) can subsequently fire, yielding markings \([p2, p4]\), \([p3, 2 \cdot p4]\), and \([2 \cdot p4]\). If transition \(t1\) fires in \(m0\), we reach marking \([p2, p3, p4]\). In this marking, transition \(t3\) can fire, yielding marking \([p2, p4]\) (the reachable markings of this marking have been already given). Otherwise, if transition \(t2\) fires in marking \([p2, p3, p4]\), the net reaches marking \([2 \cdot p3, 2 \cdot p4]\) and by firing transition \(t3\) twice, we reach markings \([p3, 2 \cdot p4]\) and \([2 \cdot p4]\).

5. The only reachable terminal marking is \([2 \cdot p4]\).

6. At marking \([p1, p3, 2 \cdot p4]\), transitions \(t1\) and \(t3\) are enabled; and at marking \([p2, p3, p4]\), transitions \(t2\) and \(t3\) are enabled.

7. If we remove place \(p1\), then we can have an infinite firing sequence: \(\langle t1, t2, t1, \ldots \rangle\). As each firing of transition \(t2\) produces a token in place \(p3\), we can reach infinitely many different markings, meaning, the number of markings increases. In contrast, removing place \(p3\) reduces the number of reachable markings. The intuition is that the firing of transition \(t3\) does not change the marking, whereas this was the case in the presence of place \(p3\).

**Solution 2.6**

1. We obtain \(P = \{p1, p2, p3\}\), \(T = \{t1, t2, t3, t4\}\), \(F = \{(p1, t2), (t2, p2), (p2, t1), (t1, p1), (p3, t3), (t3, p3), (t3, p4), (t4, p2), (p2, t4)\}\), and \(m0 = [p2, p3]\).

2. We obtain \(t1 = \{p2\}\), \(t1^* = \{p1\}\), \(t2 = \{p2\}\), \(t2^* = \{p1\}\), \(t3 = \{p2\}\), \(t3^* = \{p3\}\), \(t4 = \{p2, p3\}\), \(t4^* = \{p2\}\).

3. Transitions \(t1\), \(t3\), and \(t4\) are enabled at \(m0\), because each of their respective input places contains one token. In contrast, transition \(t2\) is not enabled at \(m0\), because its input place \(p1\) does not contain a token.

4. The initial marking is \([p2, p3]\). The firing of transition \(t1\) yields marking \([p1, p3]\). By firing transition \(t2\), \(m0\) is reached again. The firing of transition \(t3\) in \(m0\) yields marking \([2 \cdot p3]\). The firing of transition \(t4\) in \(m0\) yields marking \([p2]\). If we continue with the firing of transition \(t3\), we reach marking \([p3]\). Firing transitions \(t1\) and \(t2\) subsequently in marking \([p2]\) yields marking \([p4]\) and then marking \([p2]\) again.

5. There are two reachable terminal markings: \([p3]\) and \([2 \cdot p3]\).

6. At marking \([p2, p3]\), transitions \(t1\), \(t3\), and \(t4\) are enabled; and at marking \([p2]\), transitions \(t1\) and \(t3\) are enabled.
7. If we remove place $p_1$, the postset of transition $t_2$ is the empty set; that is, transition $t_2$ has no input places. As a consequence, transition $t_2$ is now in any reachable marking enabled (Note that the definition of an enabled transition “all input place of transition $t_2$ contain at least one token” is equivalent to “there is no input place of transition $t_2$ that contains less than one token”). So transition $t_2$ can subsequently fire infinitely often, thus producing an infinite number of tokens in place $p_2$. Accordingly, infinitely many markings are reachable in the modified net.

**Solution 2.7**

If place $p$ is an input place of transition $t$ but no output place of $t$, then $m'(p) = m(p) - 1$. If place $p$ is an output place of transition $t$ but no input place of $t$, then $m'(p) = m(p) + 1$. The third case is less obvious. If place $p$ is either no input and also no output place of transition $t$ or input and output place of $t$, then we have $m'(p) = m(p)$. Clearly, if $p$ is neither input nor output place of $t$, the number of tokens keep unchanged. Likewise, the number of tokens does not change if $t$ produces a token in place $p$ and consumes a token from $p$. Formally, we have

$$m'(p) = \begin{cases} 
  m(p) - 1, & \text{if } p \in t^{\bullet} \setminus t^{•}, \\
  m(p) + 1, & \text{if } p \in t^{•} \setminus t^{\bullet}, \\
  m(p), & \text{if } (p \in t^{•} \land p \in t^{•}) \lor (p \notin t^{\bullet} \land p \notin t^{•}).
\end{cases}$$

As the third condition is fulfilled only if the first and second condition are not fulfilled, we can simplify the formalization as follows:

$$m'(p) = \begin{cases} 
  m(p) - 1, & \text{if } p \in t^{•} \setminus t^{•}, \\
  m(p) + 1, & \text{if } p \in t^{•} \setminus t^{•}, \\
  m(p), & \text{otherwise}.
\end{cases}$$

**Solution 2.8**

The transition system $(S, TR, s_0)$ can be specified as follows. The set $S$ of states is the set $M$ of all markings of the Petri net system in Figure 19. The initial state is equal to the initial marking, and the transition relation $TR$ contains all transitions that can occur from any of the states in $S$.

1. We obtain $S = \{(0,0,0),(1,0,0),(0,1,0),(0,0,1),(1,1,0),(1,0,1), (0,1,1), (1,1,1)\}$, $s_0 = (1,0,0)$, and $TR = \{((1,0,0),(0,1,0)), ((0,1,0),(0,0,1))$, $((0,0,1),(1,0,0))$, $((1,1,0),(1,0,1)), ((0,1,1),(0,1,1)), ((0,1,1),(1,1,0))\}$. Note that with the restriction of the number of tokens in a place to at most one, transitions like $((1,1,1),(0,2,1))$ are not possible.

2. We can formalize the states as $S = \{(x,y,z) \mid x,y,z \in \mathbb{N}\}$. The initial state $s_0$ is equal to the initial marking $m_0 = (1,0,0)$. The transition relation can be specified by the union of the following three sets:

$$TR = \left\{ (x+1, y, z), (x, y+1, z) \mid x, y, z \in \mathbb{N} \right\} \cup \left\{ (x+1, y, z), (x, y, z+1) \mid x, y, z \in \mathbb{N} \right\} \cup \left\{ (x, y, z+1), (x+1, y, z) \mid x, y, z \in \mathbb{N} \right\}.$$ 

The first, the second, and the third set contains all possible states that can be reached by firing transition $start$, $change$, and $end$, respectively.

**Solution 2.9**

*This solution is just an example. There are many possible solutions to any given modeling exercise.*

We model each event by a transition and each state by a place. Figure 28 depicts the corresponding Petri net system. The production of the next item starts only after the previous item has been sent (event “send”) or discarded (event “negative”). To avoid that transition $g$ is enabled before one of these transitions has fired, we modeled this state of the production process as a place free. The second state of the production process, “busy”, does not have to explicitly modeled, because it is incorporated in the state of the product.
Solution 2.10

This solution is just an example. There are many possible solutions to any given modeling exercise.

We model each event as a transition and each state as a place. Figure 29 depicts the corresponding Petri net system. The modeling is straightforward. Observe that we model the step when a patient enters the practice and registers at the reception by consuming the token from place $s_{\text{reception}}$ and by producing the token in place $s_{\text{reception}}$, because the secretary does not change her state.

Solution 2.11

This solution is just an example. There are many possible solutions to any given modeling exercise.

Figure 30 depicts the model of the fairy-tale. Let us first separately consider the states a frog can be in. A frog can be on the “bridge”, in the “stream”, on “bank1”, or on “bank2”. We model each of these states as a place. There are five events: the frog jumps from the bridge (“jump”), swims to one of the banks (“swim1” and “swim2”), or hops from the respective bank back on the bridge (“hop1” and “hop2”). We model each event as a transition. To model the girl, we need a place and a token in this place (state “girl”) and an event “kiss” modeled as a transition $\text{kiss}$. To ensure that the girl picks up every third frog, we added transition $\text{jump3}$ and place $\text{stream3}$. After transition $\text{jump}$ fired twice, transition $\text{jump3}$ is enabled. This is guaranteed by places $\text{counter1}$ and $\text{counter2}$. When the girl kisses the frog and puts it back on the bridge (i.e., transition $\text{kiss}$ fires), then two tokens are produced in place $\text{counter2}$ and the next two frogs can jump using transition $\text{jump}$.
Figure 30: The Petri net of Exercise 2.11.

Figure 31 depicts another model of the fairy-tale. In contrast to Figure 30, in which every third frog is kissed in every run, the model in Figure 31 is less restrictive and only guarantees that every third frog is kissed in every infinite run, thereby assuming that the three transitions in the postset of place $bridge$ fire with equal probability.

Figure 31: Another Petri net of Exercise 2.11.

Solution 2.12

This solution is just an example. There are many possible solutions to any given modeling exercise.

Figure 32 depicts the Petri net. A CityCat can be in the harbor $X$; in one of the four stops $A$, $B$, $C$, or $D$; or between stops $A$ and $B$, $B$ and $C$, or $C$ and $D$. We model each state as a place. To distinguish whether a CityCat is moving upstream or downstream, each state between two stops is either upstream or downstream (e.g., “AtoB” and “BtoA”). The same distinction holds for stop $B$ and stop $C$ where a CityCat cannot turn (e.g., we have states “B_up” and “B_down”). The transitions model the flow of the CityCats. With places $A_{free}$, $B_{free}$, $C_{free}$, and $D_{free}$, we guarantee that only one CityCat can dock at a stop at a time. Initially, these places hold a token each; place $X$ contains initially 10 tokens, each representing a CityCat.
Solution 2.13

This solution is just an example. There are many possible solutions to any given modeling exercise.

Figure 33 depicts the unfolded Petri net system. As place free has a capacity of two, the unfolding has three places: free_0, free_1, and free_2. A token in place free_0 simulates zero tokens in place free, a token in free_1 simulates one token in place free, and a token in place free_2 simulates two tokens in place free. Likewise, the unfolding has places in_stock_0, in_stock_1, and in_stock_2. In the net with place capacities, transition deliver is enabled if there is either one token in place free (and hence one token in place in_stock) or two tokens in place free (and hence no token in place in_stock). Therefore, we need to unfold transition deliver into transitions deliver_12 and deliver_01. Likewise, we unfold transition take into transitions take_12 and take_01. The former transition simulates that the number of tokens in place free increments from one to two and the latter transition simulates that the number of tokens in place free increments from zero to one.

This construction shows that place capacities do not increase the expressiveness of Petri nets, because there always exists an unfolding that simulates the net with place capacities. Instead, the use of place capacities results in a more compact and readable process model.
Solution 2.14

This solution is just an example. There are many possible solutions to any given modeling exercise.

1. Figure 34 depicts the Petri net modeling this business process. Place \( p0 \) collects the incoming claims. We model the decision whether a claim is simple or complex by a nondeterministic choice (transitions \( \text{simple} \) and \( \text{complex} \)). For simple claims, checking the insurance policy of the insured party for validity (transition \( \text{s\_check} \)) and retrieving the statement of a local authority (transition \( \text{s\_retrieve} \)) is performed concurrently. The result is checked against the policy. As this check may have two outcomes, positive and negative, two transitions (\( \text{s\_check\_p} \) and \( \text{s\_check\_n} \)) model the decision by a nondeterministic choice. In the complex case, three events are executed concurrently: checking the insurance policy of the insured party for validity (transition \( \text{c\_check} \)), retrieving the statement of a local authority (transition \( \text{c\_retrieve} \)), and asking for two witness statements (transition \( \text{ask\_w} \)). The two witness statements can be received concurrently (transitions \( w1 \) and \( w2 \)). Afterward, the results are checked against the policy. Again, this is modeled by a nondeterministic choice (transitions \( \text{c\_check\_p} \) and \( \text{c\_check\_n} \)). Finally, transitions \( \text{pay} \) and \( \text{send\_letter} \) model the payment (if the previous check was positive) and sending the letter of rejection (if the previous check was negative). We join the two alternative branches, because after the check against the policy both kinds of claims are processed the same way.

![Figure 34: The Petri net of Exercise 2.14(1).](image)

2. To model the new procedure, we add transition \( \text{validity\_check} \). The firing of this transition takes a claim from place \( p0 \) and puts it in place \( p14 \) from which the payment can be arranged. Transition \( \text{validity\_check} \) should be only enabled if the system is processing more than 150 claims. To model this, we make use of place capacities and introduce places \( \text{claims} \) and
max. Place claims counts the number of claims that are in processing. It is an output place of transitions simple and complex and an input place of transitions s_check_p, s_check_n, c_check_p, and c_check_n. Place max ensures that place claims has a capacity of at most 150 tokens. To this end, place max contains in the initial marking 150 tokens. That way, transitions simple and complex are only enabled if place max contains at least one token; that is, the number of claims in processing has not exceeded 150 yet. To guarantee that transition validity_check is enabled only if the number of claims in processing exceeded 150, this transition has to consume 150 tokens from place claims and to produce the same number of tokens in this place. As the sum of tokens in places claims and max is equal to 150, place max contains only 150 tokens (and thus enables transition validity_check) if the system is processing 150 claims. Figure 35 illustrates the construction.

