

C.A. Middelburg and M.A. Reniers

Introduction to Process Theory

September 6, 2005

Technische Universiteit Eindhoven

Faculteit Wiskunde en Informatica

Capaciteitsgroep Informatica

Eindhoven

Contents

1. Transition systems	3
1.1 Informal explanation	3
1.2 Formal definition	9
1.3 Programs and transition systems	12
1.4 Automata and transition systems	18
1.5 Equivalences on processes	22
1.6 Hennessy-Milner logic	40
2. Concurrency and Interaction	51
2.1 Informal explanation	51
2.2 Formal definitions	55
3. Composition	65
3.1 Informal explanation	65
3.2 Formal definition of the compositions	67
A. Set theoretical preliminaries	81
A.1 Sets	81
A.2 Relations and functions	83
A.3 Sequences	84

1. Transition systems

The notion of transition system can be considered to be the fundamental notion for the description of process behaviour. This chapter is meant to acquire a good insight into this notion and its relevance for the description of process behaviour. First of all, we explain informally what transition systems are and give some simple examples of their use in describing process behaviour (Section 1.1). After that, we define the notion of transition system in a mathematically precise way (Section 1.2). For a better understanding, we next investigate the connections between the notion of transition system and the familiar notions of program (Section 1.3) and automaton (Section 1.4). Then, we discuss several notions of equivalence on transition systems (Section 1.5). Those equivalences are useful because they allow us to abstract from details of transition systems that we often want to ignore. Finally, we look at a means of expressing properties of processes in terms of a logic called Hennessy-Milner logic (Section 1.6).

1.1 Informal explanation

Transition systems are often considered to be the same as automata. Both consist of states and labeled transitions between states. The main difference is that automata are primarily regarded as abstract machines to recognize certain languages and transition systems are primarily regarded as a means to describe the behaviour of interacting processes. In the case of transition systems, the intuition is that a transition is a state change caused by performing the action labeling the transition. A transition from a state s to a state s' labeled by an action a is usually written $s \xrightarrow{a} s'$. This can be read as “the system is capable of changing its state from s into s' by performing action a ”. Let us give an example to illustrate that it is quite natural to look at real-life computer-based systems as systems that change their state by performing actions.

Example 1.1.1 (Simple telephone system). We consider a simple telephone system. In this telephone system each telephone is provided with a process, called its basic call process, to establish and maintain connections with other telephones. Actions of this process include receiving an off-hook or on-hook

signal from the telephone, receiving a dialed number from the telephone, sending a signal to start or to stop emitting a dial tone, ring tone or ring-back tone to the telephone, and receiving an alert signal from another telephone – indicating an incoming call. Suppose that a basic call process is in the idling state. In this state, it can change its state to the initial dialing state by receiving an off-hook signal from the telephone. Alternatively, it can change its state to the initial ringing state by receiving an alert signal from another telephone. In the initial dialing state, it can change its state to another dialing state by sending a signal to start emitting a dial tone to the telephone. In the initial ringing state, it can change its state to another ringing state by sending a signal to start emitting a ring tone to the telephone. And so forth.

Transition systems have been devised as a means to describe the behaviour of systems that have only discrete state changes. Despite this underlying purpose of transition systems, they can deal with continuous state changes as well. However, such use of transition systems will not be treated in this book. Instead, we focus on acquiring a good insight into the basic concepts of transition systems. That is not for pedagogical reasons alone. Systems that have only discrete state changes are still of utmost importance in the practice of developing computer-based systems and will remain so for a long time. Here are a couple of examples of the use of transition systems in describing the behaviour of systems with discrete state changes.

Example 1.1.2 (Bounded counter). We first consider a very simple system, viz. a bounded counter. A bounded counter can perform increments of its value by 1 till a certain value k is reached and can perform decrements of its value by 1 till the value 0 is reached. As states of a bounded counter, we have the natural numbers 0 to k . State i is the state in which the value of the counter is i . As actions, we have *inc* (increment) and *dec* (decrement). As transitions of a bounded counter, we have the following:

- for each state i that is less than k , a transition from state i to state $i + 1$ labeled with the action *inc*, written $i \xrightarrow{\text{inc}} i + 1$;
- for each state i that is less than k , a transition from state $i + 1$ to state i labeled with the action *dec*, written $i + 1 \xrightarrow{\text{dec}} i$.

If the number of states and transitions is small, a transition system can easily be represented graphically. The transition system describing the behaviour of the bounded counter is represented graphically in Figure 1.1 for the case where $k = 3$. In the graphical representation of transition systems, circles or ovals are used to denote the states of the transition system. In many cases, for easy reference, the identity of the state is written inside the circle or oval. A transition $s \xrightarrow{a} s'$ is represented by an arrow, labelled by the action name a , from the circle representing the state s to the circle representing the state s' . Notice that the bounded counter has a finite number of states and a finite number of transitions. Furthermore, the bounded counter will never become

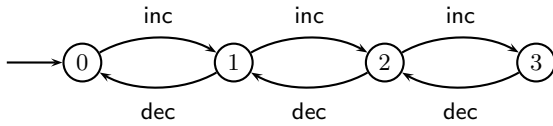


Fig. 1.1. Transition system for the bounded counter

inactive, i.e. it will never reach a state from which no transition is possible. Thus, the finiteness of the bounded counter does not keep the counter from making an infinite number of transitions. The bounded counter can easily

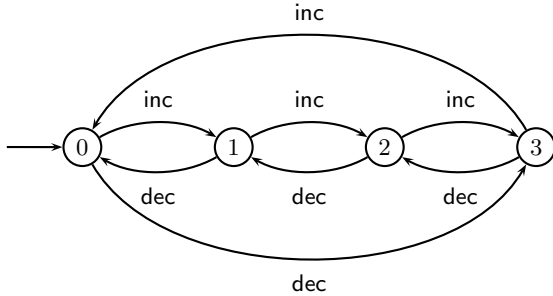


Fig. 1.2. Transition system for the modulo counter

be adapted to become a counter modulo $k + 1$, i.e. a counter whose value becomes 0 by performing an increment by 1 when its value is k and whose value becomes k by performing a decrement by 1 when its value is 0. We have the same states and actions as before and we have two additional transitions (see Figure 1.2):

- a transition from state k to state 0 labeled with the action `inc`, written $k \xrightarrow{\text{inc}} 0$;
- a transition from state 0 to state k labeled with the action `dec`, written $0 \xrightarrow{\text{dec}} k$.

Example 1.1.3 (Bounded buffer). We next consider another simple system, viz. a bounded buffer. A bounded buffer can add new data to the sequence of data that it keeps if the capacity l of the buffer is not exceeded, i.e. if the length of the sequence of data that it keeps is not greater than l . As long as it keeps data, it can remove the data that it keeps – in the order in which they were added. As states of a bounded buffer, we have the sequences of data of which the length is not greater than l . State σ is the state in which the sequence of data σ is kept in the buffer. As actions, we have `add(d)` (add d) and `rem(d)` (remove d) for each datum d . As transitions of a bounded buffer, we have the following:

- for each datum d and each state σ that has a length less than l , a transition from state σ to state $d\sigma$ labeled with the action $\text{add}(d)$, written $\sigma \xrightarrow{\text{add}(d)} d\sigma$;
- for each datum d and each state σd , a transition from state σd to state σ labeled with the action $\text{rem}(d)$, written $\sigma d \xrightarrow{\text{rem}(d)} \sigma$.

The transition system describing the behaviour of the bounded buffer is represented graphically in Figure 1.3 for the case where $l = 2$ and the only data involved are the natural numbers 0 and 1. Although it has a finite capacity,

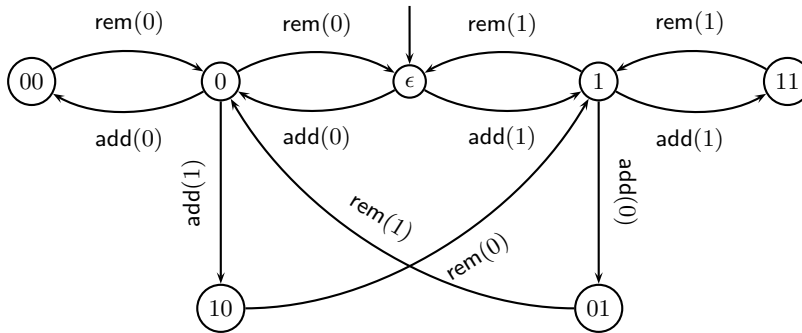


Fig. 1.3. Transition system for the bounded buffer

the bounded buffer will have an infinite number of states and an infinite number of transitions in the case where the number of data involved is infinite.

The bounded buffer can easily be adapted to become unreliable, e.g. to get into an error state by adding a datum when it is full. We have one additional state, say *err*, no additional actions, and the following additional transitions:

- for each datum d and each state σ that has a length equal to l , a transition from state σ to state *err* labeled with the action $\text{add}(d)$, written $\sigma \xrightarrow{\text{add}(d)} \text{err}$.

Notice that this unreliable bounded buffer will become inactive when it reaches the state *err*. Thus, the additional feature of this buffer may keep it from making an infinite number of transitions.

It is usual to designate one of the states of a transition system as its initial state. At the start-up of a system, i.e. before it has performed any action, the system is considered to be in its initial state. The expected initial states of the bounded counter from Example 1.1.2 and the bounded buffer from Example 1.1.3 are 0 and ϵ (the empty sequence), respectively. The initial state of a transition system is indicated by an unconnected incoming arrow.

Although bounded counters and buffers arise frequently as basic components in computer-based systems, they are not regarded as typical examples of real-life computer-based systems. In the following example, we consider a simplified version of a small real-life computer-based system, viz. a calculator.

Example 1.1.4 (Calculator). We consider a calculator that can perform simple arithmetical operations on integers. It can only perform addition, subtraction, multiplication and division on integers between a certain values, say min and max . As states of the calculator, we have pairs (i, o) , where $min \leq i \leq max$ or $i = *$ and $o \in \{\text{add, sub, mul, div, eq, clr, *}\}$. State (i, o) is roughly the state in which the result of the preceding calculations is i and the operator that must be applied next is o . If $o = *$, the operator that must be applied next is not available; and if in addition $i = *$, the result of the preceding calculations is not available either. As initial state, we have the pair $(*, *)$. As actions, we have $rd(i)$ (read operand i) and $wr(i)$ (write result i), both for $min \leq i \leq max$, and $rd(o)$ (read operator o), for $o \in \{\text{add, sub, mul, div, eq, clr}\}$. As transitions of the calculator, we have the following:

- for each i with $min \leq i \leq max$:
 - a transition $(*, *) \xrightarrow{rd(i)} (i, *)$,
 - a transition $(i, \text{clr}) \xrightarrow{wr(0)} (*, *)$,
 - a transition $(i, \text{eq}) \xrightarrow{wr(i)} (i, *)$;
- for each i with $min \leq i \leq max$ and $o \in \{\text{add, sub, mul, div, eq, clr}\}$:
 - a transition $(i, *) \xrightarrow{rd(o)} (i, o)$;
- for each i with $min \leq i \leq max$ and j with $min \leq j \leq max$:
 - a transition $(i, \text{add}) \xrightarrow{rd(j)} (i + j, *)$ if $min \leq i + j \leq max$,
 - a transition $(i, \text{sub}) \xrightarrow{rd(j)} (i - j, *)$ if $min \leq i - j \leq max$,
 - a transition $(i, \text{mul}) \xrightarrow{rd(j)} (i \cdot j, *)$ if $min \leq i \cdot j \leq max$,
 - a transition $(i, \text{div}) \xrightarrow{rd(j)} (i \div j, *)$ if $min \leq i \div j \leq max$ and $j \neq 0$.

The transition system describing the behaviour of the calculator is represented graphically in Figure 1.4 for the case where $min = 0$ and $max = 1$. Although the extremely small range of integers makes this case actually useless, it turns out to be difficult to represent the transition system graphically. The textual description given above is still intelligible. However, it is questionable whether this would be the case for a more realistic calculator.

Examples like Example 1.1.4 indicate that in the case of real-life systems we probably need a way to describe process behaviour more concisely than by directly giving a transition system. This is one of the issues treated in the remaining chapters of this book.

Example 1.1.5 (Automatic teller machine). We consider the operation of a simple automatic teller machine (ATM) and we focus on the interaction between the user and the ATM. The user inserts his/her bank card (insert card)

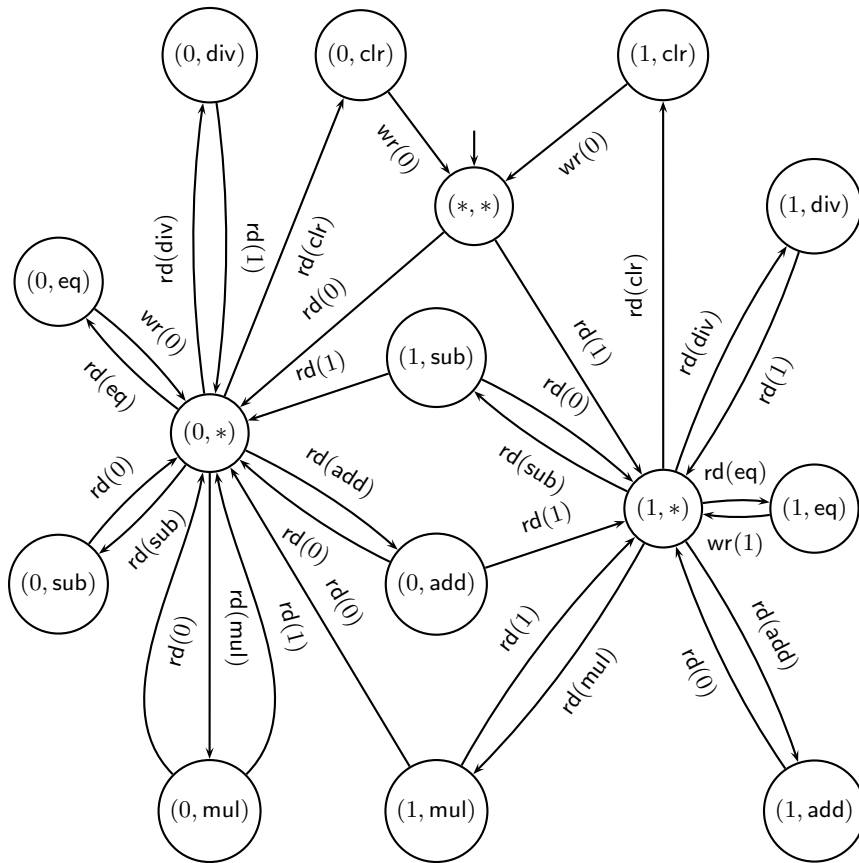


Fig. 1.4. Transition system for the calculator

and enters the Personnel Identification Number (enter PIN). The ATM signals to the user the outcome of some validation procedure (PIN valid or PIN invalid). In case the PIN is invalid, the ATM ejects the bank card (eject card) and returns to the idle state. In case, the PIN is validated, the user enters an amount of cash to be withdrawn (enter amount). After consulting with the bank the ATM signals whether this cash amount is balanced by the account of the user (accept amount) or not accepted (cancel). In the latter case the bank card is ejected. In case of acceptance, the ATM offers the cash (withdrawn) and ejects the bank card. The transition system for this ATM is given in Figure 1.5. Observe that the names of the states are not shown in this figure.

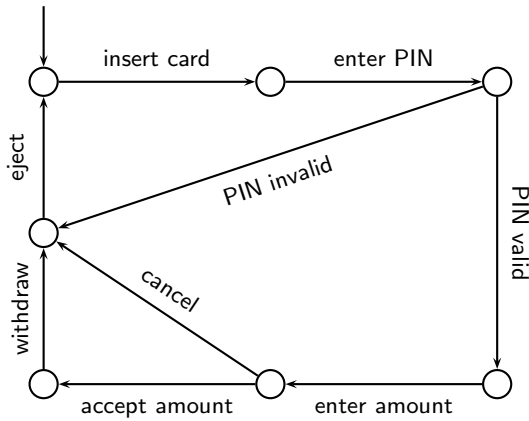


Fig. 1.5. Transition system for the user interface of an ATM

In describing the ATM in Example 1.1.5 we have chosen not to consider the procedure of checking the PIN and the procedure of checking whether the amount of cash is balanced by the bank account in detail. The reason to do so is that in this example our interest is with the user of the ATM. For the user of the ATM only the outcome of such procedures is of interest.

Exercise 1.1.1 (Unreliable bounded buffer). Give a transition system for the unreliable bounded buffer as discussed in Example 1.1.3.

Exercise 1.1.2 (One-place buffer). Give a transition system for a one-place buffer that can store values from the set $D = \{0, 1, 2\}$. The actions are $\text{put}(d)$ for putting a datum $d \in D$ into the buffer and $\text{take}(d)$ for taking a datum $d \in D$ from the buffer.

Exercise 1.1.3 (ATM with stop button). Adapt the transition system for the simple ATM from Example 1.1.5 in such a way that the user can push a stop button (stop) resulting in the abortion of the transaction. Of course the bank card (if already inserted) should in all cases be returned to the user. Pay attention to the situation that the transaction is confirmed by the ATM (probably this means that the money is already subtracted from the account of the user).

Exercise 1.1.4 (ATM with PIN-retry). Adapt the transition system for the simple ATM from Example 1.1.5 in such a way that upon invalidation of the PIN, the user can try again to enter a valid PIN. It should be the case that after entering three invalid PINs the bank card is not ejected anymore.

1.2 Formal definition

With the previous section, we have prepared the way for the formal definition of the notion of transition system.

Definition 1.2.1. A *transition system* T is a quintuple $(S, A, \rightarrow, \downarrow, s_0)$ where

- S is a set of *states*;
- A is a set of *actions*;
- $\rightarrow \subseteq S \times A \times S$ is a set of *transitions*;
- $\downarrow \subseteq S$ is a set of *successfully terminating states*;
- $s_0 \in S$ is the *initial state*.

If S and A are finite, T is called a *finite* transition system. We write $s \xrightarrow{a} s'$ and $s \downarrow$ instead of $(s, a, s') \in \rightarrow$ and $s \in \downarrow$, respectively. We write $\text{act}(T)$ for A , i.e. the set of actions of T . The set $\twoheadrightarrow \subseteq S \times A^* \times S$ of *generalized transitions* of T is the smallest subset of $S \times A^* \times S$ satisfying:

- $s \xrightarrow{\epsilon} s$ for each $s \in S$;
- if $s \xrightarrow{a} s'$, then $s \xrightarrow{a} s'$;
- if $s \xrightarrow{\sigma} s'$ and $s' \xrightarrow{\sigma'} s''$, then $s \xrightarrow{\sigma\sigma'} s''$.

The notations A^* , the set of all sequences over A , and $\sigma\sigma'$, the concatenation of sequences, are defined formally in Appendix A.3.

A state $s \in S$ is called a *reachable* state of T if there is a $\sigma \in A^*$ such that $s_0 \xrightarrow{\sigma} s$. The set of all reachable states of a transition system T is denoted $\text{reach}(T)$. A state $s \in S$ is called a *terminal* state of T if not $s \downarrow$ and there are no $a \in A$ and $s' \in S$ such that $s \xrightarrow{a} s'$. A state $s \in S$ is called a *deadlock* state of T if s is a reachable state of T and a terminal state of T .

We will now return to some of the transition systems introduced informally in the previous section. By convention, the set of successfully terminating states is considered to be empty, unless stated otherwise.

Example 1.2.1 (Bounded counter). We look again at the bounded counter from Example 1.1.2. Formally, the behaviour of a bounded counter with bound k is described by the transition system (S, A, \rightarrow, s_0) where

$$\begin{aligned} S &= \{i \in \mathbb{N} \mid i \leq k\}, \\ A &= \{\text{inc}, \text{dec}\}, \\ \rightarrow &= \{(i, \text{inc}, i+1) \mid i \in \mathbb{N}, i < k\} \cup \{(i+1, \text{dec}, i) \mid i \in \mathbb{N}, i < k\}, \\ s_0 &= 0. \end{aligned}$$

All states of this finite transition system are reachable. It does not have terminal states, and hence also no deadlock states.