![Figure 35: The Petri net of Exercise 2.14(2).](image-url)

3. Modeling that transition simple is only enabled if more than 150 claims are in the pipeline is easy. We add a place claims (see Figure 35) that counts the number of claims. Transition simple must then consume 151 tokens from place claims and produce the same number of tokens in place claims. Now we need to ensure that transition complex is only enabled if place claims contains less than or equal to 150 tokens. As we do not know the maximal number of claims being processed at the same time, we do not have a place capacity of place claims. Theoretically, place claims may contain infinitely many tokens. Consequently, we cannot unfold this place, as this would result in an infinite number of places violating the requirement stated in the definition of a classical Petri net. Hence, this choice cannot be modeled as a Petri net (using place capacities and arc multiplicities). Note that we can model the business process if we assume some upper bound for the number of customers.
**Solution 2.15**

_This solution is just an example. There are many possible solutions to any given modeling exercise._

1. Figure 36 depicts the business process model. Place $p_0$ collects the arrived traffic offenses. The offenses are then registered (transition $\text{register}$). After registration, two procedures are performed concurrently. The first branch models the judgment of the traffic offense. A nondeterministic choice of transitions $\text{normal}$ and $\text{severe}$ models the decision whether the traffic offense is “normal” or “severe”. If the traffic offense is normal, there is only a single judgment step (transition $\text{judge}$); otherwise, there are a temporary and a final judgment step (transitions $t\text{\_judge}$ and $f\text{\_judge}$). The second branch investigates the history (transition $\text{investigate\_history}$). Information about earlier traffic offenses (transition $\text{earlier\_offenses}$) and about other offenses committed by the offender (transition $\text{other\_offenses}$) are collected. Then the fine is determined (transition $\text{determine}$). We model the evaluation of the fine again as a nondeterministic choice (transitions $\text{no\_fine}$ and $\text{fine}$). In the case of a fine, first a letter is sent (transition $\text{send}$), and then the traffic offense is archived (transition $\text{archive}$). If no fine is determined, the traffic offense is immediately archived, and a token is produced in the input place of transition $\text{archive}$.

![Figure 36: The Petri net of Exercise 2.15(1).](image)

![Figure 37: The Petri net of Exercise 2.15(2).](image)

2. Like in Exercise 2.14(2), we add two places $\text{offenses}$ and $\text{max}$ to the Petri net in Figure 36. Place $\text{offenses}$ is an output place of transition $\text{register}$ and an input place of transition $\text{archive}$. 
archive. It counts the number of traffic offenses being processed at the same time. Place max ensures that offenses contains at most 100 tokens. To this end, it is an input place of transition register and an output place of transition archive. That way, transition register is only enabled if the number of traffic offenses in processing is less than 100; that is, if there is at least one token in place max. To model the new procedure, we add a transition quick_check. Place p0 is an input place and place p14 is an output place of this transition. In addition, this transition consumes 100 tokens from place offenses and produces 100 tokens in place offenses. By the construction of places max and offenses, transition quick_check is only enabled if there are 100 traffic offenses being processed. Figure 37 sketches this construction.

Solution 2.16

This solution is just an example. There are many possible solutions to any given modeling exercise.

1. Figure 38 depicts the Petri net modeling the business process. Each assembly step consists of two events, “start” and “end”; each machine is either in state “busy” or in state “free”. To start an assembly step, the corresponding machine must be free and the assemblies must be available. If the assembly step ends, then the involved machine changes to state “free” and the corresponding product is produced. A token in place sp1 and sp2 represents an assembled semi-product 1 and semi-product 2, respectively.

2. The run ⟨start_step1, end_step1, start_step2, end_step2, start_step3, end_step3⟩ yields a marking where place bike contains a token. Note that several places still contain unused parts afterward.

Figure 38: The Petri net of Exercise 2.16.
Solution 2.17

This solution is just an example. There are many possible solutions to any
given modeling exercise.

1. From the description, we identify two objects that we need to model: the “life cycle” of a student in the language institute and the state of a course. Let us first consider the “life cycle” of a student in the language institute. Students can be in any of the following states: (1) not yet registered at the language institute (state “free”); (2) registered (state “registered”); (3) participant of any of the three courses English, German, and French (states “E_student”, “G_student”, and “F_student”); (4) left a course (state “left”); and (5) deregistered (state “deregistered”). A student can register at the language institute (event “register”), choose one of the three courses (events “choose_E”, “choose_G”, and “choose_F”), leave the respective course (events “leave_E”, “leave_G”, and “leave_F”), choose a new course (event “new_choice”), and deregister (event “deregister”). We model each state as a place and each event as a transition. A token models a student being in the respective state. As students can choose between the three courses, we model this by a nondeterministic choice between transitions choose_E, choose_G, and choose_F. Place left collects all students who left a course.

According to the specification, each course has a maximum number of participants. For the English course, for example, we have to ensure that place E_student holds at most 20 tokens; that is, at most 20 students may choose the English course. We achieve this by adding a place maxE, which is initially marked with 20 tokens. Each transition that produces a token in place E_student (i.e., transition choose_E) consumes a token from place maxE. Likewise, each transition that consumes a token from place E_student (i.e., transition leave_E) produces a token in place maxE. That way, the sum of tokens in the two places E_student and maxE is equal to 20 at any marking. We do the same construction for the two other courses.

Having modeled the states of a student, it remains to model the states of a course. A course can be in three states: it is open (state “open”), it is started (state “lesson”), and it is finished (state “holiday”). To switch between these three states, we need three events: the course starts (event “start”); it finishes (event “finish”); and it opens again (event “empty”). With three places and three transitions, we can model the “life cycle” of a course. Figure 39 depicts the complete model of the language institute.

According to the specification of the business process, students may only choose a course as long as it has not been started. For this reason, transition choose_E checks whether the course is still in state “E_open” (i.e., there must be a token in place E_open). The same construction is used for the German and the French course. Another requirement is that students who participate at a course cannot remain in that course after it has finished. In Figure 39, we guarantee that transition empty_E is not enabled until all students have left the course. This is the case if place E_student does not contain any token and consequently place maxE contains 20 tokens. Adding place E_holiday to the input and output places of transition leave_E guarantees that students cannot drop a chosen course before it ends. Again, the same construction has been used for the German and French course.

2. In our model this means that the sum of tokens in places registered, EParticipant, GParticipant, FParticipant, and left must be less than or equal to 25. We can achieve this easily by adding a place maxR (maximum registrations) that contains initially 25 tokens. Place maxR has to be an input place of transition register and an output place of transition deregister. That way, for each newly registered student, we consume a token from maxR; and for each deregistered student, we produce a token in place maxR. This construction guarantees the required capacity. Figure 40 shows the extension.

3. Because of this requirement, we have to make sure that the institute offers first the German course and after this course has finished, it offers the French course, and so on. To this end, we have to remove arc (empty_G, G_open), because after the German course is
finished and all students have left, the French course has to be opened. In addition, we add an arc \((\text{empty}_G, F_{\text{open}})\). Likewise, we remove arc \((\text{empty}_F, F_{\text{open}})\) and add an arc \((\text{empty}_F, G_{\text{open}})\). Finally, we need to remove the token from place \(F_{\text{open}}\). Figure 40 shows the extension.

Figure 39: The Petri net modeling the whole language institute.
Figure 40: The Petri net modeling the extended language institute.
Solution 2.18

This solution is just an example. There are many possible solutions to any given modeling exercise.

Figure 41 is an example of a solution. Each transition in the reachability graph corresponds to a transition in Figure 41. The reachability graph also illustrates that transitions $c$ and $d$ can happen in any order. Hence, these transitions can occur concurrently in Figure 41.

![Figure 41: The Petri net of Exercise 2.18.]

The reachability graph in Figure 21 illustrates that concurrency cannot be explicitly modeled as a transition system. In contrast, the Petri net in Figure 41 explicitly shows that transitions $c$ and $d$ are concurrent.

Solution 2.19

This solution is just an example. There are many possible solutions to any given modeling exercise.

Figure 42 depicts the Petri net.

![Figure 42: The Petri net of Exercise 2.19.]

32
Solution 2.20

This solution is just an example. There are many possible solutions to any given modeling exercise.

Figure 43 depicts the Petri net.

Figure 43: The Petri net of Exercise 2.20.
Solution 2.21

This solution is just an example. There are many possible solutions to any given modeling exercise.

1. Figure 44 depicts the Petri net.

![Figure 44: The Petri net of Exercise 2.21(1).](image)

2. Figure 45 depicts the Petri net.

![Figure 45: The Petri net of Exercise 2.21(2).](image)
Solution 2.22

This solution is just an example. There are many possible solutions to any given modeling exercise.

Figure 46 depicts the Petri net.

![Petri net diagram](image)

Figure 46: The Petri net of Exercise 2.22.

Figure 47 depicts an alternative solution to avoid that the switch transitions fire repeatedly.

![Petri net diagram](image)

Figure 47: The Petri net of Exercise 2.22.
Solution 2.23

\textit{This solution is just an example. There are many possible solutions to any given modeling exercise.}

Figure 48 depicts the Petri net. Note that one can add a self-loop place to the source and sink transition to avoid having transitions without input or output.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure48}
\caption{The Petri net of Exercise 2.23.}
\end{figure}

Solution 2.24

\textit{This solution is just an example. There are many possible solutions to any given modeling exercise.}

Figure 49 depicts the Petri net. Note that one can add a self-loop place to the source and sink transition to avoid having transitions without input or output.
Figure 49: The Petri net of Exercise 2.24.
3 Colored Petri nets

Exercise 3.1 Consider a product quality check within a manufacturing line. Suppose that three products A, B, and C are produced. Initially, these products are in a store. There are two robots that move a product from the store to a scanner. The scanner recognizes whether the quality of the product is low or high. Low-quality products are put into one container, high-quality products into another one. Suppose that moving a product takes two minutes and scanning a product takes five minutes.

1. Model this process as an untimed CPN.
2. Extend the CPN model in (1) with time information.

Exercise 3.2 Parking at XY airport follows some rules: Arriving cars can choose between two car parks A and B. Car parks A and B have a capacity of 50 and 100 cars, respectively. To enter a car park, the car has to pass a barrier. The barrier opens only if there is at least one available parking spot. A light shows whether this is the case. Passing the barrier takes one minute. If no parking spot is available, cars have to drive on. Cars entering car park A need one minute to find a parking spot and leave the car park after ten minutes. Because of the size of car park B, it is more difficult to find a parking spot. If more than two third of B’s capacity is reached, drivers need five minutes to find a parking spot; otherwise, it takes two minutes. Cars parking at car park B will leave after some time defined by a function $f()$.

1. Model the process of parking as a timed CPN.
2. Extend the model in (1) such that only cars with a height of less than two meters can enter the barrier of car park B.
3. To reduce the time for finding a parking spot at car park B, the airport employs five employees serving as a park pilot. Cars guided by such an employee find a parking spot within one minute; the employee is in this minute not available for other cars. Show how the original model in (1) needs to be adjusted.

Exercise 3.3 The production process of a car toy consists of four steps: assembly, painting, drying, and packaging. In the assembly step, a production worker assembles four wheels and a chassis. This step takes five minutes. Next, a painter paints the product. This takes eight minutes. After painting, the product dries for at least twenty minutes before it can be packed. The room in which the products can dry is limited in size: at most ten products can dry there at the same time. Finally, a production worker is doing the packaging of the toy car. This takes ten minutes. The company employs three production workers and two painters. The capacity of every employee is one. Model the production process as a CPN with time. Show the initial marking.

Exercise 3.4 A catering company employs two waiters and five cooks. If the company handles the catering for a party with $n$ guests. If $n \leq 10$, i.e. with not more than 10 guests, the company rejects the request as there is no money convenience. If $n > 10$, then the following (simplified) process is followed. First, a cook prepares the meal. This takes $n/2$ man hours. Second, a waiter garnishes the food. This takes $n/10$ man hours. Finally, the meal is served. This step can be done by a cook or a waiter, and it takes $n/5$ man hours of a cook and $n/10$ man hours of a waiter. All activities are executed by persons having the right qualifications; for example, a waiter cannot act as a cook.

1. Model this business process as a CPN. The unit of time is 1 (man) hour.
2. Change the model from the previous question in such a way that, for serving a meal, either two waiters or one waiter and one cook are necessary.
Exercise 3.5 Model the following business process of an auction. A list of objects is sold in the order of that list. Every object is from a client and has a minimum price. A bid should exceed the minimum price and the previous bid if any. If there was no bid during the last ten time units, then the object is sold to the highest bidder. Whenever all objects are sold, a list with the sold objects, their price, and the new owner is produced. As a simplification, assume that there is always at least one bid.

Exercise 3.6 Consider the following business process of a take-away restaurant. The restaurant receives an order through a place order_in. Color set Order is one of the declarations given:

```
color Customer = string timed;
color Product = with coffee | tea | beer | fish | chips timed;
color Order = product Customer * Product timed;
var c:Customer;
var p:Product;
```

An example of an order is (“John”,beer); that is, one serving of beer for customer John. Customers can order only one item at a time. Each incoming order gets an order number to uniquely identify a request. It takes one minute to accept an order and to attach a number to it. The acceptance of an order triggers the production of food and drinks. Things are produced in parallel whenever possible, and each item is linked to a particular customer order (i.e., given an order (“John”,beer), the beer is produced specifically for John). It takes two minutes to prepare a drink (i.e., coffee, tea, or beer) and three minutes to prepare food (i.e., one serving of fish or chips). When the item is produced, the customer is called and the items are delivered. The delivery takes 1 minute. There are five employees working in the take-away restaurant. There is one employee accepting orders and delivering items to the customers. There are two employees preparing drinks and two employees preparing food, but employees preparing the food can also prepare drinks. An employee can only do one task at a time.

Model the restaurant based on the previous description. You will need to introduce additional declarations (e.g., for the order number). Customer orders do not need to be handled in a fixed order (i.e., one customer order can overtake another one), but there should not be unnecessary waiting (i.e., resources are eager to help customers and work in parallel when possible).

Exercise 3.7 Consider the following variant of the well-known dining philosophers problem involving forks and plates. Let \( n \) be the number of philosophers and \( k \) the number of plates. The \( n \) philosophers sit around a table, and there is one fork in-between any two philosophers sitting next to each other; that is, there are \( n \) forks. Philosophers alternate between thinking and eating. To eat, a philosopher needs to take his left and right fork. Moreover, he needs to also take a plate. The plates are located at the center of the table. After eating, the plate and the two forks are returned to their original positions, and the philosopher switches back to thinking mode.

Model this problem as a Colored Petri net with two parameters \( n \) and \( k \). Note that the structure of the net needs to be independent of these two parameters; that is, by increasing \( n \) by one, a philosopher and a fork are added, and by increasing \( k \) by one, a plate is added. Such changes should only affect the initial making of the CPN model and not its structure. Clearly show the CPN model including the color sets and other declarations used.
Solutions

Solution 3.1

This solution is just an example. There are many possible solutions to any given modeling exercise.

1. Figure 50 shows the resulting CPN. We have the following declarations:

   ```
   colset Robot = with R1|R2;
   colset ProdType = with A|B|C;
   colset Quality = with low|high;
   colset Prod = product ProdType * Quality;
   colset RobxProd = product Robot * Prod;
   var p: Prod;
   var pt : ProdType;
   var r : Robot;
   var q : Quality;
   ```

![Diagram](image1.png)

Figure 50: The Petri net of Exercise 3.1(1).

2. Figure 51 shows the resulting CPN. We have adjusted the following declarations:

   ```
   colset Robot = with R1|R2 timed;
   colset Prod = product ProdType * Quality timed;
   colset RobxProd = product Robot * Prod timed;
   ```

![Diagram](image2.png)

Figure 51: The Petri net of Exercise 3.1(2).
Solution 3.2

This solution is just an example. There are many possible solutions to any given modeling exercise.

1. Figure 52 shows the resulting CPN. We have the following declarations, remembering that function \( f() \) is already defined:

```ml
colset Car = INT timed;
fun g(i) = if i < 34 then 5 else 2;
var c:Car;
var i,j:INT;
```

However, in order to deploy it in CPN Tools, function \( f() \) needs to be concretely defined. For instance, we can say that function \( f() \) returns an integer value uniformly distributed between 1 and 10 (time units):

```ml
fun f() = discrete(1,10);
```

See page 191 of the book to see the random distribution function supported by CPN Tools.

2. Figure 53 shows the resulting CPN. We have the following additional declarations:

```ml
colset Height = INT;
colset CxH = product Car * Height;
var hh:Height;
```

3. Figure 54 shows the resulting CPN. We have the following additional declaration:

```ml
colset TU = UNIT timed;
```
Figure 53: The Petri net of Exercise 3.2(2).

Figure 54: The Petri net of Exercise 3.2(3).
Solution 3.3

This solution is just an example. There are many possible solutions to any given modeling exercise.

1. Figure 55 depicts the Petri net model extended with time. Transition `assemble`, `painting`, `drying`, and `packaging` model the four steps. For each of the objects wheel, chassis, painter, and production worker, we add a place. The intermediate products are stored in places `p1`, `p2`, and `p3`. Place `car` contains all produced toy cars. Place `p3` models the room where the painted cars dry. This place has a capacity of ten tokens. The capacity has been realized by adding place `free`. Initially, place `free` contains ten tokens, and it is connected in such a way that the sum of tokens in places `free` and `p3` is always equal to ten. Transition `assemble` consumes four tokens from place `wheel`, one token from place `chassis`, and one token from place `worker`. Furthermore, it produces a token with delay of five time units in place `worker` and in place `p2`. That way, we assure that the worker token and the product token are not available for the next five time units. The same approach is used to model the other production steps. In the initial marking shown in Figure 55, there are three workers, two painters, and ten places in the drying room available.

Figure 55: The model of Exercise 3.3(1).

2. As the resulting model should be an uncolored Petri net, we need to copy transitions `painting` and `drying` and their intermediate place. Figure 56 shows the respective model. Note that place `free` must be connected to transitions `b_drying` and `r_drying` to ensure the capacity of ten tokens for place `p3`.

Figure 56: The model of Exercise 3.3(2).

Solution 3.4

This solution is just an example. There are many possible solutions to any given modeling exercise.

1. Figure 57 depicts the model. The following declarations are necessary:
colset GuestNumbers = INT timed;
colset Cook = UNIT timed;
colset Waiter = UNIT timed;
var n : GuestNumbers;
var c : Cook;
var v, w : Waiter;

We model each of the four events “Prepare”, “Reject”, “Garnish”, “Server Meal_Waiter”, and “Server Meal_Cook” as a transition. We model each event as a transition. These transitions are connected to places Orders, Prepared, Garnished, and Completed of type GuestNumbers. A token in each of these places models the status of an order, and the token value specifies the number of guests. Places Cooks and Waiters serve as a store for cooks and waiters. These places are of type Cook and Waiter, respectively. The guard of transition Prepare specifies that only orders with more than ten guests are accepted. Vice versa, if orders are for 10 or few guests, the order is rejected (by firing transition “Reject”). Each transition “consumes” the employees needed to pass the respective step. The consumed employee tokens are returned to the respective places Waiter and Cook. The delay added to these tokens avoids that the respective employee can work on different things during this delay. The same delay is also added to the token representing the order status. The marking shown models that the catering company employs five cooks and two waiters.

Although the time delays are theoretically correct, they are not accepted in CPN Tools, as they are. In CPN Tools, the division operation is only allowed when the operands are real numbers. In the figure, in the time delays, the two operators of the divisions are integers. Differently from many programming languages, such as Java, integer numbers are not automatically converted into real numbers. This needs to be manually done through the function Real.fromInt(). For instance, for transition “Prepare”, the time delay needs to be specified as follows: Real.fromInt(n)/2.0. Please also note that the integer 2 needs to be written as the real 2.0. The division result is again a real number. Time delays need to be represented as integer values. Therefore, the result of the division needs to be converted back to an integer number, for instance, through the function Real.trunc(), which simply truncates the decimal part of real numbers. As such, the actual time-delay specification for transition “Prepare” is as follows: Real.trunc(Real.fromInt(n)/2.0). Figure 58 shows the correct time delays as accepted by CPN Tools (along with modeling the question (2) of the exercise).

Figure 57: The Petri net model of Exercise 3.4(1).

2. Figure 58 depicts the resulting Petri net model, where the differences with Figure 58 are highlighted in gray. For transition “Serve Meal_Waiter”, we need to model that two waiters
are necessary. We need to consume two tokens from place "Waiter", and these two tokens need also to be produced again in this place. We use the multiset notation \([v,w]\) to denote that two tokens \(v\) and \(w\) of type "Waiter" have to be consumed and produced. For transition "Serve Meal\_Cook", it has to consume and again produce a token for place "Waiters".

Figure 58: The Petri net model of Exercise 3.4(2).

Solution 3.5

This solution is just an example. There are many possible solutions to any given modeling exercise.

Figure 59 depicts the resulting CPN model. We need two transitions, "Take a Bid" and "Next Object", to model the bidding for and the selling of an object. The list of the definitions of color sets and variables are as follows:

```plaintext
colset Object = string;
colset Owner = string;
colset Price = int;
colset Bid = product Object * Owner * Price timed;
colset DBEntry = product Object * Owner * Price timed;
colset DB = list DBEntry;
var aList, aList2 : DB;
var p,p2 : Price;
var ow,ow2 : Owner;
var dbe : DBEntry;
var ob : Object;
colset TUnit= UNIT timed;
var n : INT;
```

Place "Bids" contains all bids and is timed. Figure 59 shows an example of a set of bids, each of which is modeled as a different token associated with the time when the bid was posed. Place "Sold" all sold objects. Place "Objects" stores all products to be sold. This place is of type "DB". Initially, each element of "DB" identifies the object, the owner, and the price.

Let us ignore the time constraint for a moment. Bidding is modeled by increasing the price of the first object of the "DBEntry" list in place "Objects". Furthermore, the name of the bidder replaces the name of the object’s owner as the bidder is the new owner, unless a subsequent bid "beats" the current best. The guard at transition "Take a Bid" ensures to only consider those bids that increase over the previous bid. If the current object is sold, then it is removed from the list in place "Objects" and added to the list of sold objects in place "Sold". In the model, we assume that all bids in place "Bids" are for the current object.

Let us now consider the modeling of the time-out. Intuitively, each bid produces a token of ten time units delay in place "Timer". If there is no other bid submitted within the next ten time
units, then transition Next Object is enabled. If there is another bid within this delay, then the previously produced timed token has to be removed. This is, however, not possible, because this token is due to the delay not available. Therefore, Figure 59 has a place Counter. Each bid also increments this counter. If a timed token is available, then the guards of transitions Remove and Next Object check whether the counter is equal to one (i.e., the current bid is the only one that has been submitted during the last 10 time units). In this case, transition Next Object is enabled; otherwise, transition Remove is enabled. Both transitions decrement the counter by one and consume the timed token.

Figure 59: A CPN model of Exercise 3.5.
Solution 3.6

This solution is just an example. There are many possible solutions to any given modeling exercise.

Figure 60 depicts the Petri net. The following declarations have been added:

```plaintext
colset Products = list Product timed;
colset OrderNo = int;
colset Request = product OrderNo * Customer * Products timed;
colset Resource = with server | drinks | food;
var l:Products;
var ord:Order;
var n:OrderNo;
fun elt(p,[]) = false |
elt(p,p2::l) = (p=p2) orelse elt(p,l);
```

Figure 60: The Petri net of Exercise 3.6.
Solution 3.7

This solution is just an example. There are many possible solutions to any given modeling excercise.

Figure 61 depicts the Petri net. We have the following declarations:

val n = 10;
val k = 3;
colset PH = index ph with 1..n;
colset CS = index cs with 1..n;
colset Dish = unit with D;
var p: PH;
fun Chopsticks(ph(i)) = 1`cs(i) ++ 1`cs(if i=n then 1 else i+1);

Figure 61: The Petri net of Exercise 3.7.
4 Colored Petri nets: Hierarchy

Exercise 4.1 Consider the following concert reservation system. In this system, we give attention to the booking of seats for particular events. The system offers an interface to the user modeled by four places create_event, request, reject, and confirm. A token in place create_event creates an event—for example, “Rammstein at Gelredome on 6 December 2009”. Once the event is created, people can request for tickets by producing a token in place request. Such a request is identified by the person’s name and the event id (i.e., a string uniquely identifying the event—for example, “Rammstein at Gelredome on 6 December 2009”). We assume that for each event a fixed number of seats is available, say, 1000. These seats are numbered 1, 2, . . . , 1000. A request is rejected if no more seats are available. A request is confirmed if there is still a seat available. In this case, the seat number is returned. An example token passed back by place confirm is (“Peter Jansen”, “Rammstein at Gelredome on 6 December 2009”, 542), indicating that Peter successfully obtained a reservation for the Rammstein concert and that his allocated seat number is 542.

The following color sets, functions, and variables are given:

```
color Name = string;
color Event = string;
color Seat = int with 1..1000;
color NxE = product Name * Event;
color NxExS = product Name * Event * Seat;
fun seats() = Seat.all();
var n:Name;
var e:Event;
var s:Seat;
```

Note that the function `seats()` returns the list [1, 2, 3, . . . , 1000].

1. Model the reservation system based on the previous description. The internal database of the office should keep track of free seats and reserved seats (with corresponding name information) for each event. Clearly comment any additional declarations, arc inscriptions, color sets, and other extensions. Start with an abstract model and refine this model using the concept of hierarchy.

2. Extend the CPN model with payments and the printing of tickets. All tickets for a particular event are printed at the same time and only tickets that have been paid for are printed. It is not possible to pay twice for the same seat or for a seat that has not been reserved. Such an invalid payment is declined by the reservation system, and a rejection message is sent back to the environment. Valid payments, on the other hand, are confirmed by sending a confirmation message. The printing of the tickets is triggered by the environment (a single message triggers the printing of all paid tickets for an event), and the printed tickets are sent from the reservation system to the environment. Each ticket has a name, an event id, and a seat number. After the tickets for an event are printed, all subsequent reservations or payments for this event are rejected. Clearly comment the new interface places, additional declarations, arc inscriptions, color sets, and other extensions. Again, start with an abstract model and refine this model using the concept of hierarchy.

Exercise 4.2 Consider the top-level CPN model in Figure 62. This model depicts a transport system consisting of an environment placing requests, a routing system generating commands, and the actual trucks driving from one location to another to load and offload goods. In addition, the following variables are given:

```
var t:Truck;
var a:Action;
var c:Capacity;
var v,v0,v1,v2:Volume;
```
var 1,10,11,12:Location;
var rid:ReqID;

We assume that there are ten trucks (1, . . . , 10). Each of these trucks has a capacity of 100 cubic meters. The environment sends requests of type Request to the routing system. Each request has an identifier of type ReqID, the location from which the goods need to be shipped, the location to which the goods need to be shipped, and the volume of the shipment in cubic meters. So a value (rid,l1,l2,v) of type Request corresponds to the request for a shipment from l1 to l2 that has a volume v and is identified by rid. Once the request has been executed; that is, the shipment is completed, an acknowledgment is sent through place done.

The routing system collects these requests and issues real-time instructions to the truck drivers via place command; that is, a continuous stream of commands is sent to the trucks and these are processed one by one. A value (t,l1,rid,a) of type Command states that truck t should move to location l1 to do action a. There are two types of actions: “load” and “offload”. Both are parameterized by rid. For example, command (5,”Rotterdam”,45646,load) means that truck 5 should move to Rotterdam to load shipment 45646. Any request results in two commands: one to load the goods at the start location and one to offload the goods at the destination. Commands are executed one by one; that is, for a truck there cannot be multiple active commands. There may be times where a truck contains goods corresponding to different requests. Multiple subsequent actions can be performed at the same location (e.g., offloading three shipments and loading two new ones). This can be achieved by sending subsequent commands referring to the same location. The execution of each command is acknowledged via place ready of type Truck.