Example 1.2.2 (Unreliable bounded buffer). We also look again at the unreliable bounded buffer from Example 1.1.3. Formally, the behaviour of the unreliable bounded buffer with capacity l is described by the transition system (S, A, \rightarrow, s_0) where

$$\begin{aligned}
S &= \{\sigma \in D^* \mid |\sigma| \leq l\} \cup \{\text{err}\}, \\
A &= \{\text{add}(d) \mid d \in D\} \cup \{\text{rem}(d) \mid d \in D\}, \\
\rightarrow &= \{(\sigma, \text{add}(d), d\sigma) \mid \sigma \in D^*, |\sigma| < l\} \\
&\quad \cup \{(\sigma, \text{add}(d), \text{err}) \mid \sigma \in D^*, |\sigma| = l\} \\
&\quad \cup \{(\sigma d, \text{rem}(d), \sigma) \mid \sigma \in D^*, |\sigma| < l\}, \\
s_0 &= \epsilon.
\end{aligned}$$

The notation $|\sigma|$, the length of the sequence σ , is defined formally in Appendix A.3. All states of this transition system are reachable. It has one terminal state, viz. `err`. This state is also a deadlock state.

Henceforth, we will only occasionally introduce transition systems in this formal style. After the informal explanation and formal definition of the notion of transition system, we are now in the position to relate it to the notions of program and automaton in the next two sections.

Exercise 1.2.1 (Traffic light). Give a transition system for a traffic light. The actions are the colours of the traffic light: `red`, `yellow`, and `green`. Initially, the traffic light is `red`.

Exercise 1.2.2 (Road crossing). Give a transition system for a one-directional road crossing with two traffic lights as presented in Figure 1.6. The actions involved are `red1`, `yellow1`, `green1`, `red2`, `yellow2`, and `green2`. Make

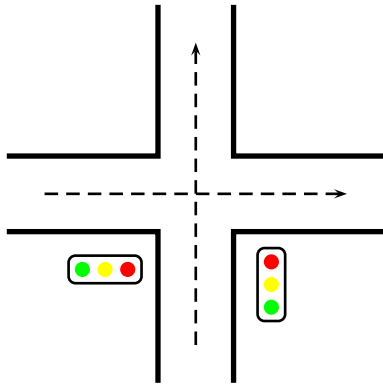


Fig. 1.6. Two traffic lights on a one-directional road crossing

sure that collisions cannot occur, assuming that drivers respect the traffic lights. Initially, both traffic lights are `red`.

Exercise 1.2.3 (Elevator system). Give a transition system for an elevator system. The elevator system serves 5 floors, numbered 0-4. The elevator starts at floor 0. The actions of the elevator system are `up` and `down` representing that the elevator goes one floor up and one floor down respectively.

Exercise 1.2.4 (Stopwatch). Consider the following stopwatch. The stopwatch has two buttons and one display. Initially, the display is empty. As soon as the start button is pushed, the stopwatch starts counting time (in seconds) from zero up. Pushing the stop button results in stopping the counting of seconds. After stopping the counting, on the display the amount of time that has elapsed is displayed.

Use the following actions: **start** for pushing the start button, **stop** for pushing the stop button, **display(s)** for displaying the elapsed time s (in seconds), and **tick** for modelling the passage of a second.

Decide for yourself how the stopwatch should react on pushing the start button while counting.

Exercise 1.2.5 (Binary adder). A binary adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use an inverse binary representation. A serial adder processes two such numbers $x = a_0a_1 \dots a_n$ and $b_0b_1 \dots b_m$ bit by bit, starting at the left hand.

Give a transition system for a binary adder. Assume that two bits are read through the actions $\text{read}(b_1, b_2)$ for $b_1, b_2 = 0, 1$, and that the bits of the sum are produced through the actions $\text{write}(b)$ for $b = 0, 1$.

1.3 Programs and transition systems

For a better understanding of the notion of transition system, we now look into its connections with the familiar notion of program.

The behaviour of a program upon execution can be regarded as a transition system. In doing so, we can abstract from how the actions performed by a program are processed by a machine, and hence from how the values assigned to the program variables are maintained. In that case, we focus on the flow of control. The states of the transition system only serve as the control points of the program and its actions are merely requests to perform actions such as assignments, tests, etc. What we have in view here will be called the behaviour of a program upon *abstract* execution to distinguish it clearly from the behaviour of a program upon execution on a machine, which applies to the processing by a machine of the actions performed by the program. Here is an example of the use of transition systems in describing the behaviour of programs upon abstract execution.

Example 1.3.1 (Factorial). We consider the following PASCAL program to calculate factorials¹:

```
PROGRAM factorial(input,output);
```

¹ For any natural number $n \in \mathbb{N}$, the factorial of n , denoted $n!$, is defined inductively by $0! = 1$ and $(n+1)! = (n+1) \times n!$.

```

VAR i,n,f: 0..maxint;
BEGIN
  read(n);
  i := 0; f := 1;
  WHILE i < n DO
    BEGIN i := i + 1; f := f * i END;
  write(f)
END

```

The behaviour of this program upon abstract execution can be described by a transition system as follows. As states of the factorial program, we have the natural numbers 0 to 7, with 0 as initial state. For easy reference we have inserted the numbers representing the states also in the PASCAL program as comments.

```

PROGRAM factorial(input,output);
VAR i,n,f: 0..maxint;
BEGIN {0}
  read(n); {1}
  i := 0; {2} f := 1; {3}
  WHILE i < n DO
    BEGIN {4} i := i + 1; {5} f := f * i END; {6}
  write(f) {7}
END

```

The states can be viewed as the values of a “program counter”. As actions, we have an action corresponding to each atomic statement² of the program as well as each test of the program and its opposite. As transitions, we have the following:

$$\begin{array}{l}
0 \xrightarrow{\text{read}(n)} 1, 1 \xrightarrow{i:=0} 2, 2 \xrightarrow{f:=1} 3, \\
3 \xrightarrow{i < n} 4, 4 \xrightarrow{i:=i+1} 5, 5 \xrightarrow{f:=f*i} 3, \\
3 \xrightarrow{\text{NOT } i < n} 6, 6 \xrightarrow{\text{write}(f)} 7.
\end{array}$$

Once the program has performed the statement `write(f)`, it cannot perform any more actions. As the program has reached its end successfully, the state associated, i.e., state 7, is designated as a final state. Hence $7 \downarrow$. The transition system for the factorial program is represented graphically in Figure 1.7.

Here is another example.

Example 1.3.2 (Greatest Common Divisor). We consider the following PASCAL program to calculate greatest common divisors³:

² An atomic statement is a statement that is not to be divided into “smaller” statements.

³ The greatest common divisor of two positive natural numbers m and n is the greatest natural number k such that k is a divisor of both m and n .

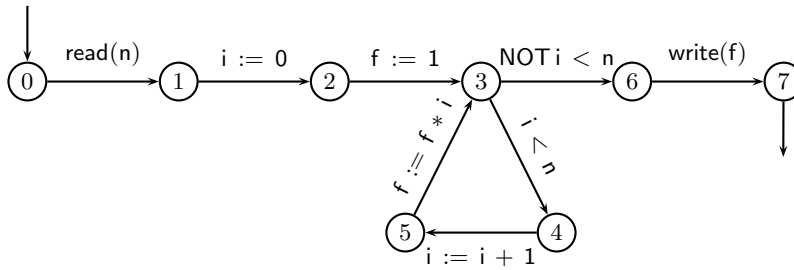


Fig. 1.7. Transition system for the factorial program

```

PROGRAM gcd(input,output);
VAR m,n: 1..maxint;
BEGIN
  read(m); read(n);
  REPEAT
    WHILE m > n DO m := m - n;
    WHILE n > m DO n := n - m
  UNTIL m = n;
  write(m)
END
  
```

The behaviour of this program upon abstract execution can be described by a transition system as follows. As states of the greatest common divisor program, we have the natural numbers 0 to 8, with 0 as initial state. As actions, we have an action corresponding to each atomic statement of the program as well as each test of the program and its opposite. As transitions, we have the following:

$$\begin{aligned}
 0 &\xrightarrow{\text{read}(m)} 1, 1 \xrightarrow{\text{read}(n)} 2, \\
 2 &\xrightarrow{m > n} 3, 3 \xrightarrow{m := m - n} 2, \\
 2 &\xrightarrow{\text{NOT } m > n} 4, 4 \xrightarrow{n > m} 5, 5 \xrightarrow{n := n - m} 4, 4 \xrightarrow{\text{NOT } n > m} 6, 6 \xrightarrow{\text{NOT } m = n} 2, \\
 6 &\xrightarrow{m = n} 7, 7 \xrightarrow{\text{write}(m)} 8.
 \end{aligned}$$

The final state of this program is state 8, hence $8 \downarrow$. The transition system for the greatest common divisor program is represented graphically in Figure 1.8.

Notice that the transition systems described in Examples 1.3.1 and 1.3.2 have a single successfully terminating state.

A transition system derived from a program in the way described and illustrated above is reminiscent of a flowchart. However, the underlying idea is that the transition system describes the behaviour of the program upon execution in such a way that it can act concurrently and interact with a machine that processes the actions performed by the program. If it does so,

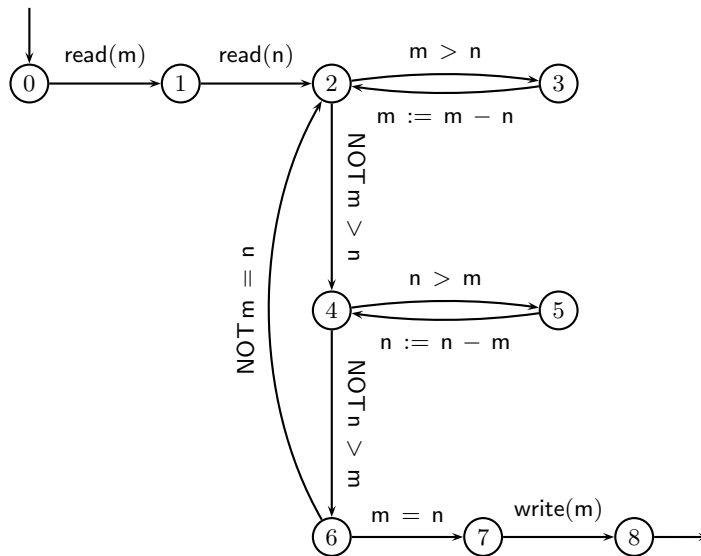


Fig. 1.8. Transition system for the greatest common divisor program

the combined behaviour can be regarded as the behaviour of the program upon execution on a machine. Interaction between processes is one of the issues treated in the remaining chapters of this book. We can also directly give a transition system describing the behaviour of the program upon execution on a machine. In that case, we have to take into account that an assignment changes the value of a program variable, the values of the program variables determine whether a test succeeds, etc. This is illustrated in the following couple of examples, which are concerned with the same programs as the previous two examples.

Example 1.3.3 (Factorial). We consider again the program from Example 1.3.1. The intended behaviour of this program upon execution on a machine can be described by a transition system as follows. As states of the program, we have pairs (l, s) , where $l \in \mathbb{N}$ with $0 \leq l \leq 7$ and $s = (i, n, f)$ with $i, n, f \in \{i \in \mathbb{N} \mid i \leq \text{maxint}\} \cup \{*\}$. These states can be viewed as follows: l is the value of the program counter and $s = (i, n, f)$ is the storage that keeps the values of the program variables i , n , and f in that order. The special value $*$ is used to indicate that no value has yet been assigned to a program variable. The initial state is $(0, (*, *, *))$. As actions, we have again an action corresponding to each atomic statement of the program as well as each test of the program and its opposite. As transitions, we have the following:

- for each n :
 - a transition $(0, (*, *, *)) \xrightarrow{\text{read}(n)} (1, (*, n, *))$,
 - a transition $(1, (*, n, *)) \xrightarrow{i:=0} (2, (0, n, *))$,

- a transition $(2, (0, n, *)) \xrightarrow{f:=1} (3, (0, n, 1));$
- for each i, n, f such that $i < n$ and $f = i!$:
 - a transition $(3, (i, n, f)) \xrightarrow{i < n} (4, (i, n, f)),$
 - a transition $(4, (i, n, f)) \xrightarrow{i:=i+1} (5, (i+1, n, f)),$
 - a transition $(5, (i+1, n, f)) \xrightarrow{f:=f*i} (3, (i+1, n, f \cdot (i+1)));$
- for each i, n, f such that $i = n$ and $f = i!$:
 - a transition $(3, (i, n, f)) \xrightarrow{\text{NOT } i < n} (6, (i, n, f)),$
 - a transition $(6, (i, n, f)) \xrightarrow{\text{write}(f)} (7, (i, n, f)).$

There are some noticeable differences between this transition system and the transition system from Example 1.3.1. The two relevant intuitions are as follows. In the same state, reading different numbers does not cause the same state change. In the same state, a test and its opposite do not succeed both.

Not all states are reachable. For example, states $(l, (i, n, f))$ with $i \neq *$ and $n \neq *$ for which $i > n$ holds are not reachable. We did not bother to restrict the transition system to the reachable states: we will see later that the resulting transition system would describe essentially the same behaviour.

Any state where the program counter equals 7 can be regarded a final state regardless the value of the program variables. However, only final states that additionally satisfy that $i = n$ and $f = i!$ are reachable. Thus we have the following final states:

- for each i, n, f such that $i = n$ and $f = i!$:
 - $(7, (i, n, f)) \downarrow.$

The transition system (restricted to the reachable states) for the factorial program is represented graphically in Figure 1.9 for the case where $\text{maxint} = 2$.

Example 1.3.4 (Greatest common divisor). We also consider again the program from Example 1.3.2. The intended behaviour of this program upon execution on a machine can be described by a transition system as follows. As states of the program, we have pairs (l, s) , where $l \in \mathbb{N}$ with $0 \leq l \leq 8$ and $s = (m, n)$ with $m, n \in \{i \in \mathbb{N} \mid 1 \leq i \leq \text{maxint}\} \cup \{*\}$. These states are like in Example 1.3.3. The initial state is $(0, (*, *))$. As actions, we have again an action corresponding to each atomic statement of the program as well as each test of the program and its opposite. As transitions, we have the following:

- for each m :
 - a transition $(0, (*, *)) \xrightarrow{\text{read}(m)} (1, (m, *));$
- for each m, n :
 - a transition $(1, (m, *)) \xrightarrow{\text{read}(n)} (2, (m, n));$
- for each m, n such that $m > n$:
 - a transition $(2, (m, n)) \xrightarrow{m > n} (3, (m, n)),$

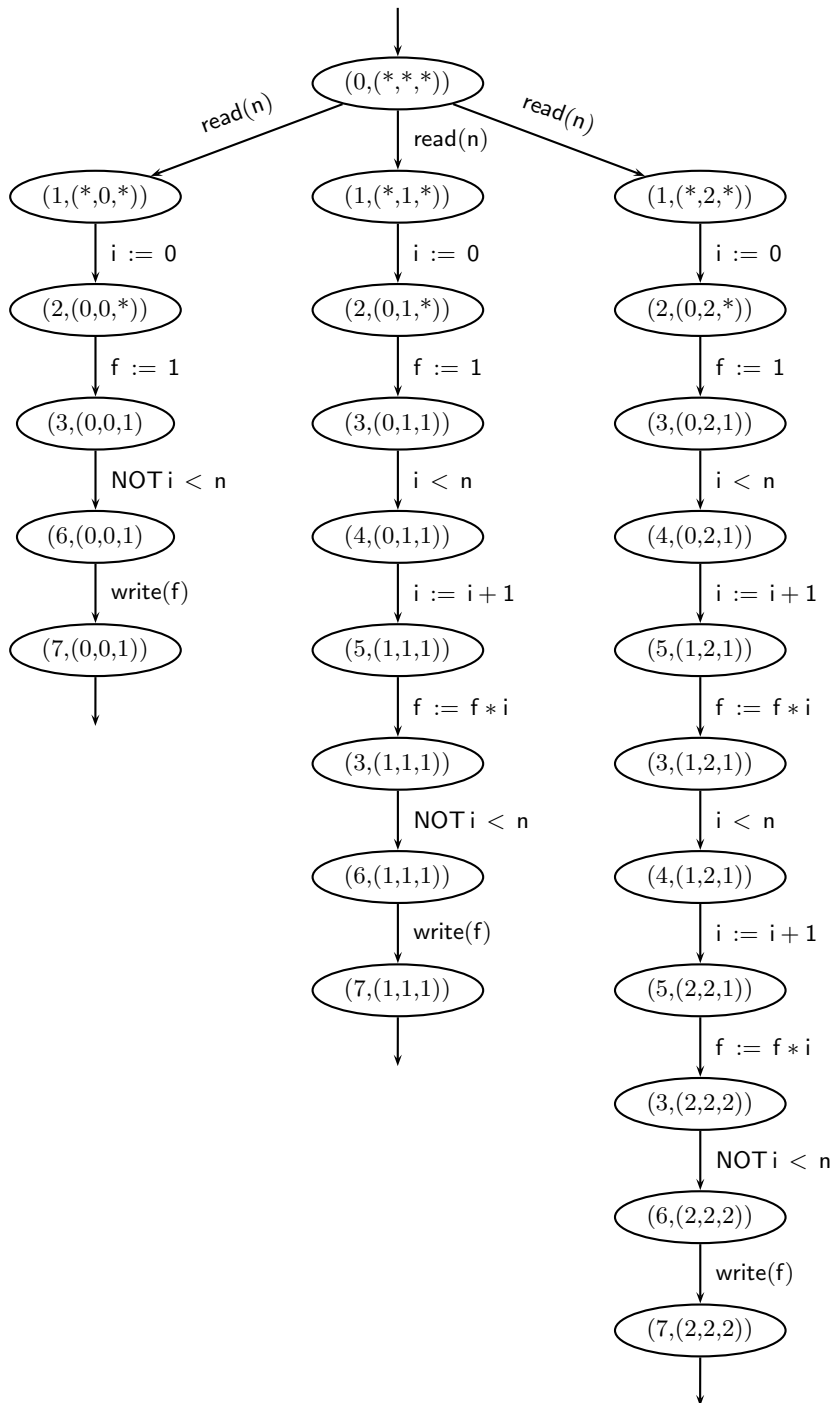


Fig. 1.9. Another transition system for the factorial program

- a transition $(3, (m, n)) \xrightarrow{m := m - n} (2, (m - n, n));$
- for each m, n such that $m \leq n$:
 - a transition $(2, (m, n)) \xrightarrow{\text{NOT } m > n} (4, (m, n));$
- for each m, n such that $m < n$:
 - a transition $(4, (m, n)) \xrightarrow{n > m} (5, (m, n)),$
 - a transition $(5, (m, n)) \xrightarrow{n := n - m} (4, (m, n - m));$
- for each m, n such that $m \geq n$:
 - a transition $(4, (m, n)) \xrightarrow{\text{NOT } n > m} (6, (m, n));$
- for each m, n such that $m \neq n$:
 - a transition $(6, (m, n)) \xrightarrow{\text{NOT } m = n} (2, (m, n));$
- for each m, n such that $m = n$:
 - a transition $(6, (m, n)) \xrightarrow{m = n} (7, (m, n)),$
 - a transition $(7, (m, n)) \xrightarrow{\text{write}(m)} (8, (m, n)).$

The final states of this transition system are those where the value of the program counter equals 8 and additionally the values of the program variables m and n are equal:

- for each m, n such that $m = n$:
 - $(8, (m, n)) \downarrow.$

The differences between this transition system and the transition system given in Example 1.3.2 are of the same kind as between the transition systems given for factorial program. Like in Example 1.3.3, not all states are reachable. The transition system for the greatest common divisor program is represented graphically in Figure 1.10 for the case where $maxint = 2$.