As indicated before, each truck has a capacity, and each shipment requires a volume. The control system should avoid that the maximum capacity is exceeded, but it should allow for combined shipments; that is, there are times where a truck contains goods corresponding to multiple requests. There is no need to split shipments over multiple trucks, because each request results in precisely two commands for a truck (“load” and “offload”). There is continuous flow of requests coming from the environment according to the stochastic interarrival distribution discrete(10,15). The time required to execute a command is given by function delay. Given two locations l1 and l2, delay(l1,l2) is the time required to move from l1 to l2 and to (off)load one shipment.

Model the three subpages environment, routing system, and trucks according to the preceding specification. There is no need to think of a sophisticated planning and routing strategy as long as the solution meets the specification (e.g., things should t in a truck and it should be possible to combine shipments).

Exercise 4.3 Consider an organization that produces different flavors of yogurt (e.g., peach, pear,
and strawberry). The organization has five machines to produce yogurt; each machine can produce any flavor. Each machine has to be cleaned at regular intervals and when the production changes to a new flavor. The organization employs seven workers with different capabilities: two cleaners for cleaning the machines and five production workers for running the machines.

The organization receives orders from customers. Each order has an order ID, a flavor, and a volume (expressed in “units”). An example order is the triple \( (17, \text{peach}, 2) \) specifying an order of two units of peach yogurt identified by the order ID 17.

Suppose a very simple scheduler that assigns an incoming order to a machine that has just produced yogurt of the same flavor. One production worker is needed to run the machine and the production step takes one hour for each unit; that is, for order \( (17, \text{peach}, 2) \), the production takes two hours because two units of yogurt need to be produced. If no such machine is available, a machine is cleaned by two cleaners. Subsequent the order can be produced involving one production worker. Cleaning a machine takes two hours. After the yogurt of an order has been produced, it is delivered. Figure 63 shows the top-level CPN model that depicts the interface between the organization and its environment:

1. Model subpage Producer based on the previous description as a classical Petri net (i.e., a Petri net with indistinguishable black tokens without time). It is not necessary to distinguish different flavors or workers with different capabilities.

Figure 64 illustrates how the environment is designed; the necessary declarations are as follows:

```plaintext
colset INT = int timed;
colset Id = INT;
colset Vol = int with 1..3;
colset Flavor = with peach | pear | strawberry;
colset Order = product Id * Flavor * Vol timed;
var id:Id;
var f:Flavor;
var v:Vol;
```

2. Model subpage Producer based on the previous description as a CPN. Make sure that all activities (cleaning and production) are executed by workers having the right capability; for example, a production worker cannot clean a machine. Assume that the scheduler processes incoming orders according to the FIFO (First In, First Out) principle. If the first order cannot be processed because all available machines have produced yoghurt of a different flavor, then one machine is cleaned to process this order. Initially, two machines can produce peach yogurt, two machines pear yogurt, and one machine strawberry yoghurt.

Clearly identify any additional declarations, arc inscription, color sets, and other extensions that are needed. Use the CPN notation used in CPN Tools or the notation used in the lecture material.
3. Extend the CPN model in (2) with a more efficient scheduler. If the first order can only be produced after cleaning a machine, but there is a pending order that can be processed with a machine available, then process the latter order; that is, clean a machine only if necessary. Clearly identify any additional declarations, arc inscription, color sets, and other extensions that are needed. Use the CPN notation used in CPN Tools or the notation used in the lecture material.

**Exercise 4.4** In this exercise, we consider a vending machine.

Figure 65 depicts the top-level CPN model that specifies the interface between the user and the vending machine. A user orders a set of products via place order. If none of the ordered products is available, then the vending machine rejects this order (place reject). Otherwise, an offer is made which shows the user which products can be purchased at what price (place offer). Then, the user can insert one or more coins into the vending machine to pay the requested price (place coin). After the user has paid the requested price or more, the vending machine adds the inserted amount to its cash register, returns the change (place return), and dispenses the ordered products (place product). The vending machine implements a simple timeout mechanism: After accepting the order, the machine expects the user to insert coins within a certain time interval. At the end of this time interval, the machine checks whether the user has inserted a coin. If this is the case, the coin is consumed and the timeout is reset; otherwise, a timeout occurs and the vending machine returns all the coins the user has inserted (place return).

1. Model subpage VendingMachine based on the previous description as a classical Petri net (i.e., a Petri net with indistinguishable black tokens and without time). Note that the user inserts coins one-by-one (i.e., there are possibly multiple tokens in place coin), but you can model the return of these coins in one atomic action (i.e., one token produced for place return represents all returned coins).
2. Provide a precise model of subpage VendingMachine as a CPN. Use the previous description and the additional details below. The following declarations are given:

\[
\begin{align*}
\text{colset} & \text{ Product } = \text{ string; } \\
\text{colset} & \text{ LP } = \text{ list Product; } \\
\text{colset} & \text{ Coin } = \text{ subset } \text{ INT} \text{ with } [5, 10, 20, 50, 100, 200]; \\
\text{colset} & \text{ Coins } = \text{ list Coin; } \\
\text{colset} & \text{ Price } = \text{ int; } \\
\text{colset} & \text{ LPxP } = \text{ product LP * Price; }
\end{align*}
\]

\[
\begin{align*}
\text{var} & \text{ lp:LP; } \\
\text{var} & \text{ p:Price; } \\
\text{var} & \text{ c:Coin; } \\
\text{var} & \text{ lc:Coins; }
\end{align*}
\]

\[
\begin{align*}
\text{fun} & \text{ value([], )} = 0 | \\
& \text{ value(c::lc)} = c + \text{value(lc)};
\end{align*}
\]

\[
\begin{align*}
\text{fun} & \text{ calChange(given, toPay, [], )} = [] | \\
& \text{ calChange(given, toPay, c::lcoins)} = \\
& \quad \text{if (value(given) - toPay )} \geq c \\
& \quad \text{then c:: calChange(given, toPay + c, c::lcoins) } \\
& \quad \text{else calChange(given, toPay, lcoins);}
\end{align*}
\]

\[
\begin{align*}
\text{fun} & \text{ change(given, toPay) = calChange(given toPay, [200,100,50,20,10,5]);}
\end{align*}
\]
Order To decide which of the ordered products are available, the vending machine has a data base, storing for each product the price and the number of items in stock. For example, for an order [Coke, Coke, Mars, Twix], the machine may return ([Coke, Twix], 170), meaning only one Coke and one Twix can be purchased and will take Euro 1.70. As can be seen from this example, the price is stored in Euro cents. That way, type int can be used. Initially, the stock of the machine contains two Cokes, three Spa, one Mars, and two Twix. The prices are Euro 0.80 for a Coke, Euro 0.50 for a Spa, Euro 1.00 for a Mars, and Euro 0.90 for a Twix.

Payment The user must insert one or more coins to purchase her products. Each coin is modeled by one token. As can be seen from the declaration, the user can insert only 5 cent, 10 cent, 20 cent, 50 cent, 1 Euro (100 cents), and 2 Euro (200 cent) coins. After the offer is shown to the user, the user has a time interval of five time units to insert at least one coin. At the end of this time interval, the machine cancels the order if no coin has been inserted; that is, a timeout occurs. If at least one coin has been inserted, then one coin is consumed by the machine and the timeout is reset. The same timeout strategy is applied for all additional coins: As long as the amount is less than the requested price, the vending machine offers the user a time interval of five time units to insert the next coin; otherwise, a timeout occurs and the vending machine returns all coins that have been inserted so far. The mechanism of the machine ensures that all coins are returned. You can assume that the cash register of the vending machine always contains exact change; that is, a user receives always the exact difference of its inserted coins and the price. Initially, the cash register contains five coins for each value. Two functions are provided to simplify the dealing with the coins. Function value returns the value of a list of coins; for example, value([5, 50, 20, 50]) returns 125 (i.e., Euro 1.25). Function change returns the change as a list of coins, based on the inserted list of coins (parameter given) and a price toPay. For example, change([5, 50, 20, 50], 95) returns the list of coins [20, 10]—this function calculates the difference the value of the inserted coins and the price (here: 125 - 95 = 30) and returns a list of coins of the same value.

Output After the user paid, the vending machine updates its cash register and its data base. Clearly identify any additional declarations, arc inscription, color sets, and other extensions that are needed. Use the CPN notation used by CPN Tools or the notation used in the lecture material.

Exercise 4.5 Consider the top-level CPN model of a Starbucks coffee shop in Figure 66.

![Figure 66: Top-level page describing the interface of a Starbucks coffee shop to the customer.](image)

The two places shown are of type Order. Color set Order is one of the declarations given:

colset Customer = string timed;
colset Product = with Latte | Cappuccino | Americano | Ristretto | Espresso timed;
colset Products = list Product timed;
colset Order= product Customer * Products timed;
The environment can place orders by sending a token via place customer into the Starbucks subpage. An example of an order is ("Wil", [Latte, Cappuccino, Latte, Espresso, Espresso]) (i.e., two Lattes, two Espressos and one Cappuccino for customer Wil). The requests coming in via place customer in are initially handled by Mary; that is, Mary accepts the order and handles the payment. The time required to do this depends on the number of items. If the customer orders \( n \) items, then Mary needs \((n + 1) \cdot 10\) seconds to handle the request. Hence the example order mentioned previously takes one minute. After recording the order and payment, the orders are allocated to an available employee in a first-come, first served manner. There are five employees (Mary, John, Eve, Sue, Pete) also referred to as resources. If an order is allocated to a particular employee, the employee starts preparing the coffees one-by-one. It takes 40 seconds to make a Latte or Cappuccino. The other drinks can be prepared in 20 seconds (because there is no need to heat any milk). All the drinks corresponding to a particular order are made by the same person and this person can only prepare one drink at a time. Note that Mary can play two roles: She can take orders and prepare the drinks for a particular order (but not at the same time). There is only one coffee machine that has a capacity of three, that is, at most three coffees can be prepared in parallel. Once all drinks have been prepared, the customer leaves via place customer out.

Model the subpage Starbucks based on the previous description. Clearly indicate any additional declarations, arc inscriptions, and color sets. Use the CPN notation used in CPN Tools or the notation used in the lecture material.

**Exercise 4.6** Consider the top-level CPN model as in Figure 67.

![Figure 67: Top-level page describing the interface of subpages Senders, Shipment, and Receivers.](image)

The following declarations are given:

colset Product = string;
colset ID = int timed;
colset Price = int;
colset Weight = int with 1..9;
colset Shipment = product ID * Product * Price * Weight timed;
var prod:Product;
var id:ID;
var price:Price;
var w:Weight;
var s:Shipment;

The subpage senders creates shipments of the form \((id, prod, price, w)\). Every shipment has a unique identifier (id), refers to a product (prod), has a value (price), and a weight (w). The minimal weight of a shipment is 1 kg and maximal weight of a shipment is 9 kg. Every shipment sent from senders to shipment via place send should eventually be delivered and hence moved from shipment to receivers via place receive.

Subpage shipment models a delivery service that owns ten delivery vans. Each van can hold a maximal total weight of 50 kg. The delivery service uses the following policy. Shipments are handled in first-come-first-serve order for shipments with the same value. However, shipments
with a higher value (as indicated by the price field) have priority over shipments with a lower value. Vans are loaded using this policy and multiple delivery vans may be loaded concurrently. A delivery van starts to drive if it is loaded for at least 80 percent—that is, 40 kg. Hence, the total cargo of a delivery van that starts driving is in-between (and including) 40 and 50 kg. One can abstract from the locations of the deliveries; simply assume that the van needs between 400 and 600 minutes to deliver all of its shipments and is then available again for a new batch of shipments.

1. Provide a CPN model for subpage shipment. Clearly show the model and list the additional declarations used. Make sure that the model is readable and list any assumptions needed.

2. Comment on the policy used by the delivery service. How can it be improved?

Exercise 4.7 We consider the treatment of patients in an accident and emergency department of a hospital. The top-level CPN model in Figure 68 specifies the interface between the patients and the hospital.

![Figure 68: Top-level page describing the interface of subpages Patients and Hospital.](image)

On an abstract level, patients enter the hospital via place enter, are treated by the medical staff of the hospital (i.e., doctors and nurses) and finally leave the hospital via place leave.

Patients, who enter the hospital, are identified by their name (you can assume that this name is unique) and a list of symptoms. For example, the pair (“Rob”, [flue, pain]) specifies that patient Rob has symptoms of a flue and pain.

Once a patient has entered the hospital, a secretary registers the patient. To this end, the secretary checks whether the patient already exists in the database. If this is not the case, an entry consisting of the patient’s name and an empty documentation is added to the database. In our example, (“Rob”, []) would be added. The secretary also determines the priority of the patient based on his symptoms. This is modeled by the provided function getPriority().

Patients must wait until the clinical personal needed for their treatment is available. There are two doctors and five nurses; see values vNurses and vDoctors. The selection of a patient and the clinical personal works as follows: To treat a patient, one doctor and at least one nurse are required. The actual number of nurses is determined by the number of symptoms of the patient. For each symptom, one nurse with the right qualification for treating this symptom is assigned (note that nurses are identified by their qualification). For example, to patient Rob, we would assign one doctor, one nurse who is qualified for treating flue, and one nurse who is qualified for treating pain. You can assume that the list of symptoms of a patient is a set rather than a multiset; that is, each symptom occurs only once. For the sake of simplicity, we assume that there is no fixed order between patients having the same priority. However, we distinguish between patients with high and low priority. Patients with high priority take priority over patients with
low priority; that is, whenever there is a high-priority patient waiting, low-priority patients have to wait. A patient is selected if there a doctor and nurses with the right qualifications are available.

Once a patient enters the doctor’s office, the treatment starts. The treatment of a low-priority patient takes 10 time units; the treatment of a high-priority patient takes 15 time units. Function getTreat() models which treatment is actually applied. Then, the database of the hospital is updated; that is, to the patient’s database entry, a pair (listOfSymptoms, treatment) is added. For example, if getTreat() returns talk for Rob, then a pair ([flue, pain], talk) is added to Rob’s database entry. After updating the database, the patient leaves the hospital; in our example, a token with value “Rob” is produced on place leave.

Provide a precise model of subpage Hospital as a CPN. In particular, provide an adequate model for

- Adding new patients to the database;
- Queuing patients regarding their priority;
- Selecting a patient for whom the required personal is available;
- Updating the database as part of the treatment.

Use the previous description and the additional details below. The following declarations are given:

```plaintext
colset Name = string;
colset Symptom = with flue|cut|pain;
colset Lsym = list Symptom;
colset Patient = product Name * Lsym;
colset Priority = with low|high;
colset Treatment = with bandage|medicine|talk;
colset Docu = product Lsym * Treatment;
colset Ldocu = list Docu;
colset DBentry = product Name * Ldocu;
colset DB = list DBentry;
colset Doctor = string;
colset Nurse = Symptom;
```

// initial markings: there are five nurses and two doctors
val vNurses = [flue, flue, pain, cut, cut];
val vDoctors = ["D1", "D2"];

// samples a treatment with equal probability
fun getTreat() = Treatment.ran();

// samples a priority from a Bernoulli distribution
fun getPriority() = if bernoulli(0.9) = 1 then low else high;
```

Clearly identify any additional declarations, arc inscription, color sets, and other extensions that are needed. Use the CPN notation used by CPN Tools or the notation used in the lecture material. Make sure that your model and all inscriptions are readable.
Solutions
Solution 4.1

This solution is just an example. There are many possible solutions to any given modeling exercise.

1. Figure 69 depicts the CPN model.

```
color Name = string;
color Event = string;
color Seat = int with 1..1000;
color NxE = product Name * Event;
color NxExS = product Name * Event * Seat;
color FreeSeats = list Seat;
color TakenSeat = product Seat * Name;
color TakenSeats = list TakenSeat;
'color EventData = product Event * FreeSeats * TakenSeats;
fun seats() = Seat.all();
var n:Name;
var e:Event;
var s:Seat;
var fs:FreeSeats;
var ts:TakenSeats;
```

Figure 69: A CPN model of Exercise 4.1(1).

Beside the four interface places `create_event`, `request`, `reject`, and `confirm`, which have been given by the specification, a place `database` as been added. Furthermore, there are three transitions modeling the three possible events. For place `database`, which is of type `EventData`, the declarations have been extended. Each token in this place specifies for a single event the event id, the number of free seats, and a list of reserved seats. The model is straightforward: transition `create_event` adds an event (together with 1000 free seats and an empty list of taken seats) to the database. If a customer requests for an event that is sold out (i.e., no free seats are available), a token is produced in place `reject`. If there is at least one free seat, then this seat is removed from the list of free seats, the name and the seat number are added to the list of taken seats, and a token is produced in place `confirm`.

2. Figure 70 depicts the modified CPN model.

The three newly added transitions `reject_payment`, `confirm_payment`, and `print` model the three respective events. Places `pay`, `reject_payment`, `confirm_payment`, `print_tickets`, and `tickets` model the interface to the environment. In the declarations, we added a Boolean `Paid` to each taken seat, indicating whether this seat has been paid. If a customer books a ticket, then this Boolean is initially set to false. Consider now the three events. If customers have paid for their ticket (i.e., for a given name and event id, there is no reserved seat which has not been paid), then this payment request is rejected. Otherwise, this entry is removed from
Figure 70: A CPN model of Exercise 4.1(2).
list \textit{TakenSeats}. In addition, a new list containing only the seat information of the requester (where the Boolean \textit{Paid} is true) and list \textit{TakenSeats} are concatenated. In this case, the successful payment is confirmed. To print the tickets for an event, the respective database entry is taken and the free and reserved seats are removed from this database entry. That way, we make sure that all future requests for this event will be rejected. In addition, function \texttt{create} filters from list \textit{TakenSeats} all elements where customers have paid and produces this list in place \textit{tickets}.

\textbf{Solution 4.2}

\begin{center}
This solution is just an example. There are many possible solutions to any given modeling exercise.
\end{center}

Figure 71 depicts page \textit{environment}. Transition \texttt{request} models the sending of a request to the routing system. A request is of type \textit{Request}. To generate a request identifier, every firing of transition \texttt{request} increments the counter in place \textit{counter}. The token produced in place \textit{counter} has a delay of \texttt{discrete(10,15)} time units according to the specification. Transition \texttt{ready} models the receipt of the acknowledgment message, which is identified with the identifier.

\begin{verbatim}
color Volume = int;
color Location = string;
color ReqID = int timed;
color Request = product ReqID * Location * Location * Volume timed;
var v:Volume;
var l0,l1:Location;
var rid:ReqID;
\end{verbatim}

Figure 71: Page \textit{environment} of Exercise 4.2.

Figure 72 depicts page \textit{routing system}. Place \textit{available} contains initially all trucks. This place is of type \textit{TruckInfo} and specifies for each truck the capacity, the current volume, and the current request identifier. Each truck has according to the specification a capacity of 100 m$^3$, is initially empty, and the identifier is equal to zero. Places \textit{available}, \textit{pending commands}, and \textit{executed commands} model the possible states of a truck. If a request arrives, transition \texttt{load\_command} chooses an available truck and orders it via place \textit{command} to the start location. The respective truck changes to state \textit{pending commands} (i.e., the truck is on the way to a location), and the request together with the truck chosen is stored in place \textit{load-in-progress}. The latter place is of type \textit{TRequest}—that is, the product of types \textit{Truck} and \textit{Request}. The guard of \texttt{load\_command} ensures that the truck has the capacity to load the goods.

We model a truck that arrives at its location by a token in place \textit{ready}. The firing of transition \texttt{arrive} changes the state of the truck from place \textit{pending commands} to place \textit{executed commands}; that is, the truck is loaded or unloaded. Transition \texttt{loaded} models the loading of the truck. After the loading, the truck is available. The firing of transition \texttt{offload\_command} models that the truck driver gets the command to drive to his destination. Similar to transition \texttt{load\_command}, the state
of the truck changes from available to pending_commands. If the truck arrives at its destination, a token is produced in place ready. If transition arrived fires, then the state of the truck is updated to executed_commands. The truck can be offloaded by firing transition offloaded, thereby updating the volume of the truck. Afterward, the truck is again available.

Figure 72 allows a truck to perform multiple subsequent actions. If transitions load_command and offload_command have the same enabling time, then goods of another request can be loaded on the truck. As we do not model a sophisticated scheduling system, the current location of a truck is irrelevant and for this reason not stored in the places of type TruckInfo.

Figure 73 depicts page trucks. Place location stores for each truck its location. The initial location of each truck is “nowhere”. The firing of transition start takes the respective truck from its current location l0. After a delay of delay(l0,l), transition complete fires modeling that the truck reaches its destination l. As a delay, we chose a discrete distribution.

---

```plaintext
var t:Truck;
var a:Action;
var c:Capacity;
var v,v0:Volume;
var l0,l1,l2:Location;
var rid,rid0:ReqID;
```

---

Figure 72: Page routing_system of Exercise 4.2.
Solution 4.3

This solution is just an example. There are many possible solutions to any given modeling exercise.

1. The classical Petri net is shown in Figure 74.

2. The CPN model is shown in Figure 75 and the declarations in Figure 76.

3. The CPN model is shown in Figure 77 and the declarations in Figure 76.
Figure 75: CPN model of Exercise 4.3(2).

Figure 76: Declarations for Figures 75 and 77.

Figure 77: CPN model of Exercise 4.3(3).
Solution 4.4

This solution is just an example. There are many possible solutions to any given modeling exercise.

1. The classical Petri net is shown in Figure 78.

![Figure 78: CPN model of Exercise 4.4(1).](image)

2. The CPN model is shown in Figure 79 and the declarations in Figure 80.

64
Figure 79: CPN model of Exercise 4.4(2).
Solution 4.5

This solution is just an example. There are many possible solutions to any given modeling exercise.

The CPN model is shown in Figure 81. The following additional declarations have been used:

```
colset RO = product Resource * Order timed;
colset OL = list Order;
colset MachineState = int;
var ord:Order;
var ol:OL;
var n:MachineState;
fun ordertime(c,l) = 10 + (size(l) * 10);
fun maketime(p:Product) = if p = Latte orelse p=Cappuccino then 40 else 20;
```
Solution 4.6

This solution is just an example. There are many possible solutions to any given modeling exercise.

1. The CPN model is shown in Figure 82. The following addition declarations are used:

```plaintext
var w1,w2:Weight;
colset Shipments = list Shipment timed;
var sl,sl2: Shipments;