For a given programming language, the behaviour of its programs upon execution on a machine is called its operational semantics. It is usually described in a style known as structural operational semantics. This means that the behaviour of a compound language construct is described in terms of the behaviour of its constituents. The transition systems from the previous two examples were not formally based on a given (structural) operational semantics.

1.4 Automata and transition systems

For a better understanding of the notion of transition system, we looked in the previous section into its connections with the familiar notion of program. For the same reason, we now look into its connections with the familiar notion of automaton.

Automata can be regarded as a specialized kind of transition systems. In this section, we restrict ourselves to the kind of automata known as *non-deterministic finite accepters*. We also do not allow transitions labelled with

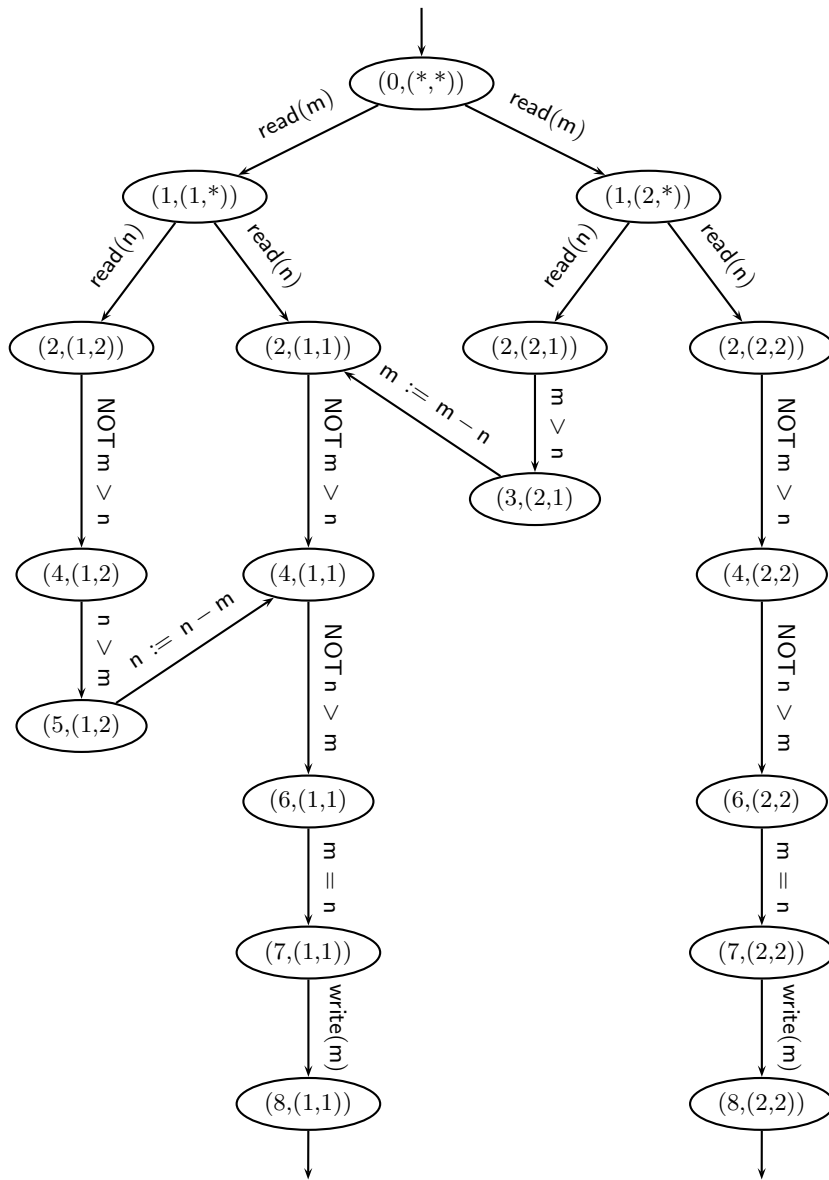


Fig. 1.10. Another transition system for the greatest common divisor program

λ (representing a transition without reading a symbol)⁴. They are illustrative for almost any kind of automata. If no confusion can arise, we will call them simply *automata*. The difference between automata and transition systems is mainly a matter of intended use. As mentioned in Section 1.1, transition systems are primarily regarded as a means to describe the behaviour of processes and automata are primarily regarded as abstract machines to recognize certain languages. Because of the different intended use, final states are indispensable in the case of automata: reaching a final state means that a complete sentence has been recognized. We do not give the standard definition of the notion of automaton. Our definition underlines the resemblance to transition systems mentioned above.

Definition 1.4.1. An *automaton* M is a quintuple $(S, A, \rightarrow, s_0, F)$ where

- S is a finite set of *internal states*;
- A is a finite set of *symbols*, called the *input alphabet*;
- $\rightarrow \subseteq S \times A \times S$ is a set of *transitions*;
- $s_0 \in S$ is the *initial state*;
- $F \subseteq S$ is a set of *final states*.

The set $\rightarrow \subseteq S \times A^* \times S$ of *generalized transitions* of M is defined exactly as for transition systems. The *language* accepted by M , written $\mathcal{L}(M)$, is the set $\{\sigma \in A^* \mid s_0 \xrightarrow{\sigma} s \text{ for some } s \in F\}$.

An important difference between automata and transition systems is that in automata both the set of internal states and the set of symbols are finite. In the standard definition of the notion of automaton, we have a *transition function* $\delta: S \times A \rightarrow \mathcal{P}(S)$ instead of a set $\rightarrow \subseteq S \times A \times S$ of transitions. If we take δ such that $s' \in \delta(s, a)$ if and only if $s \xrightarrow{a} s'$, then we get an automaton according to the standard definition.

If we regard symbols as actions of reading the symbols, automata are simply transition systems with designated final states. An automaton can be considered to accept certain sequences of symbols as follows. A transition of an automaton is regarded as a state change caused by reading a symbol. A sequence of symbols $a_1 \dots a_n$ is accepted if a sequence of consecutive state changes from the initial state to one of the final states can be obtained by reading the symbols a_1, \dots, a_n in turn. This informal explanation can be made more precise as follows.

Let M be the automaton $(S, A, \rightarrow, s_0, F)$ and let A' be the set of actions $\{\text{read}(a) \mid a \in A\}$. Now consider the transition system $T = (S, A', \rightarrow', F, s_0)$ where $s_1 \xrightarrow{\text{read}(a)'} s_2$ iff $s_1 \xrightarrow{a} s_2$. The sentences of the language accepted by M are exactly the sequences of symbols that can be consecutively read by T till a state is reached that is contained in F .

⁴ As any non-deterministic finite acceptor with λ -transitions can be replaced by a non-deterministic finite acceptor without such λ -transitions such that the two non-deterministic finite acceptors accept the same language, this imposes no real limitations.

Let us look at a simple example of the use of automata in recognizing a language.

Example 1.4.1 (Sentences). We consider a very simple language. A sentence of the language consists of a noun clause followed by a verb followed by a noun clause. A noun clause consists of an article followed by a noun. A noun is either *man* or *machine*. A verb is either *simulates* or *mimics*. An example sentence is *the man mimics a machine*. This language is accepted by the following automaton. As internal states of the automaton, we have pairs (p, i) , where $p \in \{\text{left}, \text{right}\}$ and $i \in \mathbb{N}$ with $0 \leq i \leq 2$. The choice of states is not really relevant. We could have taken the natural numbers 0 to 5 equally well, but the choice made here allows for a short presentation of the automaton. The initial state is $(\text{left}, 0)$ and the only final state is $(\text{right}, 2)$. The input alphabet consists of *a*, *the*, *man*, *machine*, *simulates* and *mimics*. As transitions, we have the following:

- for $p = \text{left}, \text{right}$:
 - a transition $(p, 0) \xrightarrow{a} (p, 1)$,
 - a transition $(p, 0) \xrightarrow{\text{the}} (p, 1)$,
 - a transition $(p, 1) \xrightarrow{\text{man}} (p, 2)$,
 - a transition $(p, 1) \xrightarrow{\text{machine}} (p, 2)$;
- a transition $(\text{left}, 2) \xrightarrow{\text{simulates}} (\text{right}, 0)$;
- a transition $(\text{left}, 2) \xrightarrow{\text{mimics}} (\text{right}, 0)$.

The automaton for our very simple language is represented graphically in Figure 1.11. It is obvious that this automaton accepts the same sequences

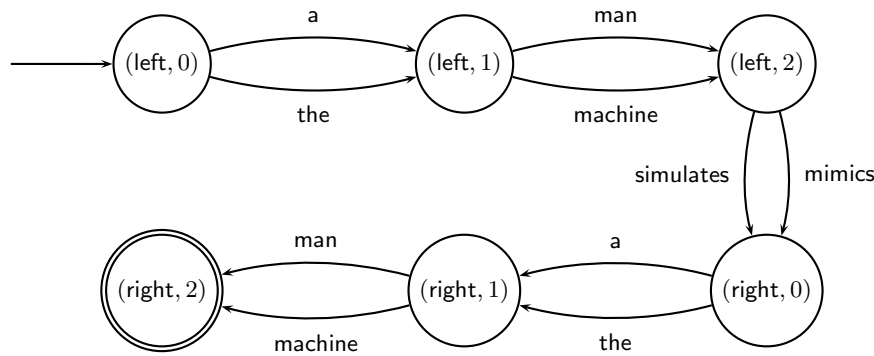


Fig. 1.11. Automaton accepting a very simple language

of symbols as the finite transition system obtained from this automaton by replacing the symbols *a*, *the*, *man*, *machine*, *simulates* and *mimics* by actions of reading these symbols.

Conversely, we can also view any finite transition system as an automaton by regarding its actions as symbols and its successfully terminating states as final states. This is interesting because the sequences of actions it can consecutively perform are an important aspect of the behaviour of a process. We will get back to that later in Section 1.5. Here is an example that illustrates the potential usefulness of focussing on the sequences of actions that a system can consecutively perform.

Example 1.4.2 (Bounded counter). We consider the bounded counter with bound k from Example 1.1.2. The behaviour of this bounded counter is described by a transition system in Example 1.2.1. The sequences of actions that can be performed are exactly the sequences w that satisfy the following condition:

- for all prefixes⁵ v of w , $0 \leq n_{\text{inc}}(v) - n_{\text{dec}}(v) \leq k$;

where $n_a(u)$ stands for the number of occurrences of action a in sequence u . This description of the sequences of actions that can be performed may be regarded as the specification of the intended system.

If we designate all reachable states of the transition system as final states, the transition system can be viewed as an automaton recognizing the language on the alphabet $\{\text{inc}, \text{dec}\}$ that consists of the sequences $w \in \{\text{inc}, \text{dec}\}^*$ satisfying the conditions just mentioned. When viewing the transition system as an automaton, the point is that `inc` and `dec` are considered to be symbols to be read instead of actions to be performed.

The following is known from automata theory. The languages that can be accepted by an automaton as defined here, i.e. a non-deterministic finite acceptor without λ -transitions, are exactly the regular languages. Intuitively, a regular language has a structure simple enough that a limited memory is sufficient to accept all its sentences. Many actual languages are not regular. Broader language categories include the context-free languages and the context-sensitive languages. They can be accepted by automata of more powerful kinds: non-deterministic pushdown acceptors for context-free languages and linear bounded acceptors for context-sensitive languages. Those kinds of automata are in turn closely related to restricted kinds of infinite transition systems.

1.5 Equivalences on processes

In this section, we look at a couple of notions that are taken up to abstract from those details of transition systems that are often supposed to be irrelevant.

⁵ See Appendix A.3 for a definition of prefixes.

Example 1.5.1. Suppose that we want to implement a program that prints the natural numbers from 1 upto N (for some $N \geq 1$). There are many programs that solve this problem. For example the programs

```
PROGRAM numbers(input,output);
VAR i: 0..maxint;
CONST n = N
BEGIN
  FOR i := 1 TO n DO write(i) OD
END
```

and

```
PROGRAM numbers(input,output);
VAR j: 0..maxint;
CONST n = N
BEGIN
  FOR j := n DOWNTO 1 DO write(n-j+1) OD
END
```

both result in a list of consecutive natural numbers starting from 1 and ending with N being printed on the screen. The transition systems that could be used to express the execution of these programs are $T = (S, A, \rightarrow, \downarrow, s_0)$ and $T' = (S', A', \rightarrow', \downarrow', s'_0)$ where

$$\begin{aligned} S &= \{0, 1, \dots, N\}, \\ A &= \{0, 1, \dots, N\}, \\ \rightarrow &= \{(i, i+1, i+1) \mid 0 \leq i < N\}, \\ \downarrow &= \{N\}, \\ s_0 &= 0 \end{aligned}$$

and

$$\begin{aligned} S' &= \{0, 1, \dots, N\}, \\ A' &= \{0, 1, \dots, N\}, \\ \rightarrow' &= \{(j, N-j+1, j-1) \mid 0 < j \leq N\}, \\ \downarrow' &= \{0\}, \\ s'_0 &= N \end{aligned}$$

respectively. Obviously, these transition systems are not the same as $\rightarrow \neq \rightarrow'$, $\downarrow \neq \downarrow'$, and $s_0 \neq s'_0$. Nevertheless, if we are interested in the results of the programs on screen, we can observe no difference between the two programs.

Definition 1.5.1 (Isomorphism). Two transition systems $T = (S, A, \rightarrow, \downarrow, s_0)$ and $T' = (S', A', \rightarrow', \downarrow', s'_0)$ are *isomorphic*, notation $T \cong T'$, if and only if there exists a bijective⁶ function $h: \text{reach}(T) \rightarrow \text{reach}(T')$ such that

⁶ See Appendix A.2 for a definition of bijective functions.

1. $\bar{h}(s_0) = s'_0$;
2. whenever $\bar{h}(s_1) = s'_1$ and $\bar{h}(s_2) = s'_2$, then $s_1 \xrightarrow{a} s_2$ if and only if $s'_1 \xrightarrow{a'} s'_2$;
3. whenever $\bar{h}(s) = s'$, then $s \downarrow$ if and only if $s' \downarrow$.

Example 1.5.2. For the previous example the function \bar{h} from the states S (which are precisely the reachable states) of T to the (reachable) states S' of T' defined by $\bar{h}(s) = N - s$ is a bijective function that proves $T \cong T'$. The transition systems and the function \bar{h} proving the isomorphism between them are illustrated in Figure 1.12 for $N = 4$.

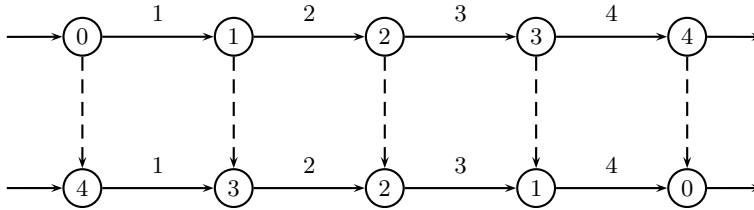
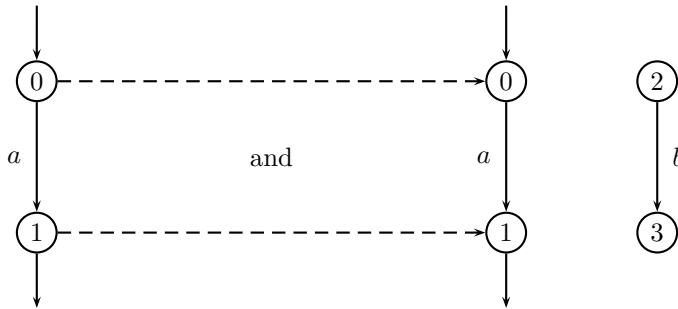


Fig. 1.12. Isomorphic transition systems

Property 1.5.1. Isomorphism is an equivalence relation, i.e., a reflexive, symmetric, and transitive relation.

Example 1.5.3. Consider the transition systems given below. As one can easily see, the second transition system has states that cannot be reached from the initial state by any sequence of actions. Nevertheless, these transition systems are considered to be isomorphic.



The function \bar{h} that proves this isomorphism is given by $\bar{h}(0) = 0$ and $\bar{h}(1) = 1$.

In the sequel we will mostly neglect the unreachable states of transition systems as well as the transitions starting from those. This can be considered an operation on transition systems, called reduction.

Definition 1.5.2. Let $T = (S, A, \rightarrow, \downarrow, s_0)$ be a transition system. Then the *reduction* of T , written $\text{red}(T)$, is the transition system $(S', A', \rightarrow', \downarrow', s_0)$ where

- $S' = \text{reach}(T)$;
- $A' = \{a \in A \mid \text{for some } s, s' \in S': s \xrightarrow{a} s'\}$;
- $\rightarrow' = \rightarrow \cap (S' \times A' \times S')$;
- $\downarrow' = \downarrow \cap S'$.

Any transition system is isomorphic to its reduction, which is a connected transition system.

Property 1.5.2. Let T be a transition system. Then the following holds:

$$T \cong \text{red}(T).$$

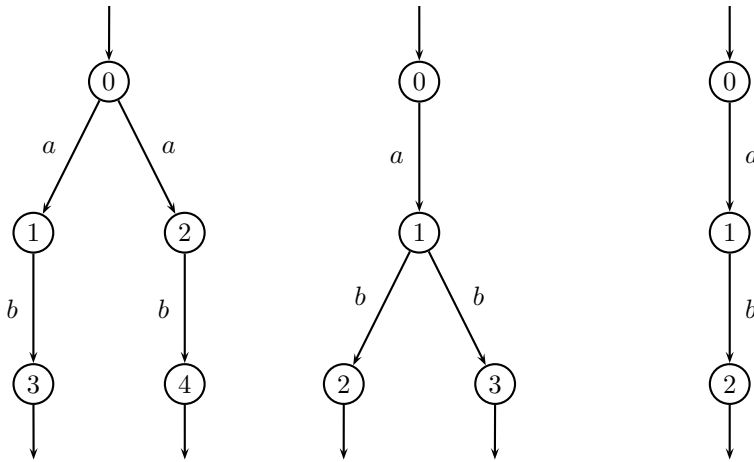
Proof. The function $h : \text{reach}(T) \rightarrow \text{reach}(T')$ defined by $h(s) = s$ for all $s \in \text{reach}(T)$ is a bijective function from the reachable states of T to the reachable states of $\text{red}(T)$ (which are actually the same as the reachable states of T). It also satisfies the conditions from the definition of isomorphy (Definition 1.5.1).

Usually, transition systems show details that are not considered to be relevant to the behaviour of processes. There are, for example, applications of transition systems where only the sequences of actions that can be performed consecutively starting from the initial state and ending in a successfully terminating state, called the terminating traces of the transition system, matter. This is, for example, the case when transition systems are used for the same purposes as automata, i.e., to accept languages.

Definition 1.5.3 (Language equivalence). Let $T = (S, A, \rightarrow, \downarrow, s_0)$ be a transition system. A *terminating trace* of T is a sequence $\sigma \in A^*$ such that $s_0 \xrightarrow{\sigma} s$ and $s \downarrow$ for some $s \in S$. We write $\text{lang}(T)$ for the set of all terminating traces of T . Two transition systems T and T' are *language equivalent*, written $T \equiv_1 T'$, if $\text{lang}(T) = \text{lang}(T')$.

Obviously the terminology used here is based on viewing a transition system as an automaton by regarding its actions as symbols and its successfully terminating states as final states, cf. Section 1.4.

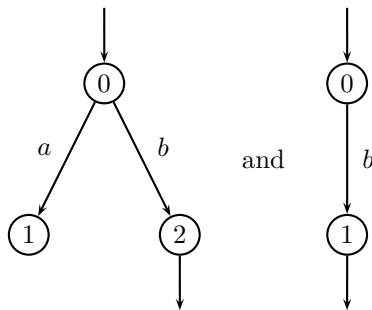
Example 1.5.4. The following transition systems are language equivalent as for all of them the set of terminating traces only consists of the trace ab .



Observe that these transition systems all differ with respect to isomorphy.

The following example illustrates that language equivalence abstracts from the parts of a transition system that do not lead to a successfully terminating state.