fun sadd((id1,prod1,price1,w1),[]) = [(id1,prod1,price1,w1)] |
    sadd((id1,prod1,price1,w1),(id2,prod2,price2,w2)::sl) =
        if price1 > price2 then (id1,prod1,price1,w1):
            (id2,prod2,price2,w2)::sl
        else (id2,prod2,price2,w2)::sadd((id1,prod1,price1,w1),sl);
fun sum([]) = 0|
    sum((id,prod,price,w)::sl) = w+sum(sl);
fun drivetime() = discrete(400,600);
```

2. The above solution has the problem that multiple vans are loaded concurrently; that is, they are all being loaded at the same time. As a result it may take long before a van has sufficient shipments to start driving (40 kg). Hence, it seems better to only load only one van at the same time. Another (less serious) problem is that shipments are loaded the moment a van is available and shipments are queuing. They are taken out of the queue when they are loaded even when the van does not start to drive. As a result, shipments with a higher price may need to wait for shipments with a lower price.
This solution is just an example. There are many possible solutions to any given modeling exercise.

The CPN model is shown in Figure 83. The following addition declarations are used:

\[
\begin{align*}
\text{colset Lnurse} &= \text{list Nurse}; \\
\text{colset InTreat} &= \text{product Name} \times \text{Lsym} \times \text{Doctor} \times \text{Lnurse} \text{ timed}; \\
\text{var } n : \text{Name}; \\
\text{var } s : \text{Symptom}; \\
\text{var } ls : \text{Lsym}; \\
\text{var } u : \text{Priority}; \\
\text{var } db : \text{DB}; \\
\text{var } d : \text{Doctor}; \\
\text{var } ln : \text{Lnurse}; \\
\text{fun updateDB}(n, []) &= [(n, [])] | \\
& \quad \text{updateDB}(n, (n1, d)::l) = \begin{cases} 
\text{if } n = n1 & \text{then } (n1, d)::l \\
\text{else } (n1, d)::\text{updateDB}(n, l); 
\end{cases} \\
\text{fun addDB}(n, ls, t, []) &= [] | \\
& \quad \text{addDB}(n, ls, t, (n1, d)::l) = \begin{cases} 
\text{if } n = n1 & \text{then } (n1, (ls, t)::d)::l \\
\text{else } (n1, d)::\text{addDB}(n, ls, t, l); 
\end{cases}
\end{align*}
\]
Figure 83: CPN model of Exercise 4.7.
5 Functions

Exercise 5.1 Define a polymorphic function \textit{elem} that checks whether a given element is contained in a list.

Examples: \text{elem}(2,[1,2,3]) = true; \text{elem}(4,[1,2,3]) = false

Exercise 5.2 Define a polymorphic function \textit{del} that deletes the first occurrence of an element in a list.

Examples: \text{del}(2,[1,2,3,2])=[1,3,2]; \text{del}(2,[1,3])=[1,3]

Exercise 5.3 Define a polymorphic function \textit{dell} that deletes a list \textit{l1} from a list \textit{l2}.

Examples: \text{dell}([1,2],[4,1,2])=[4]; \text{dell}([1],[2,5])=[2,5]

Exercise 5.4 Show that we can use function \textit{foldr} to define the concatenation of two lists without any need for recursion. Define a function \textit{conc} such that \text{conc}([a, b, c], [d, e, f]) evaluates to [a, b, c, d, e, f].

\textit{foldr \textit{f \ z \ l}} returns \textit{f(e1, f(e2, ..., f(\textit{en}, z) ...))} where \textit{l} = \text{[e1, e2, ..., en]}.

Exercise 5.5 Define a polymorphic function \textit{reverse} that reverses any list; for example, reverse([1,2,3]) should return [3,2,1].

Exercise 5.6 Write a polymorphic function \textit{exists} that checks whether an arbitrary list contains an element that satisfies a predicate \textit{p}. The return value of this function is true if such an element exists and false otherwise.

Exercise 5.7 Write a polymorphic function \textit{filter} that collects from an arbitrary list all elements that satisfy a predicate \textit{p}. The return value of this function is the empty list [] if no such element exists and a list with all elements satisfying \textit{p} otherwise.

Exercise 5.8 Define a polymorphic function \textit{len} that gives the length of a given list.

Example: \text{len}([1,2,3])=3.

Exercise 5.9 Write a polymorphic function \textit{dropn} that returns what is left after dropping the first \textit{n} elements from a given list.

Example: \text{dropn}([1,2,3,4,5],2)=[3,4,5].

Exercise 5.10 Write a polymorphic function \textit{returnn} that returns the first \textit{n} elements from a given list.

Example: \text{returnn}([1,2,3,4,5],2)=[1,2].

Exercise 5.11 CPN ML provides a function \textit{map \textit{f \ l}} that uses a function \textit{f} on each element in list \textit{l} and returns a list with all the results. Use function \textit{map} to project list prodList to a list containing only the first element of each tuple:

\text{Color prod} = \text{product int * string};
\text{Color prodList} = \text{list prod};

Example: \text{[(1,"s"),(2,"t")]} is projected to [1,2].

Exercise 5.12 This exercise is an continuation of the Brisbane CityCat system; see Exercise 2.12.

The Brisbane CityCat system is an urban transportation system using catamarans to quickly move people along the Brisbane River. Let us assume that there are four stops named A, B, C, and D. CityCats move from one stop to the other, first upstream (A,B,C,D) and then downstream (D,C,B,A). There are ten CityCats. Initially, all CityCats are in a dedicated harbor denoted by \textit{X}. Depending on the workload, CityCats are put into service (moved from harbor \textit{X} to stop \textit{A}) or taken out of service (moved from stop \textit{A} to harbor \textit{X}). The number of CityCats in service may, therefore, vary between zero and ten. Possible moves of a CityCat are X,A,B,C,D,C,B,A,X;
X,A,B,C,D,C,B,A,X; or X,A,B,C,D,C,B,A,X. A CityCat can only turn at stop D; that is, a move X,A,B,A,X or X,A,B,C,A,X is not possible. The stops have a capacity of one; that is, only one CityCat can dock at a particular stop at a time. The capacity of the river is large enough to fit all CityCats in-between any two stops. Model the Brisbane CityCat system as a CPN (including its initial marking). Moreover, take the following five aspects into account:

1. Distinguish individual CityCats (1,2, . . . , 10).
2. There are two types of CityCats: slow and fast ones. CityCats 7, 8, 9, and 10 are of a newer generation; they only need five minutes to move from one stop to another. The older CityCats (1,2, . . . ,6) need ten minutes.
3. CityCats are not allowed to overtake one another. Therefore, a slower CityCat may slow down a faster one. Moreover, there can be a queue of CityCats in front of a stop. In this case, a first-come, first-served (FCFS) queuing discipline is used.
4. Upstream CityCats have priority over downstream CityCats. For example, if there are CityCats queuing for stop B, then the CityCats originating from A (upstream) have priority over CityCats originating from C (downstream).
5. Each stop at A, B, C, or D takes five minutes to allow passenger to embark or disembark.

**Exercise 5.13** Consider the following Petri net with an inhibitor arc connecting t3 and p; see Figure 84.

![Figure 84: Illustration of the net with an inhibitor arc.](image)

Only a fragment of the total net is shown; that is, t1, t2, and t3 have connections to places in the rest of the net but are not shown. The inhibitor arc specifies the requirement that place p needs to be empty when t3 fires; that is, t3 is only enabled if all of its input places contain sufficient tokens and place p contains zero tokens. Note that the inhibitor arc has no effect on the production or consumption of tokens.

1. Assume that place p is 5-bounded; that is, there can never be more than five tokens in place p. Modify the fragment shown such that the inhibitor arc is removed without changing the behavior of the rest of the net; that is, replace the fragment depicted by a behaviorally equivalent classical Petri net fragment.
2. Replace the fragment with an equivalent Colored Petri Net fragment. Suppose that place p is of type T (e.g., `colset T = string`). It is not allowed to assume that place p is bounded. Clearly indicate the color sets and arc inscriptions.
3. Assume that the inhibitor arc is replaced by a reset arc connecting t3 and p. The semantics of this arc is that all tokens in p are removed when firing t3; that is, if p contains k tokens, then the firing of t3 implies removing all k tokens. Note that the enabling of t3 does not depend on the number of tokens in p; that is, if all input places contain sufficient tokens, t3 can fire even if p is empty. Again it may be assumed that place p is 5-bounded. Modify the
fragment shown such that the reset arc is removed without changing the behavior of the rest
of the net; that is, replace the fragment containing the reset arc by a behaviorally equivalent
classical Petri net fragment.

4. Replace the previous fragment by an equivalent Colored Petri Net fragment. It is not allowed
to assume that place $p$ is bounded; but it may be assumed that the values of the tokens in
$p$ do not matter. Clearly indicate the color sets and arc inscriptions.
Solutions

Solution 5.1
fun elem(e,[]) = false |
  elem(e,x::l) = if e=x then true else elem(e,l);

Solution 5.2
fun del(e,[]) = [] |
  del(e,x::l) = if e=x then l else x::del(e,l);

Solution 5.3
fun dell([],l2) = l2 |
  dell(e::l1,l2) = dell(l1,del(e,l2));

Solution 5.4
fun conc(xs,ys) = foldr op:: ys xs;
Let \(xs = [a, b, c]\). According to the definition of function foldr, function conc(xs,ys) evaluates
to \(op :: (a, (op :: (b, (op :: (c, ys))))) = a :: (b :: (c :: ys))\). That means, list \(xs\) is unfolded and its
elements are consecutively in reversed order concatenated with \(ys\).

Solution 5.5
fun reverse(xs) = foldl op:: [] xs;
Let \(xs = [a, b, c]\). According to the definition of function foldl, function reverse(xs) evaluates
to \(op :: (c, (op :: (b, (op :: (a, []))))) = c :: (b :: (a :: []))\).

Solution 5.6
fun exists(p,[]) = false |
  exists(p,(x::xs)) = p(x) orelse exists(p,xs);
The base is the empty list, and the inductive step is the nonempty list. For an empty list,
predicate \(p\) evaluates to false, hence the function returns false. In the inductive step, the function
returns true if the head of the list satisfies predicate \(p\); otherwise, the function exists is applied to
the tail of the list.

Solution 5.7
fun filter(p,[]) = [] |
  filter(p,(x::xs)) = if p(x) then x::filter(p,xs) else filter(p,xs);
The base is the empty list, and the inductive step is the nonempty list. For an empty list,
the function returns the empty list, because there is no element that satisfies predicate \(p\). In the
inductive step, the head of the list is checked. If it satisfies predicate \(p\), then it is concatenated to
a list of elements collected from the tail; otherwise, the function filter is applied to the tail of the
list.

Solution 5.8
fun len([],) = 0 |
  len(x::l) = 1 + len(l);

Solution 5.9
fun dropn([],n) = [] |
  dropn(l,0) = l |
  dropn(x::l,n) = dropn(l,n-1);

Solution 5.10
fun returnn([],n) = [] |
  returnn(l,0) = [] |
  returnn(x::l,n) = x::returnn(l,n-1);

Solution 5.11
map #1 prodList
Solution 5.12

This solution is just an example. There are many possible solutions to any given modeling exercise.

Figure 85 depicts the CPN model. There are three kinds of places: places modeling the position of a CityCat are of type \textit{CC}, places modeling whether a stop is free are of type unit, and places modeling the FCFS queues are of type \textit{CClist}. The network structure of this model extends the one of the Petri net model in Figure 32 by adding six places for modeling the FCFS queues. Place \textit{BtoAL} models the downstream queue between stops \textit{B} and \textit{A} and place \textit{AtoBL} the upstream queue between these stops. Likewise, places \textit{CtoBL}, \textit{BtoCL}, \textit{CtoDL}, and \textit{DtoCL} model the respective
queues between stops $B$ and $C$ and between $C$ and $D$. A queue is modeled as a list of CityCats, hence ensuring that CityCats cannot overtake each other (see the third requirement). To distinguish individual CityCats (see the first requirement), each CityCat is identified by a number. Function $d(c)$ implements the time CityCats need to move from one stop to the next. The time is as a delay assigned to the respective CityCat token when moving from one stop to the next one (see the second requirement). Transition from $D$ to $C$ can fire if the corresponding downstream queue $B$ to $C$ is empty, and transition from $C$ to $D$ can fire if place $A$ to $B$ contains the empty list. That means, all CityCats moving upstream have to pass before a CityCat moving downstream can continue its service (see the fourth requirement). Finally, when a CityCat moves to a stop, a delay of five time units is assigned to it. This models the time necessary to let passengers embark or disembark (see the fifth requirement). In the initial marking, expression $CC.all()$ generates the ten CityCats in place $X$, each of the “free” places (e.g., $A.free$) contains a token indicating that no CityCat docks at any of the four stops, and the queues are empty.

**Solution 5.13**

<table>
<thead>
<tr>
<th>This solution is just an example. There are many possible solutions to any given modeling exercise.</th>
</tr>
</thead>
</table>

1. Figure 86 depicts the CPN model.

![Figure 86: A CPN model of Exercise 5.13(1).](image)

2. Figure 87 depicts the CPN model.

3. Figures 88 and 89 depict two possible solutions.

4. Figure 90 depicts the CPN model.
Clearly indicate the color sets and arc inscriptions. (0.50 points)

c) Assume that the inhibitor arc is replaced by a reset arc connecting t3 and p. The behaviorally equivalent classical Petri net fragment. (0.50 points)

Figure 87: A CPN model of Exercise 5.13(2).

Figure 88: A CPN model of Exercise 5.13(3).

Figure 89: An alternative CPN model of Exercise 5.13(3).

Figure 90: A CPN model of Exercise 5.13(4).
6 Petri nets standard properties

Exercise 6.1 Draw for the Petri net systems in Figures 91–96 the reachability graph and analyze whether they are bounded, terminating, live, deadlock free, reversible, and have dead transitions or home-markings.

Figure 91: Petri net systems for analyzing standard properties.

Figure 92: Petri net systems for analyzing standard properties.
Figure 93: Petri net systems for analyzing standard properties.

Figure 94: Petri net systems for analyzing standard properties.

Figure 95: Bank contract.
Figure 96: Simplified Crosstalk algorithm.
Solutions

Solution 6.1

91(a) 1. The reachability graph is depicted in Figure 97.

Figure 97: Reachability graph of Figure 91(a).

2. The net is 2-bounded; see the reachability graph.
3. The net is terminating; there is only one run and this run is finite.
4. The net has a deadlock: \([\text{free}, 2 \cdot \text{done}]\).
5. The net has no dead transitions because \((\text{start}, \text{stop})\) is a run.
6. The net is not live because it has a deadlock.
7. Deadlock \([\text{free}, 2 \cdot \text{done}]\) is a home-marking.
8. The net is not reversible because it has a deadlock.

91(b) 1. The reachability graph is infinite.
2. There is an infinite run \(\langle t_1, t_4, t_5, t_1, t_4, t_5, \ldots \rangle\) that witnesses that place \(p_9\) and hence the net is unbounded.