Example 1.5.5. The transition systems



called T_1 and T_2 , respectively, are language equivalent ($T_1 \equiv_1 T_2$) as for both transition systems the set of terminating traces only consists of the trace b : $\text{lang}(T_1) = \text{lang}(T_2) = \{b\}$. Thus the sequence of actions a is neglected in the comparison of the transition systems.

Property 1.5.3. Language equivalence is an equivalence relation.

Transition systems that are isomorphic are identical except for the identity of their states. As trace equivalence also abstracts from the identity of the states, any two isomorphic transition systems are necessarily also language equivalent. This is expressed by the following property.

Property 1.5.4. Any two isomorphic transition systems are also language equivalent: if $T \cong T'$, then $T \equiv_1 T'$.

Notice that in all cases where a transition system is used to accept a language, as described in Section 1.4, only the terminating traces are relevant. In fact, language equivalence of automata and language equivalence of the transition systems representing these automata are identical.

Property 1.5.5. Let $T = (S, A, \rightarrow, s_0, F)$ and $T' = (S', A', \rightarrow, s'_0, F')$ be automata. Then, $\mathcal{L}(T) = \mathcal{L}(T')$ if and only if $T_1 \equiv_1 T'_1$, where $T_1 = (S, A, \rightarrow, F, s_0)$ and $T'_1 = (S', A', \rightarrow, F', s'_0)$.

A prominent aspect of language equivalence is that only the sequences of actions are considered that end in a successfully terminating state. Very often, processes are not supposed to terminate in any state. There are applications of transition systems where all the sequences of actions that can be performed consecutively starting from the initial state of a transition system, called the traces of the transition system, matter. Here is a simple example of a case where only the traces matter.

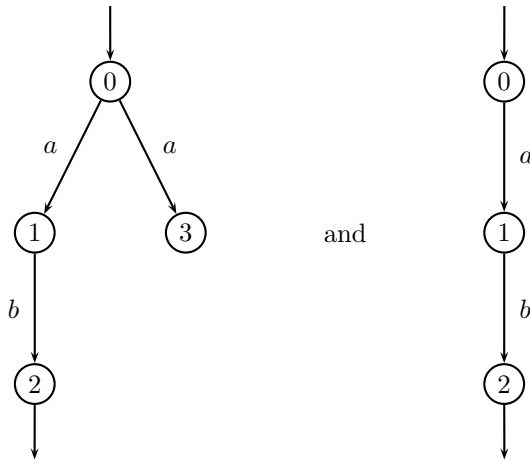
Example 1.5.6. We consider again the the bounded counter with bound k from Example 1.1.2. Its traces are exactly the traces w for which the condition $0 \leq n_{\text{inc}}(w) - n_{\text{dec}}(w) \leq k$ holds⁷. This description of its traces expresses all we expect from the bounded counter: we regard any transition system that has those traces as a bounded counter. For this reason, only the traces are relevant in this case.

In all those cases where only the traces of the transition system matter, it is useful to ignore all other details. This is done by identifying transition systems that not only have the same set of terminating traces, but additionally have the same set of traces. Such transition systems are called trace equivalent. Here is a precise definition.

Definition 1.5.4 (Trace equivalence). Let $T = (S, A, \rightarrow, \downarrow, s_0)$ be a transition system. A *trace* of T is a sequence $\sigma \in A^*$ such that $s_0 \xrightarrow{\sigma} s$ for some $s \in S$. We write $\text{traces}(T)$ for the set of all traces of T . Then two transition systems T and T' are *trace equivalent*, written $T \equiv_{\text{tr}} T'$, if $\text{traces}(T) = \text{traces}(T')$ and $\text{lang}(T) = \text{lang}(T')$.

Example 1.5.7. Consider the transition systems

⁷ Recall from Exercise 1.4.2 that $n_a(u)$ stands for the number of occurrences of action a in sequence u .



called T_1 and T_2 , respectively. These transition systems have the same terminating traces: $\text{lang}(T_1) = \text{lang}(T_2) = \{ab\}$. Also, they have the same traces: $\text{traces}(T_1) = \text{traces}(T_2) = \{\epsilon, a, ab\}$. Hence, these transition systems are considered trace equivalent: $T_1 \equiv_{\text{tr}} T_2$.

Example 1.5.8. The transition systems T_1 and T_2 from Example 1.5.5 are not trace equivalent ($T_1 \not\equiv_{\text{tr}} T_2$). The transition system T_1 has a trace a , whereas the transition system T_2 does not have such a trace. More precisely: $\text{traces}(T_1) = \{\epsilon, a, b\}$ and $\text{traces}(T_2) = \{\epsilon, b\}$.

Property 1.5.6. Trace equivalence is an equivalence relation.

From the previous discussions and the above example, it should be clear that trace equivalence identifies less transition systems than language equivalence.

Property 1.5.7. Any two trace equivalent transition systems are also language equivalent: if $T \equiv_{\text{tr}} T'$, then $T \equiv_1 T'$.

As can be seen from the above definition and examples trace equivalence abstracts from the precise identity of the states of the transition systems and from the moments of branching. Isomorphy only abstracts from the identity of the states and is therefore a stronger equivalence (i.e., it identifies less transition systems). These observations lead to the following property.

Property 1.5.8. Any two isomorphic transition systems are also trace equivalent: if $T \cong T'$, then $T \equiv_{\text{tr}} T'$.

We will see below that there are also cases where not only the traces of the transition system matter. In those cases, trace equivalence is obviously not the right equivalence to make use of.

There exist different viewpoints on what should be considered relevant to the behaviour of processes. The equivalence known as bisimulation equivalence is based on the idea that not only the traces of equivalent transition

systems should coincide, but also the stages at which the choices of different possibilities occur. Therefore, bisimulation equivalence is said to preserve the branching structure of transition systems. Here is an example of a case where apparently not only the traces matter, but also the stages at which the choices of different possibilities occur.

Example 1.5.9. We consider a split connection between nodes in a network. A split connection has one input port and two output ports. A datum that has been consumed at the input port can be delivered at either one of the output ports. That is, the choice of the output ports is resolved after the datum has been consumed. The behaviour of a split connection with input port k and output ports l and m can be described as follows. We assume a set of data D . As states of the split connection, we have $*$ and the data $d \in D$, with $*$ as initial state. As actions, we have $s_i(d)$ (send d at port i) and $r_i(d)$ (receive d at port i) for $i = k, l, m$ and $d \in D$. As transitions, we have the following:

- for each $d \in D$, a transition $* \xrightarrow{r_k(d)} d$;
- for each $i \in \{l, m\}$ and $d \in D$, transitions $d \xrightarrow{s_i(d)} *$.

The transition system for the split connection is represented graphically in Figure 1.13 for the case where $D = \{0, 1\}$. Next we consider a transition

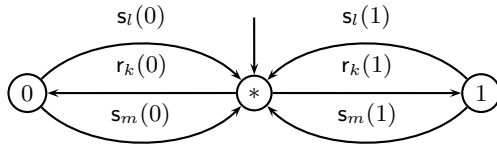


Fig. 1.13. Transition system for the split connection

system that is trace equivalent to the one just presented. As states, we have the pairs (i, d) for $i = k, l, m$ and $d \in D \cup \{*\}$, with $(k, *)$ as initial state. As actions, we still have $s_i(d)$ and $r_i(d)$ for $i = k, l, m$ and $d \in D$. As transitions, we have the following:

- for each $i \in \{l, m\}$ and $d \in D$: $(k, *) \xrightarrow{r_k(d)} (i, d)$, $(i, d) \xrightarrow{s_i(d)} (k, *)$.

This transition system is represented graphically in Figure 1.14 for the case where $D = \{0, 1\}$. This transition system does not describe the intended behaviour of the split connection correctly. A datum that has been consumed cannot be delivered at either of the output ports because the choice of the output ports is resolved at the instant that the datum is consumed. So, we

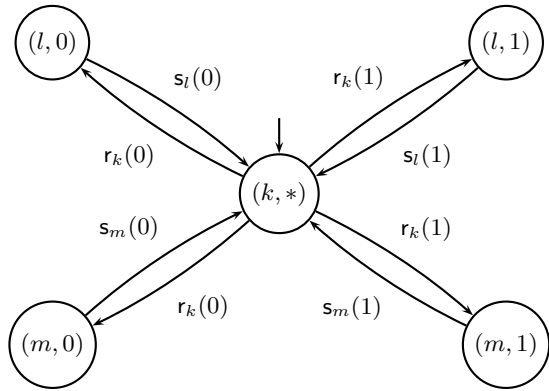


Fig. 1.14. Transition system for the split-like connection

do not want to identify this transition system with the previous one. They are not identified by bisimulation equivalence.

What is exactly meant by “the stages at which the choices of different possibilities occur” in our intuitive explanation of bisimulation equivalence becomes clear in the following informal definition. Two transition systems T and T' are bisimulation equivalent if their states can be related such that:

- the initial states are related;
- if states s_1 and s'_1 are related and in T a transition with label a is possible from s_1 to some s_2 , then in T' a transition with label a is possible from s'_1 to some s'_2 such that s_2 and s'_2 are related;
- likewise, with the role of T and T' reversed;
- if states s and s' are related and s is a successfully terminating state in T , then s' is a successfully terminating state in T' ;
- likewise, with the role of T and T' reversed.

This means that, starting from any pair of related states, T can simulate T' and that conversely T' can simulate T .

Bisimulation equivalence can also be characterized as follows: it identifies transition systems if they cannot be distinguished by any conceivable experiment with an experimenter that is only able to detect which actions are performed at any stage and whether the system has terminated successfully. The kind of identifications made by bisimulation equivalence is illustrated with the following example.

Example 1.5.10. We consider a merge connection between nodes in a network. A merge connection has two input ports and one output port. Each datum that has been consumed at one of the input ports is delivered at the

output port. The behaviour of a merge connection with input ports k and l and output port m can be described as follows. We assume a set of data D . As states, we have the pairs (i, d) for $i = k, l, m$ and $d \in D \cup \{*\}$, with $(m, *)$ as initial state. As actions, we have again $s_i(d)$ and $r_i(d)$ for $i = k, l, m$ and $d \in D$. As transitions, we have the following:

- for each $i \in \{k, l\}$ and $d \in D$: $(m, *) \xrightarrow{r_i(d)} (i, d)$, $(i, d) \xrightarrow{s_m(d)} (m, *)$.

This transition system for the merge connection is represented graphically in Figure 1.15 for the case where $D = \{0, 1\}$. Next we consider the following

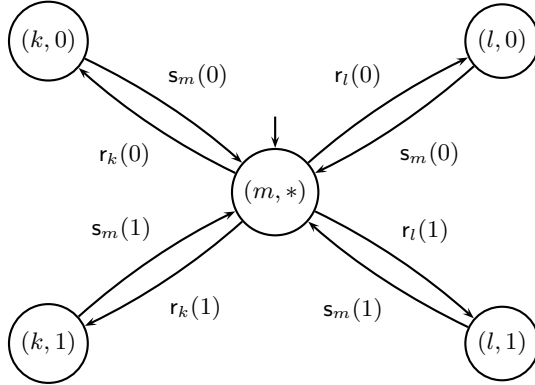


Fig. 1.15. Transition system for the merge connection

transition system. As states, we have $*$ and the data $d \in D$, with $*$ as initial state. As actions, we still have $s_i(d)$ and $r_i(d)$ for $i = k, l, m$ and $d \in D$. As transitions, we have the following:

- for each $i \in \{k, l\}$ and $d \in D$, a transition $* \xrightarrow{r_i(d)} d$;
- for each $d \in D$, a transition $d \xrightarrow{s_m(d)} *$.

This transition system is represented graphically in Figure 1.16 for the case where $D = \{0, 1\}$. This transition system describes the intended behaviour of the merge connection correctly as well. Is this transition system identified with the previous one by bisimulation equivalence? Yes, it is: relate state $(m, *)$ to state $*$ and, for each $i \in \{k, l\}$ and $d \in D$, state (i, d) to state d .

Let us now give the formal definition of bisimulation equivalence.

Definition 1.5.5. Let $T = (S, A, \rightarrow, \downarrow, s_0)$ and $T' = (S', A', \rightarrow', \downarrow', s'_0)$ be transition systems such that $A = A'$. Then a *bisimulation* B between T and T' is a binary relation $B \subseteq S \times S'$ such that the following conditions hold:

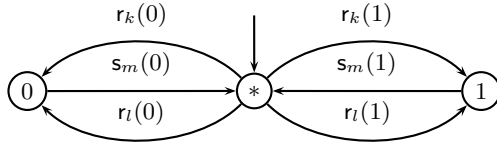


Fig. 1.16. Another transition system for the merge connection

1. $B(s_0, s'_0)$;
2. whenever $B(s_1, s'_1)$ and $s_1 \xrightarrow{a} s_2$, then there is a state s'_2 such that $s'_1 \xrightarrow{a'} s'_2$ and $B(s_2, s'_2)$;
3. whenever $B(s_1, s'_1)$ and $s'_1 \xrightarrow{a'} s'_2$, then there is a state s_2 such that $s_1 \xrightarrow{a} s_2$ and $B(s_2, s'_2)$;
4. whenever $B(s, s')$ and $s \downarrow$, then $s' \downarrow'$;
5. whenever $B(s, s')$ and $s' \downarrow'$, then $s \downarrow$.

The two transition systems T and T' are *bisimulation equivalent* (also called *bisimilar*), written $T \simeq T'$, if there exists a bisimulation B between T and T' . A bisimulation between T and T is called an *autobisimulation* on T .

Example 1.5.11. A bisimulation relation between graphical representations of transition systems is usually indicated by connecting all the states that are related by the bisimulation relation by means of a dashed line. For the bisimulation equivalent transition systems of the merge connection, this visualization is shown in Figure 1.17.

Property 1.5.9. Bisimulation equivalence is an equivalence relation.

Restriction to relations B between the reachable states of T and the reachable states of T' does not change the notion of bisimulation equivalence.

Bisimulation equivalence identifies more transition systems than isomorphy, but less than trace equivalence. In Example 1.5.9, we have seen two transition systems that are trace equivalent, but not bisimulation equivalent. The two transition systems depicted in Figure 1.18 are not isomorphic. They are, however, bisimulation equivalent.

Property 1.5.10. Any two isomorphic transition systems are also bisimulation equivalent: if $T \cong T'$, then $T \simeq T'$.

Property 1.5.11. Any two bisimilar transition systems are also trace equivalent: if $T \simeq T'$, then $T \equiv_{\text{tr}} T'$.

Let us return to the experimenter that is only able to detect which actions are performed at any stage. If performing the same experiment on a system more than once leads to the same outcome for all his (or her) experiments, the

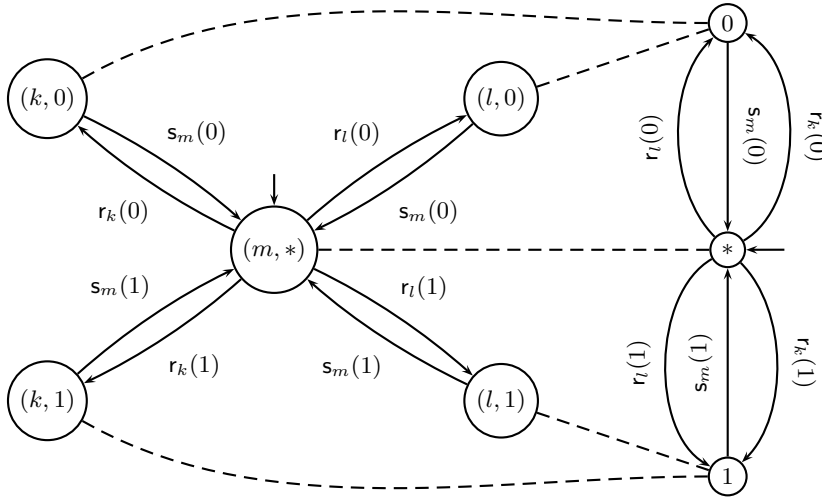


Fig. 1.17. A bisimulation relation for the merge connection transition systems

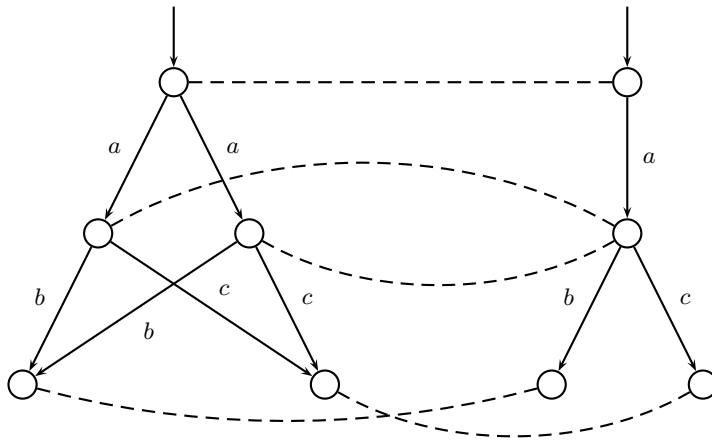


Fig. 1.18. Two transition systems that are not isomorphic, but are bisimulation equivalent

system behaves predictably. Such a system is called determinate. This is an important notion in the design of a system. In many cases, we have to arrive at a determinate system from components of which some are not determinate. This is, for example, the case with the simple data communication protocol treated in the next chapter. Here is the precise definition of determinacy.

Definition 1.5.6 (Determinacy). Let $T = (S, A, \rightarrow, \downarrow, s_0)$ be a transition system. Then T is *determinate* if the following condition holds:

whenever $s_0 \xrightarrow{\sigma} s$ and $s_0 \xrightarrow{\sigma} s'$, then there is an autobisimulation B on T such that $B(s, s')$.

For determinate transition systems trace equivalence and bisimulation equivalence coincide.

Property 1.5.12. Let $T = (S, A, \rightarrow, \downarrow, s_0)$ and $T' = (S', A', \rightarrow', \downarrow', s'_0)$ be transition systems such that $A = A'$. Then the following holds:

if T and T' are determinate, then $T \simeq T'$ if and only if $T \equiv_{\text{tr}} T'$.

A direct consequence of the above property is that for determinate transition systems, the equivalences discussed in this section, namely isomorphy, trace equivalence, and bisimulation equivalence coincide.

Property 1.5.13. For determinate transition systems T and T' the following holds: $T \cong T'$ if and only if $T \equiv_{\text{tr}} T'$ if and only if $T \simeq T'$.

The notion of determinism of a transition system is closely related to the notion of determinacy of a transition system.

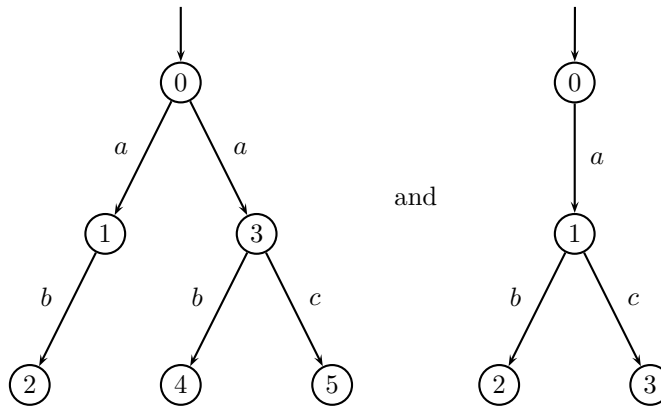
Definition 1.5.7 (Determinism). Let $T = (S, A, \rightarrow, \downarrow, s_0)$ be a transition system. Then T is *deterministic* if the following condition holds:

whenever $s_0 \xrightarrow{\sigma} s$ and $s_0 \xrightarrow{\sigma} s'$, then $s = s'$.

It is easy to see that all deterministic transition systems are determinate, but not all determinate transition systems are deterministic. One could say that a determinate transition system is deterministic up to bisimulation.

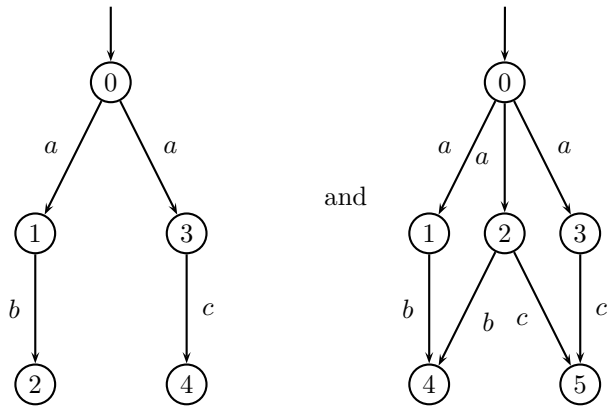
In this section and the previous one, we have shortly introduced the use of equivalences for abstraction from details of transition systems that we want to ignore. This plays a prominent part in techniques for the analysis of process behaviour. We will come back to trace and bisimulation equivalence later.

Exercise 1.5.1. Determine whether the transition systems



are language equivalent. Determine also whether they are trace equivalent, bisimulation equivalent, and isomorphic.

Exercise 1.5.2. Determine whether the transition systems



are language equivalent. Determine also whether they are trace equivalent, bisimulation equivalent, and isomorphic.

Exercise 1.5.3. Draw the transition systems $T = (S, A, \rightarrow, \downarrow, s_0)$ where

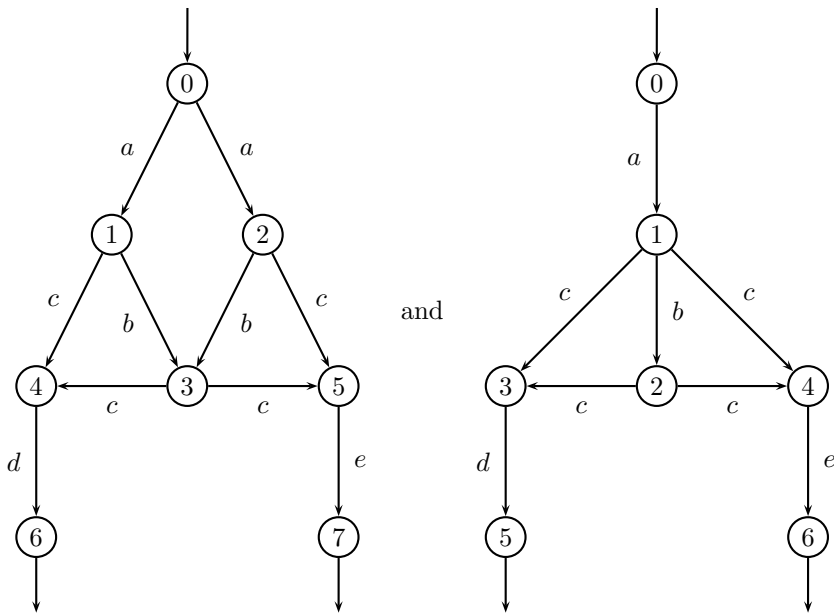
$$\begin{aligned}
 S &= \{0, 1, \dots, 7\}, \\
 A &= \{a, b, c\}, \\
 \rightarrow &= \{(0, a, 1), (1, b, 2), (2, c, 3), (0, a, 4), (4, b, 5), (5, c, 6), (5, b, 7)\}, \\
 \downarrow &= \emptyset, \\
 s_0 &= 0.
 \end{aligned}$$

and $T' = (S', A', \rightarrow', \downarrow', s'_0)$ where

$$\begin{aligned}
 S' &= \{0, 1, \dots, 4\}, \\
 A' &= \{a, b, c\}, \\
 \rightarrow' &= \{(0, a, 1), (1, b, 2), (2, c, 3), (1, b, 4)\}, \\
 \downarrow' &= \emptyset, \\
 s'_0 &= 0.
 \end{aligned}$$

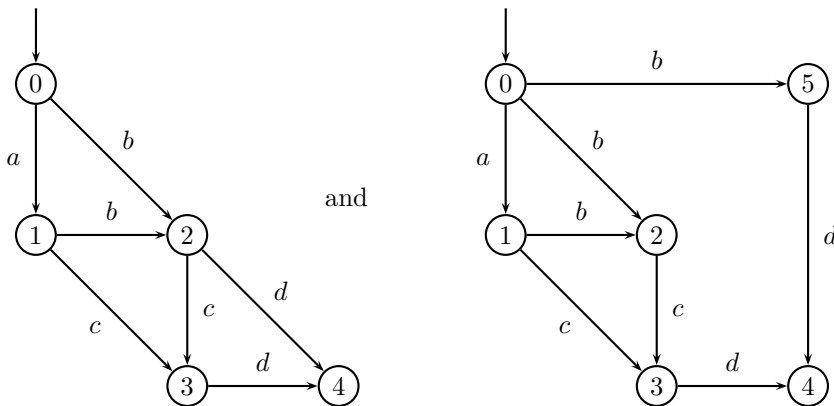
Determine whether these transition systems are trace equivalent ($T \equiv_{\text{tr}} T'$) and/or bisimulation equivalent ($T \rightleftharpoons T'$).

Exercise 1.5.4. Determine whether the transition systems



are language equivalent, trace equivalent, and/or bisimulation equivalent.

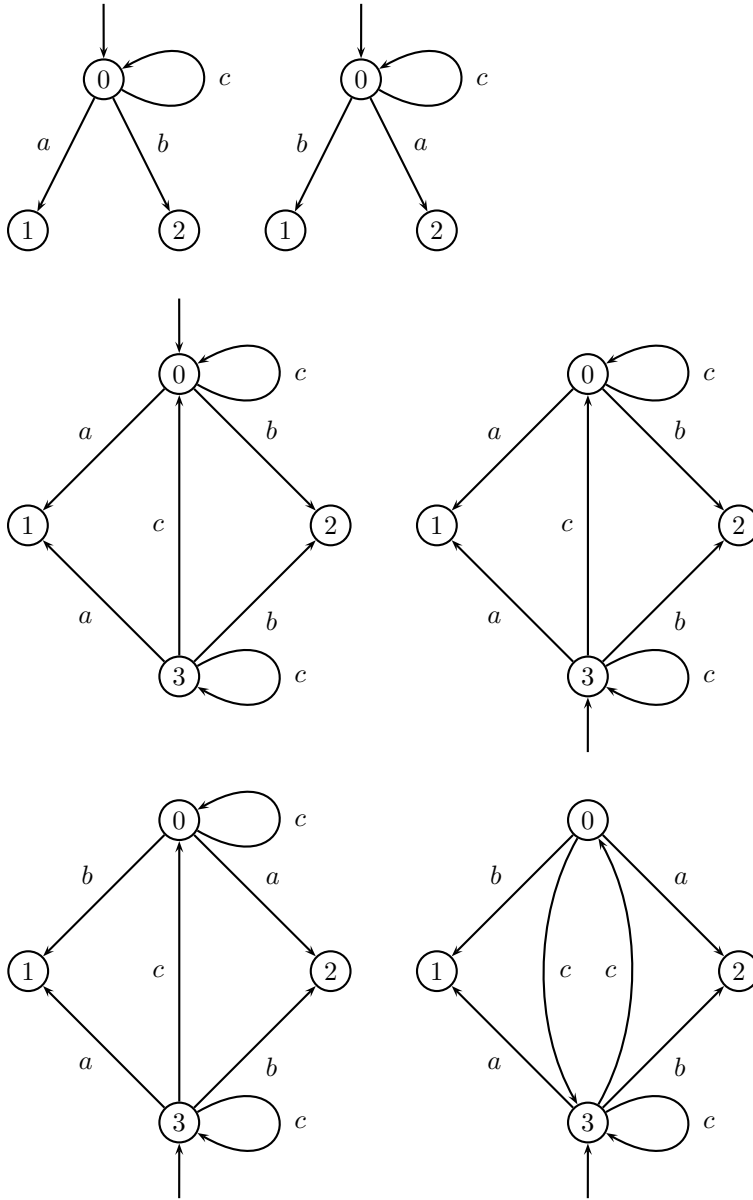
Exercise 1.5.5. Determine whether the transition systems

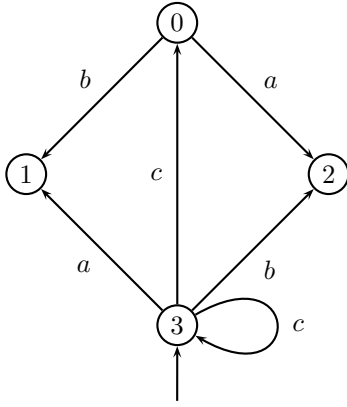


are language equivalent, trace equivalent, and/or bisimulation equivalent.

Exercise 1.5.6. Give a determinate transition system that is not deterministic.

Exercise 1.5.7. Consider the transition systems given below. Give, for each of them, the reachable states and the deadlock states. Find, for each pair of transition systems, the strongest equivalence (from: isomorphism, bisimulation equivalence, trace equivalence, and language equivalence) between them.

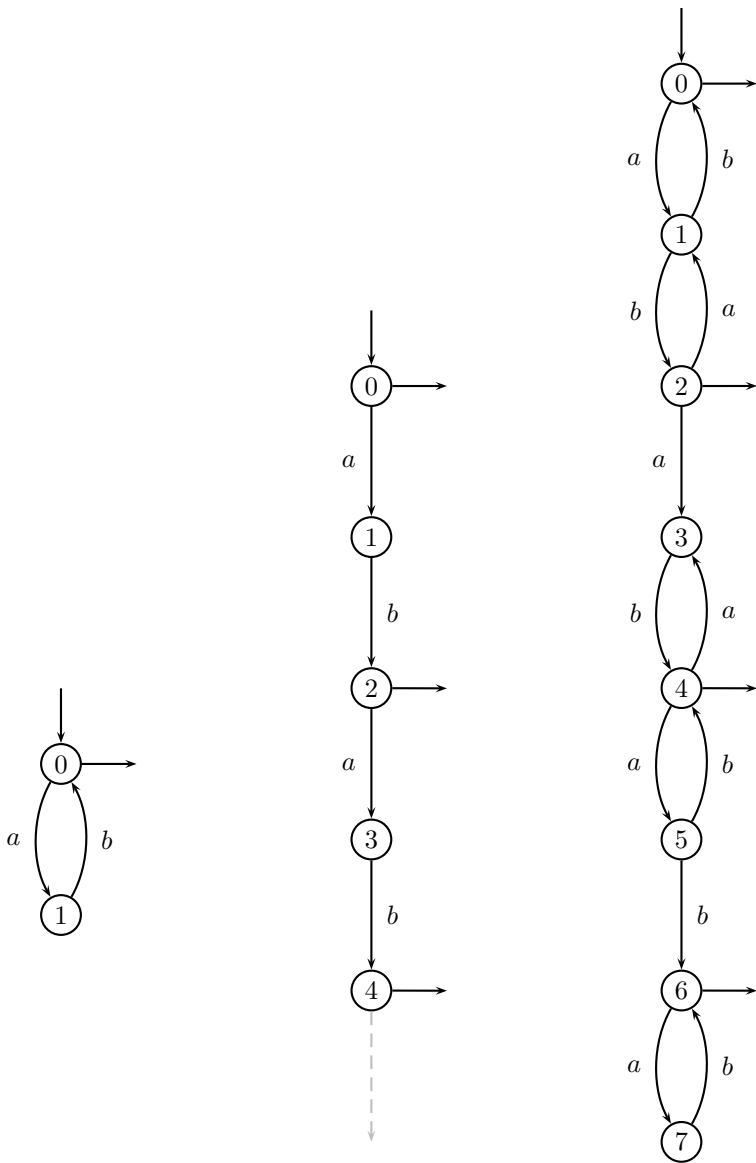




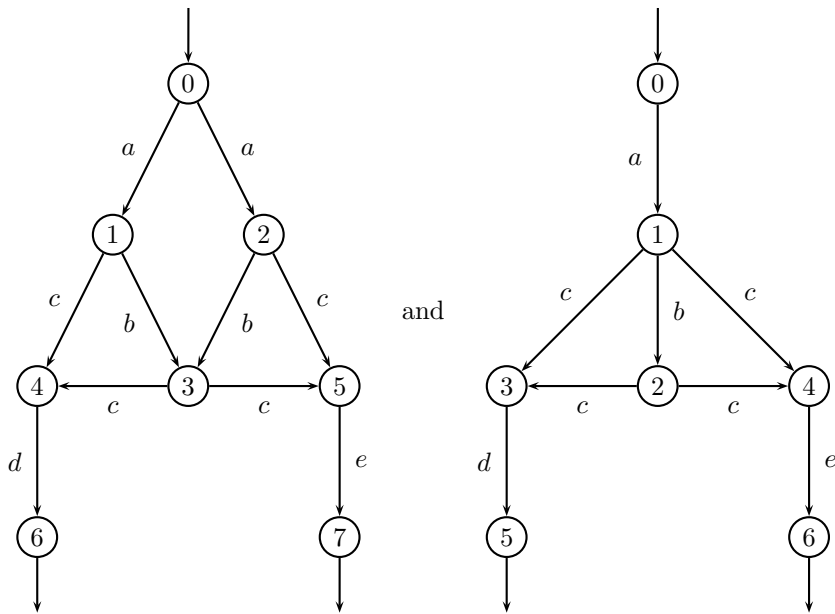
Exercise 1.5.8. Consider the transition systems given below.

1. Determine for each pair of transition systems whether these are bisimulation equivalent. If so, give a bisimulation between them; if not, give a formula in Hennessy-Milner logic that discriminates between them.
2. Give, for each of these transition systems, the maximal autobisimulation⁸. Determine whether these maximal autobisimulations are equivalence relations.
3. Let $T = (S, A, \rightarrow, \downarrow, s_0)$ be a given transition system and let B be a maximal autobisimulation on T . Then we define the B -reduction of T , notation T/B , to be the transition system $T/B = (S', A', \rightarrow', \downarrow', s'_0)$ where
 - S' is the set of equivalence classes of S under the equivalence B : $S' = S/B = \{[s]_B \mid s \in S\}$ where the equivalence class of $s \in S$ with respect to B , notation $[s]_B$, is defined as $[s]_B = \{s' \in S \mid (s, s') \in B\}$.
 - $A' = A$
 - $\rightarrow' = \{([s]_B, a, [s']_B) \mid s \xrightarrow{a} s'\}$
 - $\downarrow' = \{[s]_B \mid s \downarrow\}$
 - $s'_0 = [s_0]_B$
 Compute for each of the given transition systems, their reduction with respect to the maximal autobisimulation.
4. Prove the following proposition: for any transition system T and any maximal autobisimulation B on T , $T \rightleftharpoons T/B$.

⁸ The maximal autobisimulation of a transition system T is the largest autobisimulation. Actually, it is the union of all autobisimulations on T .



Exercise 1.5.9. Consider the following transition systems. These are not bisimulation equivalent.



Add transitions to one of these transition systems in such a way that they become bisimulation equivalent. Give the bisimulation relation that proves this.

Exercise 1.5.10 (Unbounded counter). Let the transition system $T = (S, A, \rightarrow, \downarrow, s_0)$ be given by

$$\begin{aligned}
 S &= \mathbb{N} \\
 A &= \{\text{inc}, \text{dec}\} \\
 \rightarrow &= \{(n, \text{inc}, n+1), (n+1, \text{dec}, n) \mid n \in \mathbb{N}\} \\
 \downarrow &= \emptyset \\
 s_0 &= 0
 \end{aligned}$$

Is there a finite transition system T' such that $T \Leftrightarrow T'$? If so, give such a finite transition system; if not, explain why not.

1.6 Hennessy-Milner logic

So far, we have discussed equality of transition systems. Though, in many cases, we are not so much interested in equality, but more in whether or not, a model of a real-life system satisfies some properties. Such properties are often expressed as a logical formula.

A simple, though not very expressive, logic that can be used for this purpose is Hennessy-Milner logic.

Definition 1.6.1. The formulas of Hennessy-Milner logic over a set of actions A are the following:

1. *true* and *false* are formulas,
2. \checkmark is a formula,
3. for all formulas ϕ : $\neg\phi$ is a formula,
4. for all formulas ϕ and ψ : $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \implies \psi$, and $\phi \Leftrightarrow \psi$ are formulas,
5. for any set I and formulas ϕ_i (for $i \in I$): $\bigwedge_{i \in I} \phi_i$ and $\bigvee_{i \in I} \phi_i$ are formulas,
6. for all formulas ϕ and sets of actions $K \subseteq A$: $\langle K \rangle \phi$ and $[K] \phi$ are formulas.

The collection of all formulas of Hennessy-Milner logic over A is denoted $\mathcal{HM}(A)$.

As a notational convention, for a finite set $K = \{a_1, \dots, a_n\}$, we write $[a_1, \dots, a_n] \phi$ and $\langle a_1, \dots, a_n \rangle \phi$, respectively, instead of $[\{a_1, \dots, a_n\}] \phi$ and $\langle \{a_1, \dots, a_n\} \rangle \phi$.

Now that we have introduced a means to describe properties of processes, it is time to consider the validity of such a property given a process described as a transition system. The formulas *true* and *false* are satisfied by all states and no state respectively. The formula \checkmark is satisfied if and only if the state is a successful termination state. The interpretation of the standard logical connectives is precisely as expected. The formula $[K] \phi$ is satisfied in a state s if each state that can be reached from the state s by performing an action from K satisfies the formula ϕ . Similarly, $\langle K \rangle \phi$ is satisfied if there is some state reached by an action from K that satisfies ϕ .

Definition 1.6.2 (Satisfaction). Let $T = (S, A, \rightarrow, \downarrow, s_0)$ be a transition system. The satisfaction relation $\models_T \subseteq S \times \mathcal{HM}(A)$ is defined inductively as follows: for $s \in S$

1. $s \models_T \textit{true}$ for all $s \in S$;
2. $s \models_T \textit{false}$ for no $s \in S$;
3. $s \models_T \checkmark$ if and only if $s \downarrow$;
4. $s \models_T \neg\phi$ if and only if not $s \models_T \phi$;
5. $s \models_T \phi \wedge \psi$ if and only if $s \models_T \phi$ and $s \models_T \psi$;
6. $s \models_T \phi \vee \psi$ if and only if $s \models_T \phi$ or $s \models_T \psi$;
7. $s \models_T \phi \implies \psi$ if and only if not $s \models_T \phi$, or $s \models_T \psi$;
8. $s \models_T \phi \Leftrightarrow \psi$ if and only if $s \models_T \phi$ if and only if $s \models_T \psi$;
9. $s \models_T \bigwedge_{i \in I} \phi_i$ if and only if $s \models_T \phi_i$ for all $i \in I$;
10. $s \models_T \bigvee_{i \in I} \phi_i$ if and only if $s \models_T \phi_i$ for some $i \in I$;
11. $s \models_T \langle K \rangle \phi$ if and only if $s' \models_T \phi$ for some $a \in K$ and $s' \in S$ such that $s \xrightarrow{a} s'$;
12. $s \models_T [K] \phi$ if and only if $s' \models_T \phi$ for all $a \in K$ and $s' \in S$ such that $s \xrightarrow{a} s'$.

A transition system $T = (S, A, \rightarrow, \downarrow, s_0)$ is said to satisfy a formula ϕ if and only if $s_0 \models_T \phi$. This is denoted by $\models_T \phi$.

Given a transition system T and a state s , we can express that an a -action can be performed from this state as $\langle a \rangle true$. Similarly, expressing that a certain action a cannot be performed, is achieved through the formula $[a] false$.

Example 1.6.1 (Bounded counter). Consider the bounded counter from Example 1.1.2 with bound $k = 3$. The formula $[inc] true$ expresses that initially, the bounded counter can execute the action inc and that in each state resulting from executing an inc -action, the formula $true$ must hold. In this case, there is one possible inc -transition from state 0, i.e. the initial state. As the formula $true$ holds in each state, obviously it also holds in each state reached by performing the inc -action. Hence, the bounded counter satisfies the property that initially an increment action can be performed: $\models_T [inc] true$.