3. The net has an infinite run (see (2)) and is hence not terminating.
4. The net has no deadlocks. Always one of the transitions \(t_1, t_2, t_3, t_4,\) or \(t_5\) is enabled (as there is always one token in place \(p_1, p_2, p_3,\) or \(p_4\)).
5. The net has no dead transitions because \((t_1, t_2, t_3, t_4, t_5)\) is a run.
6. The net is live, because it is reversible and any transition can be enabled from the initial marking.
7. The initial marking is a home-marking.
8. There is always one token in one of the places \(\{p_1, p_2, p_3, p_4\}\) and one token in one of the places \(\{p_5, p_6, p_7, p_8\}\) (see also place invariants). The number of tokens in place \(p_7\) is always equal to the total number of tokens in places \(p_10, p_3,\) and \(p_11\). Hence, tokens can only accumulate in place \(p_9\). Assume that there are \(k\) tokens accumulated in place \(p_9\). These tokens can be removed to return to the initial marking. As long as there are tokens in place \(p_9\), the transitions \(t_6, t_9, t_{10}\) can be executed, removing the \(k\) tokens one by one. These insights show that the net can indeed return to the initial marking from any reachable marking. Hence, the net is reversible.

92(a) 1. The reachability graph is infinite.
2. There is an infinite run \(\langle t_1, t_2, t_1, t_2, \ldots \rangle\) that witnesses that place \(p_3\) and hence the net is unbounded.
3. The net has an infinite run (see (2)) and hence is not terminating.
4. The net can reach a deadlock \([]\) when executing the run \(\langle t_1, t_2, t_1, t_3 \rangle\).
5. The net has no dead transitions because \((t_1, t_2, t_1, t_3)\) is a run.
6. The net is not live because it has a deadlock.
7. The net does not have a home-marking.
8. The net is not reversible because it has a deadlock.

92(b) 1. The reachability graph is depicted in Figure 98.
2. The net is safe (can be proved using place invariants).
3. The net has an infinite run \( \langle ta_1, pd_1, ta_1, pd_1, \ldots \rangle \) and hence is not terminating.
4. The net is deadlock free; each of the places \( f_1, \ldots, f_4 \) contains a token, thereby enabling transitions \( ta_1, \ldots, ta_4 \) or at least one of the places \( e_1, \ldots, e_4 \) contains a token, enabling the respective \( pd \) transition.
5. The net has no dead transitions because \( \langle ta_1, pd_1, ta_2, pd_2, ta_3, pd_3, ta_4, pd_4 \rangle \) is a run.
6. The net is live, because it is reversible and any transition can be enabled from the initial marking.
7. Every marking is a home-marking.
8. The net is reversible. There are only two types of transitions: \( ta \) and \( pd \). Whenever a \( ta \) transition has fired, the respective \( pd \) transition is enabled. Hence, as the net is deadlock free, it is always possible to return to the initial marking.

93(a) 1. The reachability graph is depicted in Figure 99.
2. The net is safe.
3. The net has an infinite run \( \langle t_1, t_3, t_2, t_3, t_2, \ldots \rangle \) and hence is not terminating.
4. The net is deadlock free.
5. The net has no dead transitions because \( \langle t_1, t_3, t_2, t_5, t_4, t_6 \rangle \) is a run.
6. The net is live.
7. Each marking is a home-marking.
8. The net is reversible.

93(b) 1. The reachability graph is infinite.
2. There is an infinite run \( \langle t_1, t_2, t_1, t_2, \ldots \rangle \) that witnesses that place \( p_3 \) and hence the net is unbounded.
3. The net has an infinite run (see (2)) and hence is not terminating.
4. The net is deadlock free, because either \( p_1 \) or \( p_2 \) contain always a token.
5. The net has no dead transitions because \( \langle t_1, t_2, t_4, t_3 \rangle \) is a run.

6. The net is live.

7. Every reachable marking is a home-marking.

8. The net is reversible.

94(a) 1. The reachability graph is depicted in Figure 100.

![Figure 100: Reachability graph of Figure 94(a).](image)

2. The net is safe.

3. The net has an infinite run \( \langle t_1, t_6, t_9, t_4, t_5, t_{10}, t_1, t_6, t_7, t_2, t_3, t_8, t_5, t_{10}, \ldots \rangle \) and hence is not terminating.

4. The net is deadlock free.

5. The net has no dead transitions because \( \langle t_1, t_6, t_9, t_4, t_5, t_{10}, t_1, t_6, t_7, t_2, t_3, t_8, t_5, t_{10}, \ldots \rangle \) is a run.

6. The net is live because there are no dead transitions and the net is reversible.

7. The initial marking is a home-marking.

8. The net is reversible.

94(b) 1. The reachability graph is too big to be depicted.

2. The net is 30-bounded (can be concluded from the P-invariants).

3. The net has an infinite run \( \langle t_5, t_6, t_7, t_5, t_6, t_7, \ldots \rangle \) and hence is not terminating.

4. The net is deadlock free. The right part of the model cannot remove any tokens of left part. Therefore, we can focus on the left part. Place \( p_8 \) always has the same number of tokens as \( p_9 \). Therefore, we can remove \( p_9 \) (implicit place). The total number of tokens in \( p_5 + p_7 + p_8 \) is always 30 and the number of tokens in \( p_6 + p_7 \) is always 20. Hence, at any point in time at least one of the transitions \( t_5, t_6, t_7 \) is enabled.

5. The net has no dead transitions because \( \langle t_5, t_1, t_2, t_1, t_3, t_4, t_6, t_7 \rangle \) is a run.

6. The net is not live because \( t_3 \) and \( t_4 \) can only be fired once.
7. Marking \([p_4, 30 \cdot p_5, 20 \cdot p_6]\) is home-marking—\(p_4\) is a deadlock for the right part and the left part is reversible.

8. The net is not reversible: once \(t_3\) fired, a token cannot produced on \(p_1\) or \(p_2\).

1. The reachability graph is depicted in Figure 101.

2. The net is safe. For all places except from \(fwd\) and \(inf\) this can be concluded from the \(P\)-invariants. If there is a token in place \(fwd\), then a marking \([p_2]\) can be reached in the Client. To produce a second token in \(fwd\), \(t_2\) must be fired, but to this end, transition \(t_5\) must be fired, consuming the token from \(fwd\). Similar, if there is a token in place \(inf\), then \(t_5\) and \(t_6\) must be fired to produce a second token in \(inf\). This, however, requires a token in \(fwd\). By the previous argumentation, we conclude that to produce a token in \(fwd\), transition \(t_2\) must fire, consuming the token from \(inf\).

3. The net has an infinite run \(\langle t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_8, t_{10}, t_{12}, \ldots \rangle\) and hence is not terminating.

4. The net has a deadlock: The run \(\langle t_0, t_3, t_4, t_1, t_5, t_7, t_8 \rangle\) results in a marking \([p_2, p_4]\) which enables no transition.

5. The net has no dead transitions because every transition appears in a \(T\)-invariant which is enabled at the initial marking.

6. The net is not live because it can deadlock.

7. The deadlock is a home-marking.

8. The net is not reversible because it can deadlock.

1. The reachability graph is depicted in Figure 102.

2. The net is safe (can be concluded from the \(P\)-invariant).

3. The net has an infinite run \(\langle SendLeft, RespondRight, FinishLeft, DoneRight \rangle\) and hence is not terminating.

4. The net has a deadlock: Firing \(SendLeft\) and \(SendRight\) leads to a deadlock \([p_2, p_5, p_6, p_{10}]\).

5. The net has no dead transitions because every transition appears in a \(T\)-invariant which is enabled at the initial marking.
6. The net is not live because it can deadlock.

7. The deadlock is a home-marking.

8. The net is not reversible because it has a deadlock.
7 Coverability graphs and fairness

Exercise 7.1 Consider the Petri nets in Figures 91–96 and Figure 103.

1. Construct the coverability graph.
2. Does every path in the coverability graph correspond to a run?
3. Which properties can be derived from the coverability graph?
4. Determine for each transition whether it is impartial, fair, just, or satisfies no fairness property.
Solutions

Solution 7.1

91(a) 1. The coverability graph and the reachability graph (see Figure 97) are equivalent.
2. Every path in the coverability graph trivially corresponds to a run in the net.
3. Because reachability and coverability graph are equivalent, we can derive all standard properties from it.
4. The net has no infinite run; thus all transitions are trivially impartial.

91(b) 1. Generate!
2. The path \( \langle t_1, t_4, t_5, t_1, t_6, t_9, t_{10}, t_6, t_9, t_{10}, \ldots \rangle \) in the coverability graph does not correspond to a run.
3. We can derive that place \( p_9 \) is unbounded and all other places are safe. In addition, no transition is dead because every transition appears in the coverability graph.
4. As indicated before, there is an infinite run \( \langle t_1, t_4, t_5, \ldots \rangle \). For each transition in the right part of the net, there is a marking that enables this transition continuously in this infinite run. For example, marking \([p_4, p_1, p_7]\) is reachable and enables transition \( t_8 \). Hence, all five transitions \( t_6, t_7, t_8, t_9, \) and \( t_{10} \) are unfair. It is easy to see that cannot be an infinite firing sequence without infinitely many executions of transitions \( t_1 \) and \( t_5 \). For this reason, both transitions are impartial. The remaining three transitions (i.e., \( t_2, t_3, \) and \( t_4 \)) do not need to occur infinitely often. Transition \( t_3 \) occurs infinitely often in every infinite run where it is enabled infinitely often. Hence, transition \( t_3 \) is fair. Transitions \( t_2 \) and \( t_4 \) are not fair, because, even if they are enabled infinitely often, they do not need to occur infinitely often. But these two transitions are just; that is, it is not possible to continuously enable them in any infinite run.

impartial: \( t_1, t_5 \)

fair: \( t_3 \)

just: \( t_2, t_4 \)

unfair: \( t_6, t_7, t_8, t_9, t_{10} \)

92(a) 1. The coverability graph is depicted in Figure 1.

2. Yes, every path corresponds to a run.
3. We derive that \( p_3 \) is unbounded and all other places are safe and that the net has no dead transitions because every transition appears in the coverability graph.
4. There is a single infinite run \( \langle t_1, t_2, t_1, t_2, \ldots \rangle \). Transitions \( t_1 \) and \( t_2 \) are impartial, because they occur infinitely often. In contrast, transition \( t_3 \) is trivially just, because there is no infinite run where this transition is continuously enabled from some marking onward (it is not fair, because it is enabled infinitely often).

92(b) 1. The coverability graph and the reachability graph in Figure 98 are equivalent.
2. Every path in the coverability graph trivially corresponds to a run in the net.
3. Because reachability and coverability graph are equivalent, we can derive all standard properties from it.
4. All transitions are unfair, because for any transition there exists an infinite run where this transition is continuously enabled but does not fire; for example, \( \{ta3, pd3, ta3, \ldots\} \) shows that \( ta1 \) is unfair and \( \{ta1, ta3, pd3, ta3, \ldots\} \) shows that \( pd1 \) is unfair.

93(a) 1. The coverability graph and the reachability graph in Figure 99 are equivalent.

2. Every path in the coverability graph trivially corresponds to a run in the net.

3. Because reachability and coverability graph are equivalent, we can derive all standard properties from it.

4. There exists an infinite run \( \langle t1, t3, t2, t3, t2, \ldots \rangle \) where transitions \( t1, t4, t5, \) and \( t6 \) do not occur infinitely often. So we conclude that transitions \( t1, t4, t5, \) and \( t6 \) are not impartial. Similar, existence of the infinite run \( \langle t1, t5, t4, t5, t4, \ldots \rangle \) proves that transitions \( t2 \) and \( t3 \) are not impartial. If transition \( t1 \) is enabled, it always fires. Consequently, if transition \( t1 \) is infinitely often enabled, then it fires infinitely often. So transition \( t1 \) is fair. Transition \( t6 \) is not fair, because in the infinite run \( \langle t1, t5, t3, t2, t3, t2, \ldots \rangle \) it is infinitely often enabled but does not fire. However, it is just; that is, in every infinite run where transition \( t6 \) is continuously enabled from a marking onward, it fires infinitely often. The infinite run \( \langle t1, t3, t5, t4, t5, t4, \ldots \rangle \) witnesses that transition \( t2 \) is neither fair nor just. Consequently, it is not fair. The same holds for transitions \( t3, t4, \) and \( t5. \)

93(b) 1. Figure 105 depicts the coverability graph. There exist more than one coverability graph. To calculate Figure 105, we fired transition \( t1, t4, \) and \( t2 \) first.

![Figure 105: Coverability graph of Figure 93(b).](image)

2. \( \langle t1, t2, t4, t3, t4, t3 \rangle \) is a path in the coverability graph which does not correspond to a run in the net.

3. The net does not have dead transitions because all transitions appear in the coverability graph. Place \( p3 \) is unbounded, and all other places are safe.

4. Transitions \( t1 \) and \( t2 \) are impartial, because in every infinite run they are fired infinitely often. Existence of the infinite runs \( \langle t1, t2, t1, t2, \ldots \rangle \) and \( \langle t5, t1, t2, t1, t2, \ldots \rangle \) proves that transitions \( t3 \) and \( t4 \) are not fair, because they are continuously enabled but do not fire.

94(a) 1. The coverability graph and the reachability graph in Figure 100 are equivalent.

2. Every path in the coverability graph trivially corresponds to a run in the net.

3. Because reachability and coverability graph are equivalent, we can derive all standard properties from it.

4. Transitions \( t1, t5, t6, t10 \) because they occur in every infinite run infinitely often. Existence of the infinite runs \( \langle t1, t6, t7, t3, t8, t5, t10, t1 \ldots \rangle \) and \( \langle t1, t6, t9, t4, t5, t10, t1 \ldots \rangle \) proves that transitions \( t9, t4 \) and \( t7, t3, t8 \) are not impartial. However, \( t2, t3, t4, t8 \) are fair, because whenever they are enabled they will be eventually fired. The two previous infinite runs also prove that \( t7 \) and \( t9 \) are not fair. Because they cannot be continuously
enabled in any infinite run, they are trivially just.

impartial: \( t_1, t_5, t_6, t_{10} \)

fair: \( t_2, t_3, t_4, t_8 \)

just: \( t_7, t_9 \)

94(b) 1. The coverability graph and the reachability graph are equivalent.
2. Every path in the coverability graph trivially corresponds to a run in the net.
3. Because reachability and coverability graph are equivalent, we can derive all standard properties from it.
4. Existence of infinite runs \( \langle t_5, t_1, t_2, t_1, t_2, \ldots \rangle \) and \( \langle t_5, t_6, t_7, t_5, t_6, t_7, \ldots \rangle \) shows that no transition is impartial. From these two runs, we can derive for each transition an infinite run where this transition is continuously enabled but does not fire infinitely often. Hence, all transitions are unfair.