Initially, it is impossible for the bounded counter to perform a decrement action. That this property is captured by the formula $[dec] false$ can be seen as follows. If there would be a dec -transition from state 0, leading to a state s , then the formula $false$ must be satisfied in this state s . But, by definition, the formula $false$ is not satisfied by each state. Thus the formula $[dec] false$ excludes the possibility of a dec -action to be performed.

If we consider the same two formulas but now with respect to the modulo counter from the same example, we observe that both formulas are satisfied by the modulo counter. Hence, one could say that the bounded counter and the modulo counter are different processes since they have different properties. We return to this subject shortly.

Example 1.6.2 (Traces and terminating traces). In the previous section we have seen that one way of looking at processes is through the traces of such a process. The formulas of Hennessy-Milner logic are capable of expressing the property that a certain sequence of actions $a_1 \cdots a_n$ is among the traces of a process through the formula $\langle a_1 \rangle \langle a_2 \rangle \cdots \langle a_n \rangle true$. Similarly, the formula $\langle a_1 \rangle \langle a_2 \rangle \cdots \langle a_n \rangle \checkmark$ states that the process has a terminating trace $a_1 \cdots a_n$.

Example 1.6.3 (Necessity of action execution). Very often it is desirable to not only express the possibility of certain actions occurring but also the necessity of their execution. Suppose we want to express that a certain action a has to take place. This can be captured by expressing that action a can be performed and by expressing that all actions except for a cannot be performed and that there can be no successful termination. Hence, the formula

$$\langle a \rangle true \wedge [A \setminus \{a\}] false \wedge \neg \checkmark$$

expresses that the action a must happen next.

This formula can easily be generalized to express that an action from a certain set $K \subseteq A$ must occur next:

$$\langle K \rangle true \wedge [A \setminus K] false \wedge \neg \checkmark.$$

Example 1.6.4 (Deadlock). For a given transition system T and a state s from T , the formula $[A]false \wedge \neg\checkmark$ and the formula $\neg(\langle A \rangle true \vee \checkmark)$ both express that this state represents a deadlock. Hence, absence of deadlock is specified by means of $\neg([A]false \wedge \neg\checkmark)$ or $\langle A \rangle true \vee \checkmark$.

Example 1.6.5. Consider the transition systems given in Figure 1.19. Formally

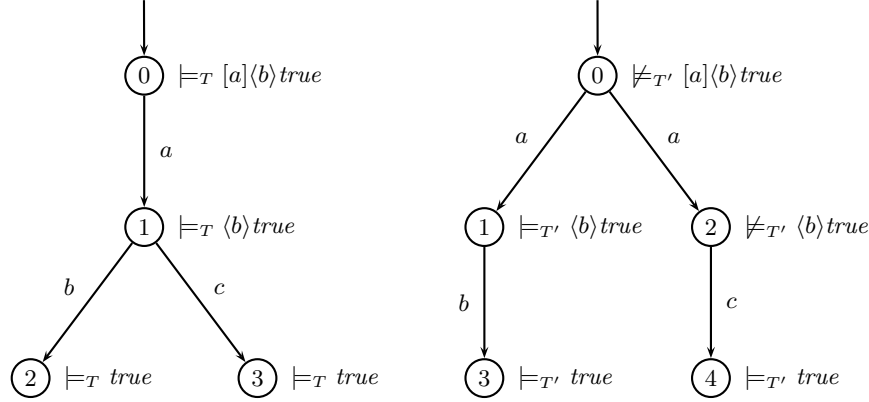


Fig. 1.19. Example of transition systems with different properties

they can be represented by transition systems $T = (S, A, \rightarrow, s_0)$ and $T' = (S', A', \rightarrow', s'_0)$ where

$$\begin{aligned} S &= \{0, 1, 2, 3\}, \\ A &= \{a, b, c\}, \\ \rightarrow &= \{(0, a, 1), (1, b, 2), (1, c, 3)\}, \\ s_0 &= 0 \end{aligned}$$

and

$$\begin{aligned} S' &= \{0, 1, 2, 3, 4\}, \\ A' &= \{a, b, c\}, \\ \rightarrow' &= \{(0, a, 1), (0, a, 2), (1, b, 3), (2, c, 4)\}, \\ s'_0 &= 0. \end{aligned}$$

Observe that these transition systems are trace equivalent ($T_1 \equiv_{tr} T_2$) and not bisimilar ($T_1 \not\equiv T_2$). Consider the formula $[a]\langle b \rangle true$, which, in words, states “after the execution of an a -transition, there is the possibility to execute a b -transition”. This property is satisfied by transition system T , but not by transition system T' : $\models_T [a]\langle b \rangle true$ and $\not\models_{T'} [a]\langle b \rangle true$. In Figure 1.19 the (relevant) formulas that are satisfied by the different states are listed.

In the previous section, processes represented by transition systems have been compared to each other by considering the transitions that can or cannot be performed. A different way of comparing transition systems is through the properties that they satisfy. It is very natural to consider two transition systems equivalent if they satisfy the same properties. The following theorem states that two transition systems are bisimulation equivalent if and only if they satisfy the same properties expressed by means of Hennessy-Milner logic.

Theorem 1.6.1. *Let $T = (S, A, \rightarrow, s_0)$ and $T' = (S', A', \rightarrow', s'_0)$ be transition systems. Then we have $T \simeq T'$ if and only if for all formulas $\phi \in \mathcal{HM}(A \cup A')$: $\models_T \phi$ if and only if $\models_{T'} \phi$.*

Exercise 1.6.1. Give a formula that discriminates the transition systems given in Exercise 1.5.1.

Exercise 1.6.2. Give a formula that discriminates the transition systems given in Exercise 1.5.2.

Exercise 1.6.3. Draw the transition systems $T = (S, A, \rightarrow, s_0)$ where

$$\begin{aligned} S &= \{0, 1, \dots, 8\}, \\ A &= \{a, b, \dots, f\}, \\ \rightarrow &= \{(0, a, 1), (1, b, 2), (1, c, 3), (3, d, 4), \\ &\quad (0, a, 5), (5, c, 6), (6, e, 7), (5, f, 8)\}, \\ s_0 &= 0, \end{aligned}$$

and $T' = (S', A', \rightarrow', s'_0)$ where

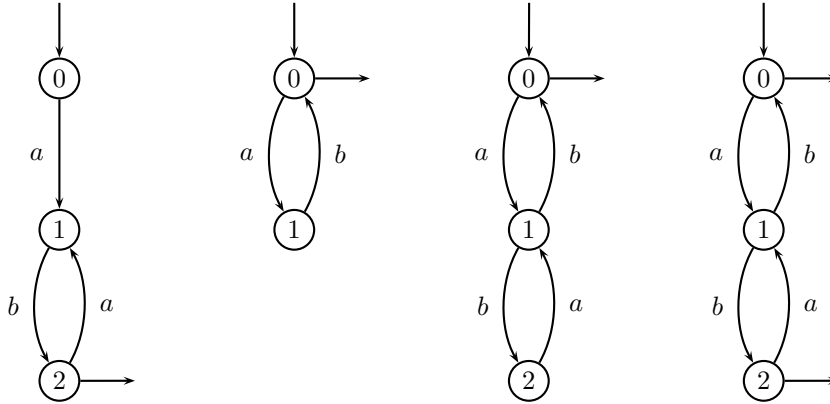
$$\begin{aligned} S' &= \{0, 1, \dots, 8\}, \\ A' &= \{a, b, \dots, f\}, \\ \rightarrow' &= \{(0, a, 1), (1, b, 2), (1, c, 3), (3, e, 4), \\ &\quad (0, a, 5), (5, c, 6), (6, d, 7), (5, f, 8)\}, \\ s'_0 &= 0. \end{aligned}$$

Determine whether these transition systems are trace equivalent ($T \equiv_{\text{tr}} T'$) and/or bisimulation equivalent ($T \simeq T'$). If they are not bisimulation equivalent give a formula ϕ that is satisfied by T , but not by T' and a formula ψ that is satisfied by T' , but not by T .

Exercise 1.6.4. Give a formula that discriminates the transition systems given in Exercise 1.5.4.

Exercise 1.6.5. Give a formula that discriminates the transition systems given in Exercise 1.5.5.

Exercise 1.6.6. Consider the following transition systems.



Determine for each pair of transition systems whether these are bisimulation equivalent. If so, give a bisimulation between them; if not, give a formula in Hennessy-Milner logic that discriminates between them.

Exercise 1.6.7. Consider the transition systems T' and T'' where $T' = (S', A', \rightarrow', \downarrow', s'_0)$ is given by

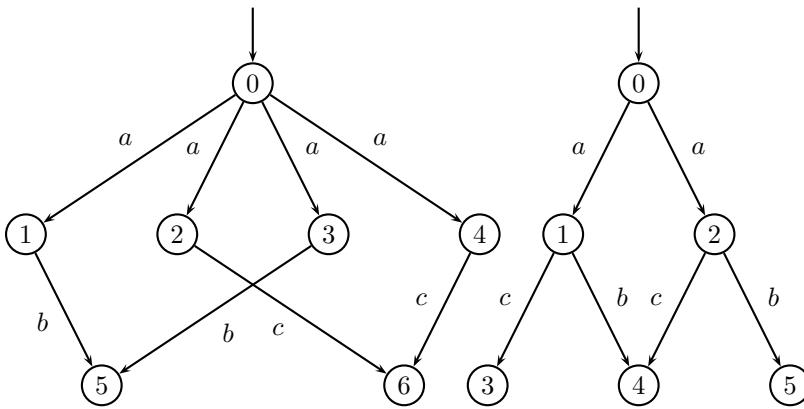
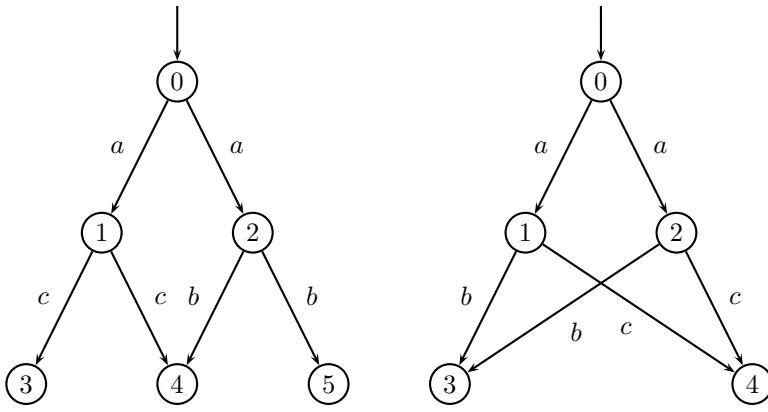
$$\begin{aligned} S' &= \mathbb{R}^{\geq 0} \\ A' &= \{\text{half}\} \\ \rightarrow' &= \{(2x, \text{half}, x) \mid x \in \mathbb{R}^{\geq 0}\} \\ \downarrow' &= \emptyset \\ s'_0 &= 1 \end{aligned}$$

and $T'' = (S'', A'', \rightarrow'', \downarrow'', s''_0)$ is given by

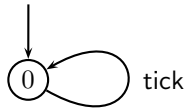
$$\begin{aligned} S'' &= \mathbb{R}^{\geq 0} \\ A'' &= \{\text{half}\} \\ \rightarrow'' &= \{(2x, \text{half}, x) \mid x \in \mathbb{R}^{> 0}\} \\ \downarrow'' &= \emptyset \\ s''_0 &= 1. \end{aligned}$$

Determine whether these transition systems are bisimulation equivalent. If so, give a bisimulation between them; if not, give a formula in Hennessy-Milner logic that discriminates between them.

Exercise 1.6.8. Consider the transition systems given below. Determine for each pair of transition systems whether these are bisimulation equivalent. If so, give a bisimulation between them; if not, give a formula in Hennessy-Milner logic that discriminates between them.



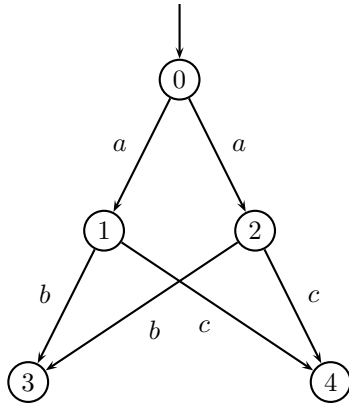
Exercise 1.6.9. Consider the following transition system.



Verify whether or not the following Hennessy-Milner formula are satisfied by this process:

1. $\langle \text{tick} \rangle \text{true}$
2. $[\text{tick}] \text{false}$
3. $[\text{tick}] (\langle \text{tick} \rangle \text{true} \wedge [\text{tock}] \text{false})$
4. $[\text{tick}] (\langle \text{tock} \rangle \text{true} \vee [\text{tick}] \text{false})$

Exercise 1.6.10. Consider the following transition system.



Verify whether or not the following Hennessy-Milner formula are satisfied by this process:

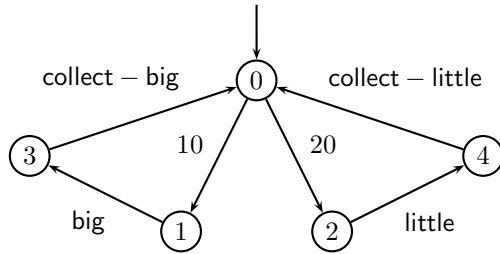
1. $[a]([a]true \vee [a]false)$
2. $[a]([b]\checkmark \wedge [c]true)$
3. $[a]([a]\checkmark \wedge [c]true)$

Exercise 1.6.11. Consider the example of the bounded counter with bound 3 (Example 1.1.2). Give a formula in Hennessy-Milner logic that expresses that it is impossible to perform 4 consecutive increments of the counter.

Exercise 1.6.12. Consider the example of a split connection (Example 1.5.9).

1. Give a formula in Hennessy-Milner logic that expresses that any datum put into the split connection via its input port, will be transferred to one of the output ports next.
2. Give a formula in Hennessy-Milner logic that expresses that it is impossible to have two consecutive input actions.
3. Similarly as the previous question, only this case for two consecutive output actions.

Exercise 1.6.13. Consider the following simple vending machine. First, a coin (either 10 or 20 eurocents) has to be deposited. If a 10 eurocent coin has been deposited a button for requesting a small cup of coffee can be depressed. Alternatively, after depositing 20 eurocents, a big coffee can be ordered. The right amount of coffee is dispensed by the vending machine.



Give formula in Hennessy-Milner logic for the following properties and verify whether these are satisfied by the vending machine:

1. a button cannot be depressed (before money is deposited);
2. after 20 eurocent is deposited, the little button cannot be depressed whereas the big one can;
3. after a coin has been deposited no other coin may be deposited;
4. after a coin is deposited and a button is depressed, an item can be collected.

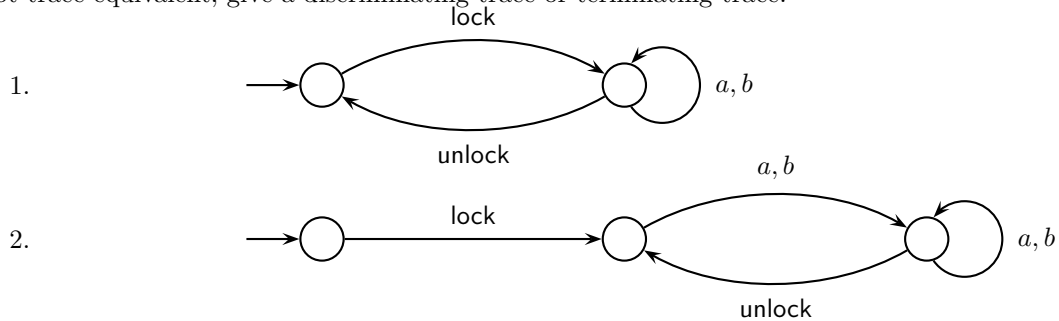
Exercise 1.6.14. Consider the following two transition systems.

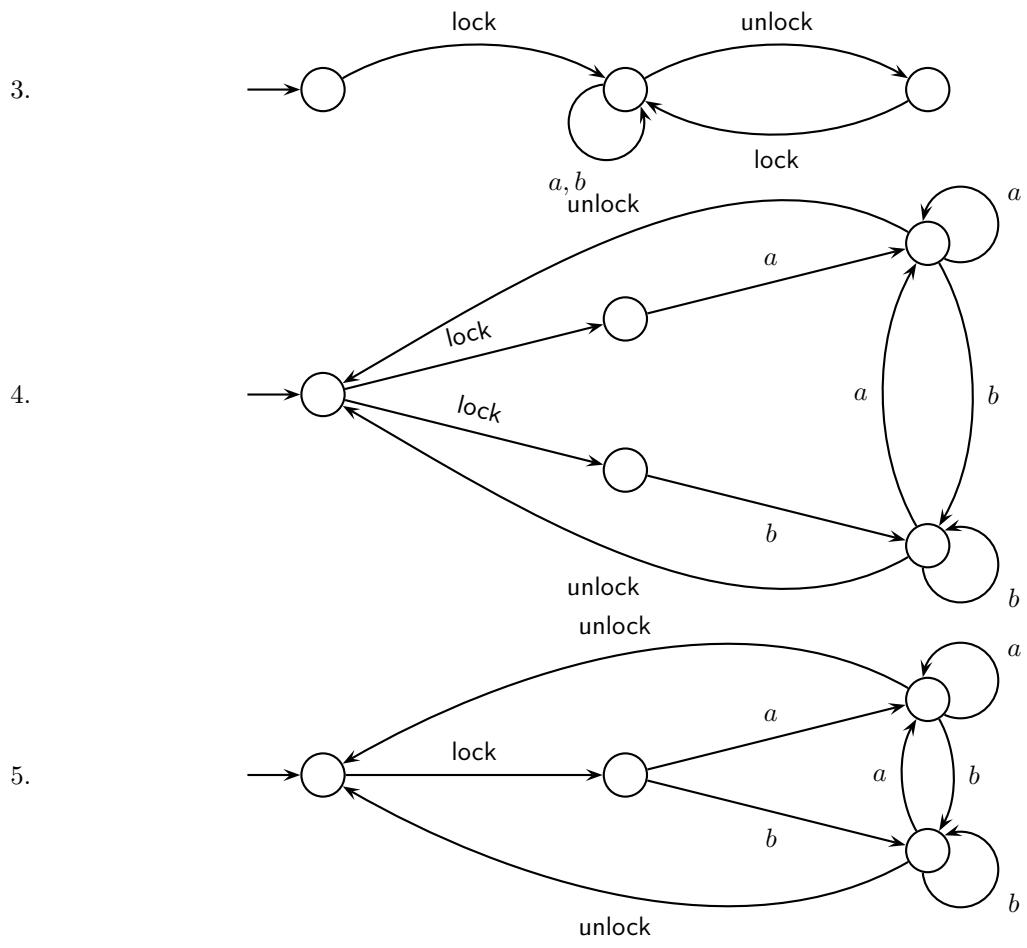


Is it possible to give a Hennessy-Milner formula that is satisfied by one of these transition systems and not by the other? If so, give such a formula; if not, prove this.

Exercise 1.6.15.

Determine for any two of the following transition systems whether they are trace equivalent, bisimulation equivalent, or neither of those. In case they are bisimulation equivalent, give a bisimulation relation that proves this. In case they are not bisimulation equivalent but are trace equivalent, give a formula in Hennessy-Milner logic that discriminates between them. In case they are not trace equivalent, give a discriminating trace or terminating trace.





2. Concurrency and Interaction

Complex systems are generally composed of several components that act concurrently and interact with each other. This chapter deals with the issue of concurrency and interaction by introducing the notion of parallel composition of transition systems. First of all, we explain informally what parallel composition of transition systems is and give a simple example of its use in describing process behaviour (Section 2.1). After that, we define the notion of parallel composition of transition systems in a mathematically precise way (Section 2.2).

2.1 Informal explanation

Sending a message to another component and receiving a message from another component are typical examples of the kinds of actions that are performed by a component of a system in order to interact with other components that act concurrently. Synchronous communication of a message between two components is a typical example of an interaction that takes place when a send action of one component and a matching receive action of the other component are performed synchronously. When two actions are performed synchronously, those actions cannot be observed separately. Therefore, the intuition is that only one action is left when two actions are performed synchronously. For instance, when a send action and a matching receive action are performed synchronously, only a communication action can be observed. It does not have to be the case that any two actions can be performed synchronously. Usually, two actions can be performed synchronously only if they can establish an interaction. That is, for example, not the case for two send actions.

Now consider the use of transition systems in describing the behaviour of systems. In the case where a system is composed of components that act concurrently and interact with each other, we would like to reflect the composition in the description of the behaviour of the system. That is, we would like to use transition systems to describe the behaviour of the components and to be able to describe the behaviour of the whole system by expressing that its transition system is obtained from the transition systems describing the behaviour of the components by applying a certain operation to those

transition systems. Parallel composition of transition systems as introduced in this chapter serves this purpose. The intuition is that the parallel composition of two transition systems T and T' can perform at each stage any action that T can perform next, any action that T' can perform next, and any action that results from synchronously performing an action that T can perform next and an action that T' can perform next. Parallel composition does not prevent actions that can be performed synchronously from being performed on their own. In order to prevent certain actions from being performed on their own, we introduce a separate operation on transition systems, called encapsulation. The reason why parallel composition and encapsulation are not combined in a single operation will be explained later at the end of Section 2.2. Here is an example of the use of parallel composition and encapsulation in describing the behaviour of systems composed of components that act concurrently and interact with each other.

Example 2.1.1 (Bounded buffers). We consider the system composed of two bounded buffers, buffer 1 and buffer 2 with capacities l_1 and l_2 respectively, where each datum removed from the data kept in buffer 1 is simultaneously added to the data kept in buffer 2. In this way, data from buffer 1 is transferred to buffer 2. We start from the bounded buffers from Example 1.1.3. In the case of buffer 1, we rename the actions $\text{add}(d)$ and $\text{rem}(d)$ into $\text{add}_1(d)$ and $\text{rem}_1(d)$, respectively. In the case of buffer 2, we rename the actions $\text{add}(d)$ and $\text{rem}(d)$ into $\text{add}_2(d)$ and $\text{rem}_2(d)$, respectively. In this way, we can distinguish between the action of adding a datum to the data kept in one buffer and the action of adding the same datum to the data kept in the other buffer, as well as between the action of removing a datum from the data kept in one buffer and the action of removing the same datum from the data kept in the other buffer.

The renamings yield the following. As states of bounded buffer i , $i = 1, 2$, with capacity l_i , we have the sequences of data of which the length is not greater than l_i . As initial state, we have the empty sequence. As actions, we have $\text{add}_i(d)$ and $\text{rem}_i(d)$ for each datum d . As transitions of bounded buffer i , we have the following:

- for each datum d and each state σ that has a length less than l_i , a transition $\sigma \xrightarrow{\text{add}_i(d)} d\sigma$;
- for each datum d and each state σd , a transition $\sigma d \xrightarrow{\text{rem}_i(d)} \sigma$.

In Figure 2.1, the transition systems for the buffers are given for the case that both have capacity 1. In the case where, for each datum d , the actions $\text{rem}_1(d)$ and $\text{add}_2(d)$ can be performed synchronously, and $\text{trf}(d)$ (transfer d) is the action left when these actions are performed synchronously, parallel composition of buffer 1 and buffer 2 results in the following transition system. As states, we have pairs (σ_1, σ_2) where σ_i ($i = 1, 2$) is a sequence of data of which the length is not greater than l_i . State (σ_1, σ_2) is the state in which the sequence of data σ_i ($i = 1, 2$) is kept in buffer i . As initial state, we have

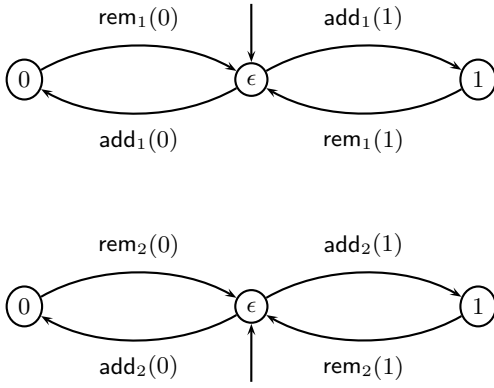


Fig. 2.1. Transition systems for the bounded buffers

(ϵ, ϵ) . As actions, we have $\text{add}_i(d)$, $\text{rem}_i(d)$ and $\text{trf}(d)$ for each datum d and $i = 1, 2$. As transitions, we have the following:

- for each datum d and each state (σ_1, σ_2) with the length of σ_1 less than l_1 , a transition $(\sigma_1, \sigma_2) \xrightarrow{\text{add}_1(d)} (d\sigma_1, \sigma_2)$;
- for each datum d and each state (σ_1, σ_2) with the length of σ_2 less than l_2 , a transition $(\sigma_1, \sigma_2) \xrightarrow{\text{add}_2(d)} (\sigma_1, d\sigma_2)$;
- for each datum d and each state $(\sigma_1 d, \sigma_2)$, a transition $(\sigma_1 d, \sigma_2) \xrightarrow{\text{rem}_1(d)} (\sigma_1, \sigma_2)$;
- for each datum d and each state $(\sigma_1, \sigma_2 d)$, a transition $(\sigma_1, \sigma_2 d) \xrightarrow{\text{rem}_2(d)} (\sigma_1, \sigma_2)$;
- for each datum d and each state $(\sigma_1 d, \sigma_2)$ with the length of σ_2 less than l_2 , a transition $(\sigma_1 d, \sigma_2) \xrightarrow{\text{trf}(d)} (\sigma_1, d\sigma_2)$.

This transition system is represented graphically in Figure 2.2 for the case where $l_1 = l_2 = 1$ and the only data involved are the natural numbers 0 and 1. For each datum d , actions $\text{rem}_1(d)$ and $\text{add}_2(d)$ can still be performed on their own. Encapsulation with respect to these actions prevents them from being performed on their own, i.e. it results in the following transition system. We have the same states as before. As actions, we have $\text{add}_1(d)$, $\text{rem}_2(d)$ and $\text{trf}(d)$ for each datum d . As transitions, we have the following:

- for each datum d and each state (σ_1, σ_2) with the length of σ_1 less than l_1 , a transition $(\sigma_1, \sigma_2) \xrightarrow{\text{add}_1(d)} (d\sigma_1, \sigma_2)$;
- for each datum d and each state $(\sigma_1, \sigma_2 d)$, a transition $(\sigma_1, \sigma_2 d) \xrightarrow{\text{rem}_2(d)} (\sigma_1, \sigma_2)$;
- for each datum d and each state $(\sigma_1 d, \sigma_2)$ with the length of σ_2 less than l_2 , a transition $(\sigma_1 d, \sigma_2) \xrightarrow{\text{trf}(d)} (\sigma_1, d\sigma_2)$.

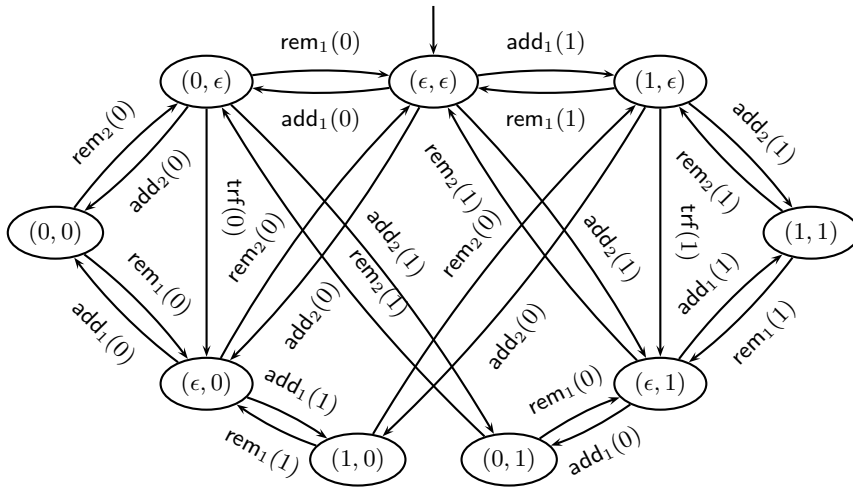


Fig. 2.2. Transition system for parallel composition of bounded buffers

This transition system is represented graphically in Figure 2.3 for the case where $l_1 = l_2 = 1$ and the only data involved are the natural numbers 0 and 1. So encapsulation is needed to prevent that the actions $rem_1(d)$ and $add_2(d)$

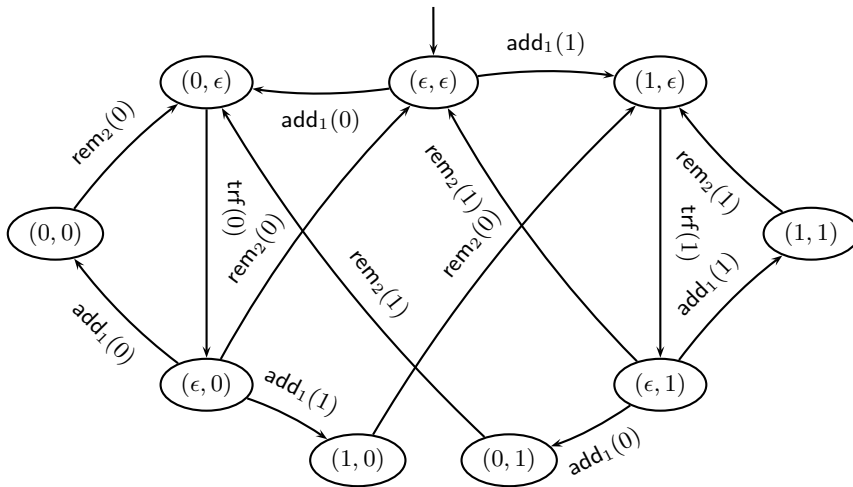


Fig. 2.3. Transition system for encapsulation of two parallel bounded buffers

do not lead to transfer of datum d from buffer 1 to buffer 2. The transition system obtained from the two bounded buffers by parallel composition and encapsulation would be bisimulation equivalent (see Section 1.5) to a bounded buffer with capacity $l_1 + l_2$ if we could abstract from the internal transfer

actions $\text{trf}(d)$. Abstraction from internal actions is one of the issues treated in the remaining chapters of this book.

Although systems composed of bounded buffers that act concurrently and interact with each other as described above actually arise in computer-based systems, they are not regarded as typical examples of real-life computer-based systems composed of components that act concurrently and interact with each other.

2.2 Formal definitions

With the previous section, we have prepared the way for the formal definitions of the notions of parallel composition of transition systems and encapsulation of a transition system.

Whether two actions can be performed synchronously, and if so what action is left when they are performed synchronously, is mathematically represented by a communication function. Here is the definition of a communication function.

Definition 2.2.1. Let A be a set of actions. A *communication function* on A is a partial function $\gamma : A \times A \rightarrow A$ satisfying for $a, b, c \in A$:

- if $\gamma(a, b)$ is defined, then $\gamma(b, a)$ is defined and $\gamma(a, b) = \gamma(b, a)$;
- if $\gamma(a, b)$ and $\gamma(\gamma(a, b), c)$ are defined, then $\gamma(b, c)$ and $\gamma(a, \gamma(b, c))$ are defined and $\gamma(\gamma(a, b), c) = \gamma(a, \gamma(b, c))$.

The reason for the first condition is evident: there should be no difference between performing a and b synchronously and performing b and a synchronously. The reason for the second condition is essentially the same, but for the case where more than two actions can be performed synchronously. Let us give an example to illustrate that it is straightforward to define the communication function needed.

Example 2.2.1. We consider again the parallel composition of bounded buffers from Example 2.1.1. In that example, for each datum d , the actions $\text{rem}_1(d)$ and $\text{add}_2(d)$ can be performed synchronously, and $\text{trf}(d)$ is the action left when these actions are performed synchronously. This is simply represented by the communication function γ defined such that $\gamma(\text{rem}_1(d), \text{add}_2(d)) = \gamma(\text{add}_2(d), \text{rem}_1(d)) = \text{trf}(d)$ for each datum d , and it is undefined otherwise.

Let us now look at the formal definitions of parallel composition and encapsulation.

Definition 2.2.2. Let $T = (S, A, \rightarrow, \downarrow, s_0)$ and $T' = (S', A', \rightarrow', \downarrow', s'_0)$ be transition systems. Let γ be a communication function on a set of actions that includes $A \cup A'$. The *parallel composition* of T and T' under γ , written $T \parallel_\gamma T'$, is the transition system $(S'', A'', \rightarrow'', \downarrow'', s''_0)$ where

- $S'' = S \times S'$;
- $A'' = A \cup A' \cup \{\gamma(a, a') \mid a \in A, a' \in A', \gamma(a, a') \text{ is defined}\}$;
- \rightarrow'' is the smallest subset of $S'' \times A'' \times S''$ such that:
 - if $s_1 \xrightarrow{a} s_2$ and $s' \in S'$, then $(s_1, s') \xrightarrow{a}'' (s_2, s')$;
 - if $s'_1 \xrightarrow{b'} s'_2$ and $s \in S$, then $(s, s'_1) \xrightarrow{b'}'' (s, s'_2)$;
 - if $s_1 \xrightarrow{a} s_2$, $s'_1 \xrightarrow{b'} s'_2$ and $\gamma(a, b)$ is defined, then $(s_1, s'_1) \xrightarrow{\gamma(a, b)}'' (s_2, s'_2)$;
- $\downarrow'' = \downarrow \times \downarrow'$;
- $s''_0 = (s_0, s'_0)$.

In the above definition, the transition relation \rightarrow'' could have been given by means of

$$\begin{aligned} \rightarrow'' = & \{((s_1, s'), a, (s_2, s')) \mid (s_1, a, s_2) \in \rightarrow, s' \in S'\} \\ & \cup \{(s, s'_1), b, (s, s'_2) \mid (s'_1, b, s'_2) \in \rightarrow', s \in S\} \\ & \cup \{(s_1, s'_1), \gamma(a, b), (s_2, s'_2) \mid (s_1, a, s_2) \in \rightarrow, \\ & \qquad \qquad \qquad (s'_1, b, s'_2) \in \rightarrow', \\ & \qquad \qquad \qquad \gamma(a, b) \text{ defined}\} \end{aligned}$$

as well.

In describing systems that are composed in parallel of more than two subsystems, we use the convention of association to the left for parallel composition to reduce the number of parentheses, e.g. we write $T_1 \parallel_{\gamma} T_2 \parallel_{\gamma'} T_3$ for $(T_1 \parallel_{\gamma} T_2) \parallel_{\gamma'} T_3$. Justification for this convention will be provided later.

Example 2.2.2. The transition systems for the two buffers with capacity 1 are given by $T = (S, A, \rightarrow, \downarrow, s_0)$ where

$$\begin{aligned} S &= \{\epsilon, 0, 1\}, \\ A &= \{\text{add}_1(d), \text{rem}_1(d) \mid d \in D\}, \\ \rightarrow &= \{(\epsilon, \text{add}_1(d), d), (d, \text{rem}_1(d), \epsilon) \mid d \in D\}, \\ \downarrow &= \emptyset, \\ s_0 &= \epsilon, \end{aligned}$$

and $T' = (S', A', \rightarrow', \downarrow', s'_0)$ where

$$\begin{aligned} S' &= \{\epsilon, 0, 1\}, \\ A' &= \{\text{add}_2(d), \text{rem}_2(d) \mid d \in D\}, \\ \rightarrow' &= \{(\epsilon, \text{add}_2(d), d), (d, \text{rem}_2(d), \epsilon) \mid d \in D\}, \\ \downarrow' &= \emptyset, \\ s'_0 &= \epsilon, \end{aligned}$$

respectively. We compute the parallel composition of T and T' under γ , the communication function from Example 2.2.1. The set of states of $T \parallel_{\gamma} T'$ is obtained by constructing the Cartesian product of the sets of states of T and T' :

$$S'' = S \times S' = \{(\epsilon, \epsilon), (0, \epsilon), (1, \epsilon), (\epsilon, 0), (0, 0), (1, 0), (\epsilon, 1), (0, 1), (1, 1)\}.$$

The set of actions is the union of the sets of actions of T and T' and additionally any actions that result from communication between the transition systems:

$$\begin{aligned} A'' &= A \cup A' \cup \{\gamma(a, a') \mid a \in A, a' \in A', \gamma(a, a') \text{ defined}\} \\ &= \{\text{add}_1(d), \text{rem}_1(d), \text{add}_2(d), \text{rem}_2(d), \text{trf}(d) \mid d \in D\}. \end{aligned}$$

The transitions are obtained by copying the transitions of transition system T and those of transition system T' . These are given by

$$\begin{aligned} &\{((s_1, s'), a, (s_2, s')) \mid (s_1, a, s_2) \in \rightarrow, s' \in S'\} \\ &= \{((\epsilon, s'), \text{add}_1(d), (d, s')), ((d, s'), \text{rem}_1(d), (\epsilon, s')) \mid d \in D, s' \in S'\} \end{aligned}$$

and

$$\begin{aligned} &\{((s, s'_1), a, (s, s'_2)) \mid (s'_1, a, s'_2) \in \rightarrow', s \in S\} \\ &= \{((s, \epsilon), \text{add}_2(d), (s, d)), ((s, d), \text{rem}_2(d), (s, \epsilon)) \mid d \in D, s \in S\} \end{aligned}$$

respectively. To this the transitions that result from communication have to be added. The only possible communication is between the actions $\text{rem}_1(d)$ and $\text{add}_2(d)$ (for each $d \in D$). The only state (in T) from which $\text{rem}_1(d)$ is possible is the state d and the only state (in T') from which $\text{add}_2(d)$ is possible is the state ϵ . Hence, the following transitions result from communication:

$$\begin{aligned} &\{((s_1, s'_1), \gamma(a, a'), (s_2, s'_2)) \mid (s_1, a, s_2) \in \rightarrow, (s'_1, a', s'_2) \in \rightarrow', \\ &\quad \gamma(a, a') \text{ defined}\} \\ &= \{((d, \epsilon), \text{trf}(d), (\epsilon, d)) \mid d \in D\}. \end{aligned}$$

So, we have obtained

$$\begin{aligned} \rightarrow'' &= \{((\epsilon, s'), \text{add}_1(d), (d, s')), ((d, s'), \text{rem}_1(d), (\epsilon, s')) \mid d \in D, s' \in S'\} \\ &\cup \{((s, \epsilon), \text{add}_2(d), (s, d)), ((s, d), \text{rem}_2(d), (s, \epsilon)) \mid d \in D, s \in S\} \\ &\cup \{((d, \epsilon), \text{trf}(d), (\epsilon, d)) \mid d \in D\}. \end{aligned}$$

As T and T' do not have successfully terminating states, also their parallel composition does not have successfully terminating states:

$$\downarrow'' = \downarrow \times \downarrow' = \emptyset \times \emptyset = \emptyset.$$

The initial state is the pair of initial states of T and T' :

$$s''_0 = (s_0, s'_0).$$

The reader can easily verify that the transition system depicted in Figure 2.2 corresponds to the transition system T'' as defined here.

If parallel composition is applied to transition systems T and T' where all states are reachable, then the states of their parallel composition will result in a transition system where all states are reachable as well.

Property 2.2.1. Let T and T' be transition systems such that all states of T and T' are reachable. Then, for arbitrary communication function γ , we have $\text{reach}(T \parallel_{\gamma} T') = \text{reach}(T) \times \text{reach}(T')$.

Similarly, if parallel composition is applied to transition systems T and T' where all actions are used in reaching states, then the parallel composition of such transition systems will result in a transition system where all actions are used in reaching states.