95 1. The coverability graph and the reachability graph in Figure 101 are equivalent.
2. Every path in the coverability graph trivially corresponds to a run in the net.
3. Because reachability and coverability graph are equivalent, we can derive all standard properties from it.
4. Transitions \( t_0, \ldots, t_6, t_8 \) are impartial because they occur on every infinite run infinitely often. Transition \( t_7 \) is trivially just, because it is never continuously enabled; it is not fair because it is enabled infinitely often in every infinite run but never occurs.

96 1. The coverability graph and the reachability graph in Figure 102 are equivalent.
2. Every path in the coverability graph trivially corresponds to a run in the net.
3. Because reachability and coverability graph are equivalent, we can derive all standard properties from it.
4. Existence of infinite runs \( \langle SR, RL, DL, FR, SR, \ldots \rangle \) and \( \langle SL, RR, DR, FL, SL, \ldots \rangle \) shows that no transition is impartial. From these runs, we determine that \( SL, RR, RL, SR \) are not fair but trivially just because they are never continuously enabled. All other transitions are fair: If \( SL \) has fired (and thus there is a token in \( p_2 \)), then \( RR \) has to fire eventually on every infinite run (otherwise, the net would deadlock) and afterward \( FL \) and \( DR \) fire eventually. Symmetrically, this argument holds for transitions \( DL \) and \( FR \).

fair: \( FL, DR, DL, FR \)

just: \( SL, RR, RL, SR \)

103(a) 1. The coverability graph is depicted in Figure 1.

![Coverability graph of the Petri net in Figure 103(a).](image)

2. \( \langle t_1, t_3, t_4, t_4 \rangle \) is a path in the coverability graph which does not correspond to a run in the net.
3. The net does not have dead transitions, because all transitions appear in the coverability graph. Places \( p_2 \) and \( p_3 \) are unbounded, and all other places are safe.
4. Transitions $t_1$ and $t_2$ are trivially fair; they are not impartial, because they do not occur infinitely often in any infinite run. Transition $t_3$ is impartial. Transition $t_4$ is unfair; the infinite run $\langle t_1, t_3, t_3, \ldots \rangle$ shows that this transition is continuously enabled but does not fire.

**103(b)**  
1. The coverability graph is depicted in Figure 1.

![Figure 107: Coverability graph of the Petri net in Figure 103(b).](image)

2. $\langle t_1, t_2, t_3, t_4, t_4 \rangle$ is a path in the coverability graph which does not correspond to a run in the net.

3. The net does not have dead transitions, because all transitions appear in the coverability graph. Place $p_4$ is unbounded, and all other places are safe.

4. Transitions $t_1$, $t_2$, and $t_3$ are impartial, because it is not possible to construct an infinite run where not all these transitions appear infinitely often. If we stop executing one of these transitions, the net deadlocks after a while. Transition $t_4$ has no fairness, because the infinite run $\langle t_1, t_2, t_3, t_1, t_2, t_3, \ldots \rangle$ witnesses that $t_4$ remains enabled but never fires.

**103(c)**  
1. The coverability graph is depicted in Figure 1.

![Figure 108: Coverability graph of the Petri net in Figure 103(c).](image)

2. In this case, all paths in the coverability graph correspond to a run of the net, because the number of tokens in place $p_4$ is increasing in each cycle, and no transition is ever blocked because of the number of tokens in this place.

3. The net does not have dead transitions, because all transitions appear in the coverability graph. Places $p_2$, $p_3$, $p_4$ are unbounded, and place $p_1$ is safe.

4. Transitions $t_1$ and $t_2$ are impartial, because it is not possible to construct an infinite run where not all these transitions appear infinitely often. If we stop executing one of these transitions, the net deadlocks after a while. Transitions $t_3$ and $t_4$ are not impartial or fair, because the infinite runs $\langle t_1, t_2, t_3, t_1, t_2, t_3, \ldots \rangle$ and $\langle t_1, t_2, t_4, t_1, t_2, t_4, \ldots \rangle$ witness that $t_4$ and $t_3$ do not need to fire infinitely often even when they are enabled infinitely often. As there is no infinite run that enables transitions $t_4$ and $t_3$ continuously, they are just (one will disable the other one).

**103(d)**  
1. The coverability graph is depicted in Figure 1.

2. $\langle t_1, t_2, t_3, t_4 \rangle$ is a path in the coverability graph which does not correspond to a run in the net.

3. The net does not have dead transitions, because all transitions appear in the coverability graph. Places $p_2$, $p_3$, $p_4$ are unbounded, and place $p_1$ is safe.
4. Transition $t_1$ is trivially fair; it is not impartial, because it does not occur infinitely often in any infinite run. Transition $t_2$ is impartial. Transitions $t_3$ and $t_4$ are unfair; the infinite run $\langle t_1, t_2, t_2, \ldots \rangle$ shows that both transitions are continuously enabled but do not fire.

103(e) 1. The coverability graph is depicted in Figure 1.

2. $\langle t_0, t_2, t_4, t_3, t_4, t_3, \ldots \rangle$ is a path in the coverability graph which does not correspond to a run in the net.

3. The net does not have dead transitions, because all transitions appear in the coverability graph. Place $p_3$ is unbounded, places $p_1, p_2$ are 2-bounded, and $p_0$ is safe.

4. Transition $t_0$ is trivially fair; it is not impartial, because it does not occur infinitely often in any infinite run. Transition $t_2$ is impartial. Transitions $t_1, t_3, and t_4$ are unfair; the infinite run $\langle t_0, t_2, t_2, \ldots \rangle$ shows that both transitions are continuously enabled but do not fire.

103(f) 1. The coverability graph is depicted in Figure 1.

2. $\langle t_0, t_2, t_4, t_3, t_4, t_4, \ldots \rangle$ is a path in the coverability graph which does not correspond to a run in the net.

3. The net does not have dead transitions, because all transitions appear in the coverability graph. Places $p_1, p_2, p_3$ are unbounded, and place $p_0$ is safe.

4. Transition $t_0$ is trivially fair; it is not impartial, because it does not occur infinitely often in any infinite run. Transitions $t_1, t_2, t_3,$ and $t_4$ are unfair; the following infinite runs show that each transition is continuously enabled but does not fire:

$t_1, t_4:$ $\langle t_0, t_2, t_4, t_3, t_3, t_3, \ldots \rangle$
$t_2:$ $\langle t_0, t_2, t_4, t_3, t_4, t_1, t_3, t_3, \ldots \rangle$
$t_3:$ $\langle t_0, t_2, t_4, t_1, t_2, t_4, t_1, t_2, \ldots \rangle$
103(g) 1. The coverability graph is depicted in Figure 1.

2. \( (t_2, t_3, t_4, t_1, t_2, t_3, t_1, t_2, t_3, \ldots) \) is a path in the coverability graph which does not correspond to a run in the net.

3. The net does not have dead transitions, because all transitions appear in the coverability graph. Place \( p_3 \) is unbounded, and all other places are safe.

4. Transition \( t_1 \) is unfair; the infinite run \( (t_2, t_3, t_4, t_4, \ldots) \) shows that \( t_1 \) is continuously enabled but does not fire. Transitions \( t_2 \) and \( t_3 \) are fair; they are not impartial, because they do not occur infinitely often on the previous infinite run. Transition \( t_4 \) is impartial, because it is impossible to continue to fire without firing \( t_4 \).

103(h) 1. The coverability graph is depicted in Figure 1.

2. \( (t_1, t_4, t_3) \) is a path in the coverability graph which does not correspond to a run in the net.

3. The net does not have dead transitions, because all transitions appear in the coverability graph. Place \( p_3 \) is unbounded, and all other places are safe.

4. In any infinite firing sequence \( t_1 \) needs to happen infinitely often. If \( t_1 \) stops firing after a while, the process will end. Hence, \( t_1 \) is impartial as it occurs infinitely often in every infinite firing sequence. All other transitions have no fairness (i.e., are not just). Consider the infinite firing sequence \( (t_1, t_1, t_1, \ldots) \). After firing \( t_1 \) three times,
all transitions are enabled and remain enabled in this particular infinite firing sequence. However, none of the other transitions needs to fire despite being enabled continuously.
8 Place- and transition invariants

Exercise 8.1 Determine place and transition invariants for the Petri nets in Figures 91–96, 103 and 114.

Exercise 8.2 We cannot prove reachability of a marking \( m \) using the incidence matrix; even for an unreachable marking \( m \), the state equation can have an integer solution. Show for the Petri net in Figure 114 that the unreachable marking \( m = [p_2,p_3] \) has an integer solution for \( \vec{x} \) in

\[
C \cdot \vec{x} = \vec{m} - \vec{m}_0.
\]

Note that \( \vec{m}_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \) and \( \vec{m} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \).

Exercise 8.3 Give a counterexample to disprove the following statements:

1. If \( i \cdot m = i \cdot m_0 \), for a place invariant \( i \) and markings \( m_0 \) and \( m \), then \( m \) is reachable from \( m_0 \).
2. If a place is bounded, then it occurs in a positive place invariant.
3. If a net has a positive transition invariant that covers each transition, then it is live and bounded.

Solutions

Solution 8.1

91(a) The base P-invariants are:

\[
done + \text{wait} + \text{busy} = 2 \\
\text{free} + \text{busy} = 1
\]

- There are no T-invariants, because the net is terminating.

91(b) The base P-invariants are:

\[
p1 + p2 + p3 + p4 = 1 \\
p5 + p6 + p7 + p8 = 1 \\
p3 + p5 + p6 + p8 + p10 + p11 = 1
\]

- T-invariants are:

\[
t1 + t4 + t5 + t6 + t9 + t10 \\
t1 + t2 + t3 + t5 + t6 + t7 + t8 + t10
\]

92(a) There are no P-invariants.

- There are no T-invariants.
92(b)  The base P-invariants are:
\[
\begin{align*}
e_1 + t_1 &= 1 \\
e_2 + t_2 &= 1 \\
e_3 + t_3 &= 1 \\
e_4 + t_4 &= 1 \\
e_1 + e_2 + f_1 &= 1 \\
e_2 + e_3 + f_2 &= 1 \\
e_3 + e_4 + f_3 &= 1 \\
e_4 + e_1 + f_4 &= 1 \\
\end{align*}
\]

- T-invariants are:
\[
\begin{align*}
ta_1 + pd_1 \\
ta_2 + pd_2 \\
ta_3 + pd_3 \\
ta_4 + pd_4 \\
\end{align*}
\]

93(a)  The base P-invariants are:
\[
\begin{align*}
p_1 + p_2 + p_4 &= 1 \\
p_1 + p_3 + p_5 &= 1 \\
\end{align*}
\]

- T-invariants are:
\[
\begin{align*}
t_1 + t_3 + t_5 + t_6 \\
t_2 + t_3 \\
t_4 + t_5 \\
\end{align*}
\]

93(b)  The base P-invariants are:
\[
\begin{align*}
p_1 + p_2 &= 1 \\
p_4 + p_5 &= 1 \\
\end{align*}
\]

- T-invariants are: \( t_1 + t_2 + t_3 + t_4 \).

94(a)  The base P-invariants are:
\[
\begin{align*}
p_1 + p_2 + p_3 + p_4 &= 1 \\
p_5 + p_6 + p_7 + p_8 &= 1 \\
p_1 + p_3 + p_4 + p_6 + p_9 + p_{10} + p_{11} &= 1 \\
p_3 + p_5 + p_6 + p_8 + p_{10} + p_{12} &= 1 \\
\end{align*}
\]

- T-invariants are:
\[
\begin{align*}
t_1 + t_2 + t_3 + t_5 + t_6 + t_7 + t_8 + t_{10} \\
t_1 + t_4 + t_5 + t_6 + t_9 + t_{10} \\
\end{align*}
\]

94(b)  The base P-invariants are:
\[
\begin{align*}
p_6 + p_7 &= 20 \\
p_1 + p_2 + p_3 + p_4 &= 1 \\
p_5 + p_7 + p_8 &= 30 \\
p_5 + p_7 + p_9 &= 30 \\
\end{align*}
\]

- T-invariants are:
\[
\begin{align*}
t_5 + t_6 + t_7 \\
t_1 + t_2 \\
\end{align*}
\]

95  The base P-invariants are:
\[
\begin{align*}
p_0 + p_1 + p_2 &= 1 \\
p_4 + p_5 + p_6 &= 1 \\
p_0 + p_2 + req + p_3 + bill &= 1 \\
\end{align*}
\]

- T-invariants are: \( t_0 + t_1 + t_2 + t_3 + t_4 + t_5 + t_6 + t_8 \).
96  - The base P-invariants are:
    \[ p_1 + p_2 + p_3 = 1 \]
    \[ p_8 + p_9 + p_{10} = 1 \]
    \[ p_1 + p_3 + p_4 + p_5 = 1 \]
    \[ p_6 + p_7 + p_8 + p_9 = 1 \]
- T-invariants are:
  \[ SendLeft + RespondRight + FinishLeft + DoneRight \]
  \[ SendRight + RespondLeft + FinishRight + DoneLeft \]

103(a)  - There are no P-invariants.
- There are no T-invariants.

103(b)  - The base P-invariant is: \( p_1 + p_2 + p_3 = 1 \).
- T-invariants are: \( t_1 + t_2 + t_3 + t_4 \).

103(c)  - The base P-invariant is: \( p_1 + p_2 + p_3 = 1 \).
- There are no T-invariants.

103(d)  - There are no P-invariants.
- There are no T-invariants.

103(e)  - There are no P-invariants.
- T-invariants are: \( 2 \cdot t_2 + t_3 + t_4 \).

103(f)  - There are no P-invariants.
- T-invariants are: \( t_1 + t_2 + t_4 \).

103(g)  - The base P-invariants are:
    \[ p_1 + p_3 = 1 \]
    \[ p_2 + p_4 = 1 \]
- T-invariants are: \( t_1 + t_2 + t_3 + t_4 \).

103(h)  - The base P-invariants are:
    \[ p_1 + p_2 = 1 \]
    \[ p_4 + p_5 = 1 \]
- T-invariants are:
  \[ t_1 + t_2 \]
  \[ 3 \cdot t_1 + t_3 \]

114  - The base P-invariants are:
    \[ p_1 + p_2 + p_4 = 1 \]
    \[ p_1 + p_3 + p_5 = 1 \]
- T-invariants are:
  \[ t_1 + t_3 + t_5 \]
  \[ t_2 + t_4 + t_5 \]

**Solution 8.2**

We obtain \( C \cdot \vec{x} = \vec{m} - \vec{m}_0 \)

\[
\begin{pmatrix}
-1 & -1 & 0 & 0 & 1 \\
1 & 0 & -1 & 0 & 0 \\
0 & 1 & 0 & -1 & 0 \\
0 & 1 & 1 & 0 & -1 \\
1 & 0 & 0 & 1 & -1 \\
\end{pmatrix}
\begin{pmatrix}
p_1 \\
p_2 \\
p_3 \\
p_4 \\
p_5 \\
\end{pmatrix}
= \begin{pmatrix}
0 \\
1 \\
1 \\
0 \\
0 \\
\end{pmatrix}
- \begin{pmatrix}
1 \\
0 \\
0 \\
0 \\
0 \\
\end{pmatrix}
\]
The resulting equation system has the integer solution \[
\begin{pmatrix}
p_1 \\
p_2 \\
p_3 \\
p_4 \\
p_5
\end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}.
\]

Solution 8.3

1. The statement “if \( i \cdot m = i \cdot m_0 \), for a place invariant \( i \) and markings \( m_0 \) and \( m \), then \( m \) is reachable from \( m_0 \)” (i.e., the converse) does not hold. Consider the place invariant \( \text{man} - \text{woman} = -1 \) for the net Figure 115. For the initial marking \( m_0 \) we calculate \( 2 - 3 = -1 \), and for the nonreachable marking \([\text{man}, \text{couple}, 2 \cdot \text{woman}]\) we also calculate \( 1 - 2 = -1 \).

![Figure 115: A Petri net.](image)

2. The statement “if a place is bounded, then it occurs in a positive place invariant” (i.e., the converse) does not hold. It is trivial to provide a net with a dead initial marking and no positive place invariant. Figure 116 provides a more interesting counterexample. The net is live and bounded if \( m_0 = [p_1] \), but there is no positive place invariant. Moreover, if \( m_0 = [2 \cdot p_1] \), then the net suddenly loses all of its nice properties. For example, place \( p_3 \) is unbounded and the net may deadlock. This shows that even though the structure did not change, properties such as boundedness may change when the initial marking changes.

![Figure 116: A bounded Petri net without positive place invariant.](image)

3. The statement “if the net has a positive transition invariant that covers each transition, then it is live and bounded” (i.e., the converse) does not hold. Figure 117 provides a counterexample. A positive transition invariant is \( t1 + t2 \), but the net is not live. The problem is that transition invariants are independent of the initial marking.

![Figure 117: Although all transitions are covered by a transition invariant, the net is not live.](image)