Property 2.2.2. Let T and T' be transition systems such that $\text{act}(T) = \text{act}(\text{red}(T))$ and $\text{act}(T') = \text{act}(\text{red}(T'))$. Then, for arbitrary communication function γ , we have $\text{act}(T \parallel_{\gamma} T') = \text{act}(\text{red}(T)) \cup \text{act}(\text{red}(T')) \cup \{\gamma(a, a') \mid a \in \text{act}(T), a' \in \text{act}(T'), \gamma(a, a') \text{ defined}\}$.

This means that by applying parallel composition no states and actions become unreachable. This is not the case for all compositions on transition systems that we encounter. For example, in encapsulating certain actions it is possible that some states and actions that used to be reachable are not so anymore. Such operations can be defined in two equally reasonable ways.

- A definition such that the resulting transition system is reduced.
- A definition such that the resulting transition system is not necessarily reduced. It may contain unreachable states and actions.

From a readability point of view, the second type of definition is easier. In this book, we always give definitions of compositions of the second type.

Definition 2.2.3 (Encapsulation). Let $T = (S, A, \rightarrow, \downarrow, s_0)$ be a transition system. Let H be a set of actions. The *encapsulation* of T with respect to H , written $\partial_H(T)$, is the transition system $(S', A', \rightarrow', \downarrow', s'_0)$ where

- $S' = S$;
- $A' = A$;
- $\rightarrow' = \rightarrow \cap (S \times (A \setminus H) \times S)$;
- $\downarrow' = \downarrow$;
- $s'_0 = s_0$.

Example 2.2.3. The encapsulation of the two parallel bounded buffers is obtained from the transition system T'' from Example 2.2.2 as follows. The states and actions of $\partial_H(T'')$ with $H = \{\text{rem}_1(d), \text{add}_2(d) \mid d \in D\}$ are simply the states and actions of T'' , respectively. The transitions of $\partial_H(T'')$ are obtained by removing all transitions with an action from H from the transitions of T'' . The successfully terminating states and the initial state remain the same as well. The resulting transition system is depicted in Figure 2.3.

In many applications, $\gamma(\gamma(a, b), c)$ is undefined for all $a, b, c \in A$. That case is called *handshaking communication*. We introduce some standardized terminology and notation for handshaking communication. Transition systems

send, receive and communicate data at *ports*. If a port is used for communication between two transition systems, it is called *internal*. Otherwise, it is called *external*. We write:

- $s_i(d)$ for the action of sending datum d at port i ;
- $r_i(d)$ for the action of receiving datum d at port i ;
- $c_i(d)$ for the action of communicating datum d at port i .

Assuming a set of data D , the communication function is defined such that

$$\gamma(s_i(d), r_i(d)) = \gamma(r_i(d), s_i(d)) = c_i(d)$$

for all $d \in D$, and it is undefined otherwise.

It is important to remember that handshaking communication is just one kind of communication. It is not required that $\gamma(\gamma(a, b), c)$ is undefined for all $a, b, c \in A$. Here is an example of another kind of communication.

Example 2.2.4. We consider a kind of communication in which three transition systems participate. A communication of this kind takes place by synchronously performing one send action and two matching receive actions. Using a notation which is reminiscent of the standardized notation for handshaking communication, this ternary kind of communication can be represented by a communication function as follows. Assuming a set of data D , the communication function is defined such that

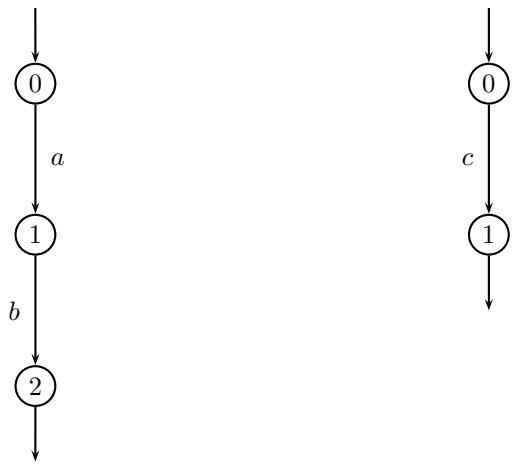
$$\begin{aligned} \gamma(r_i(d), r_i(d)) &= rr_i(d) , \\ \gamma(s_i(d), r_i(d)) &= \gamma(r_i(d), s_i(d)) = sr_i(d) , \\ \gamma(s_i(d), rr_i(d)) &= \gamma(rr_i(d), s_i(d)) = c_i(d) , \\ \gamma(sr_i(d), r_i(d)) &= \gamma(r_i(d), sr_i(d)) = c_i(d) , \end{aligned}$$

for all $d \in D$, and it is undefined otherwise. The actions $sr_i(d)$ and $rr_i(d)$ represent the possible partial communications.

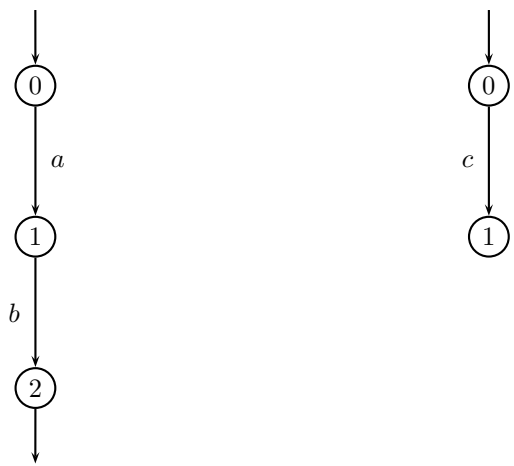
An important thing to note about the kind of communication treated in the preceding example is the following. If parallel composition and encapsulation were combined in a single operation that prevents actions that can be performed synchronously from being performed on their own, this kind of communication would be excluded.

Exercise 2.2.1. Let γ be a communication function such that $\gamma(a, b)$ is undefined for all actions a and b . Compute the parallel composition of the following pairs of transition systems.

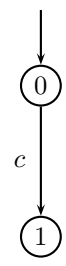
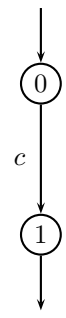
- 1.

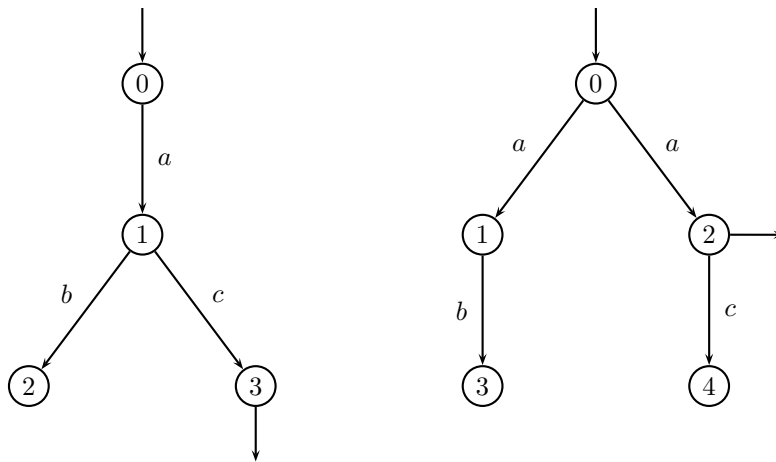


2.



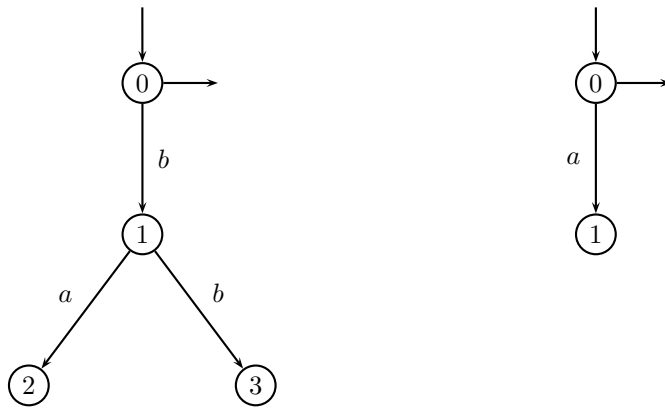
3.



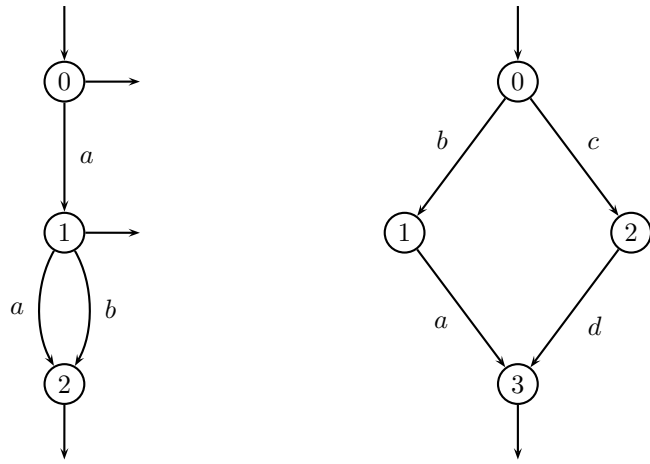


Exercise 2.2.2. Let γ be a communication function such that $\gamma(a, b) = c$ and γ is undefined otherwise. Compute the parallel composition of the following transition systems.

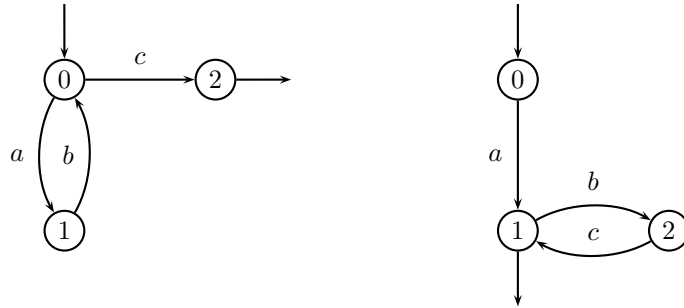
1.



2.

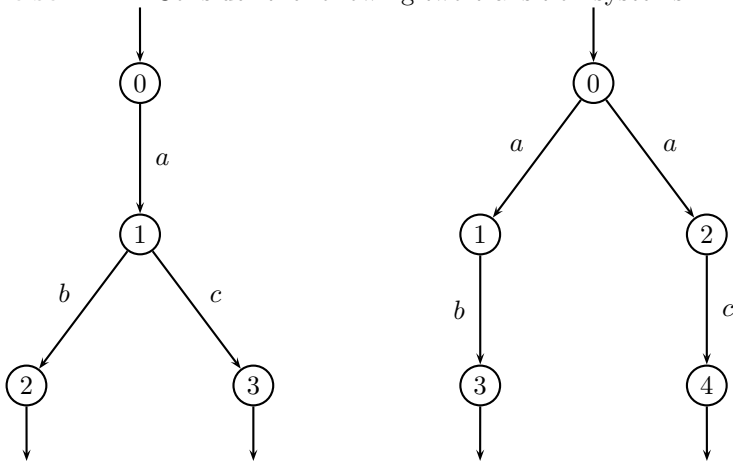


3.



Exercise 2.2.3. Compute, for each of the transition systems from the previous two exercises, their encapsulation with respect to the action b . Do the same, for the transition systems that result from the parallel compositions.

Exercise 2.2.4. Consider the following two transition systems.



These transition systems are language equivalent and trace equivalent.

1. Compute, for each of these, the encapsulation with respect to action c .
2. Are the resulting transition systems language equivalent?
3. Are the resulting transition systems trace equivalent?

Exercise 2.2.5. Suppose that we want to describe a communication protocol for the exchange of data from a set D from a *sender* to a *receiver*. The sender repeats the following behaviour. It receives the datum to be communicated, say d , from the environment through an action $r_1(d)$. Upon receipt of such a datum, the sender sends the datum to the receiver by means of the action $s_2(d)$. The sender then waits for the receipt of an acknowledgement $r_2(\text{ack})$ from the receiver of the datum.

We consider three different receiver processes.

- Receiver process 1: upon receipt of a datum ($r_2(d)$), the receiver first sends the datum to its environment ($s_3(d)$) and then sends an acknowledgement to the sender ($s_2(\text{ack})$). This behaviour is repeated indefinitely.
 - Receiver process 2: upon receipt of a datum ($r_2(d)$), the receiver first sends an acknowledgement to the sender ($s_2(\text{ack})$) and then sends the datum to its environment ($s_3(d)$). This behaviour is repeated indefinitely.
 - Receiver process 3: upon receipt of a datum ($r_2(d)$), the receiver sends an acknowledgement to the sender ($s_2(\text{ack})$) and sends the datum to its environment ($s_3(d)$). The order in which these two actions take place is unspecified and hence non-deterministic. This behaviour is repeated indefinitely.
1. Give a transition system for the sender process and the three versions of the receiver process. Also draw these transition systems for the case that $D = \{0, 1\}$.
 2. For each combination of the sender process and a receiver process compute their parallel composition where the communication function is given by $\gamma(s_2(\text{ack}), r_2(\text{ack})) = c_2(\text{ack})$ and $\gamma(s_2(d), r_2(d)) = c_2(d)$ for each $d \in D$ and γ is undefined otherwise. Draw the resulting transition systems for the case that $D = \{0, 1\}$.
 3. For each of the above parallel compositions compute the encapsulation with respect to the set of actions $H = \{s_2(d), r_2(d) \mid d \in D \cup \{\text{ack}\}\}$. Draw the resulting transition systems for the case that $D = \{0, 1\}$.
 4. Explain the differences in the transition systems that result from parallel composition and encapsulation.

Exercise 2.2.6. Suppose that we are given a communication function γ . Prove that from the conditions that are satisfied by γ it follows that: if $\gamma(a, b)$ and $\gamma(\gamma(a, b), c)$ are defined, then $\gamma(a, c)$ and $\gamma(\gamma(a, c), b)$ are defined and $\gamma(\gamma(a, b), c) = \gamma(\gamma(a, c), b)$.

Exercise 2.2.7. We define a transition system T to have deadlock, notation $\text{deadlock}(T)$, if and only if it has a deadlock state. Prove the following statements, or give a counterexample:

1. if $T \cong T'$ then $deadlock(T)$ if and only if $deadlock(T')$;
2. if $T \equiv_1 T'$ then $deadlock(T)$ if and only if $deadlock(T')$;
3. if $T \equiv_{tr} T'$ then $deadlock(T)$ if and only if $deadlock(T')$;
4. if $T \Leftrightarrow T'$ then $deadlock(T)$ if and only if $deadlock(T')$.

Exercise 2.2.8. Prove the following properties for arbitrary transition systems T and arbitrary sets of actions H_1 and H_2 :

1. $\partial_\emptyset(T) = T$;
2. $\partial_{H_1}(\partial_{H_2}(T)) = \partial_{H_1 \cup H_2}(T)$.

3. Composition

In Chapter 2, we have seen that, by means of parallel composition, a transition system can be composed of others that act concurrently and interact with each other. This is not the only conceivable way of composition. This chapter treats several basic ways in which transition systems can be composed of others that do not interact with each other. Alternative composition is used to describe that a transition system is composed of two others that act the one or the other. Sequential composition is used to describe that a transition system is composed of two others that act successively. Iteration is used to describe that the behaviour of a transition system is repeated a number of times. Many transition systems can be composed using these three ways of composition. Thus, they support mastering the complexity of large transition systems. First of all, we explain informally what alternative composition, sequential composition and iteration are, and give simple examples of their use in describing process behaviour (Section 3.1). After that, we define alternative composition, sequential composition and iteration in a mathematically precise way (Section 3.2).

3.1 Informal explanation

The alternative composition of two transition systems T and T' is a transition system describing that there is a choice between the behaviour described by T and the behaviour described by T' . The choice is resolved at the instant that one of them performs its first action or decides to terminate successfully. The sequential composition of two transition systems T and T' is a transition system describing that the behaviour described by T and the behaviour described by T' follow each other. The iteration of transition system T is a transition system describing that initially there is a choice between the behaviour described by T and successful termination, and upon successful termination of T there is this choice again. Often, we need to describe that a transition system simply acts repeatedly for ever. The no-exit iteration of transition system T is a transition system describing that initially there is the behaviour described by T , and upon successful termination of T the behaviour is again as initially. Here are a couple of examples.

Example 3.1.1. We consider the simple telephone system from Example 1.1.1. Recall that in this telephone system each telephone is provided with a process, called its basic call process, to establish and maintain connections with other telephones. Actions of this process include receiving an off-hook or on-hook signal from the telephone, receiving a dialed number from the telephone, sending a signal to start or to stop emitting a dial tone, ring tone or ring-back tone to the telephone, and receiving an alert signal from another telephone – indicating an incoming call. Initially, there is a choice between the following two alternatives:

- receiving an off-hook signal from the telephone followed by a process of which the first action is sending a signal to start emitting a dial tone to the telephone;
- receiving an alert signal from another telephone followed by a process of which the first action is sending a signal to start emitting a ring tone to the telephone.

In either case the basic call process goes back to waiting for another off-hook or alert signal after the call is terminated. Therefore, the behaviour of the basic call process of a telephone can be described as the no-exit iteration of a process that is itself the alternative composition of two subprocesses, one reacting to an off-hook signal sent to the basic call process and the other reacting to an alert signal sent to the basic call process. The first one of these subprocesses first goes through a dialling phase and after that through a calling phase. So, the behaviour of this process can itself be described as the sequential composition of a subprocess for the dialling phase and a subprocess for the calling phase. And so forth.

Example 3.1.2. In order to control telephone answering, the control component of an answering machine has to communicate with the recorder component of the answering machine, the telephone network, and the telephone connected with the answering machine. When an incoming call is detected, the answering is not started immediately:

- if the incoming call is broken off or the receiver of the telephone is lifted within a certain period, answering is discontinued;
- otherwise, an off-hook signal is issued to the network when this period has elapsed and after that a pre-recorded message is played.

Upon termination of the message, the recorder is started and a beep signal is issued to the network. The recorder is stopped when:

- either the call is broken off;
- or a certain time period has passed in the case where the call has not been broken off earlier.

Thereafter, an on-hook signal is issued to the network. The behaviour of the control component can be described as the no-exit iteration of a process

that is itself the sequential composition of three subprocesses, one checking whether the receiver is not lifted when an incoming call is detected, one controlling the answering with the pre-recorded message, and one controlling the recording of a message from the caller. Each of these subprocesses must respond properly if the call is broken off prematurely. Therefore, the behaviour of each of them can be described as an alternative composition with one of the alternatives reacting to signals indicating that the call is broken off prematurely.

3.2 Formal definition of the compositions

We will always consider two transition systems the same if they are isomorphic. Because of this, the disjointness requirement on the sets of states that occur in the definitions of alternative composition and sequential composition given below does not cause any loss of generality. Moreover, it does not matter that an arbitrary fresh initial state is chosen in the case of alternative composition: up to isomorphism the result is independent of the particular choice.

Unreachable states, and transitions between them, are never really relevant to the behaviour described by the transition system. For example, transition systems that differ only with respect to unreachable states are isomorphic. In fact, we are only interested in the reachable part of transition systems. Let us now look at the formal definitions of alternative composition, sequential composition, and iteration.

Definition 3.2.1 (Alternative composition). Let $T = (S, A, \rightarrow, \downarrow, s_0)$ and $T' = (S', A', \rightarrow', \downarrow', s'_0)$ be transition systems such that $S \cap S' = \emptyset$. The *alternative composition* of T and T' , written $T + T'$, is the transition system $(S'', A'', \rightarrow'', \downarrow'', s''_0)$ where

- $S'' = \{s''_0\} \cup S \cup S'$;
- $A'' = A \cup A'$;
- \rightarrow'' is the smallest subset of $S'' \times A''_\tau \times S''$ such that:
 - for all $s \in S$ and $a \in A_\tau$: if $s_0 \xrightarrow{a} s$, then $s''_0 \xrightarrow{a}'' s$;
 - for all $s' \in S'$ and $a \in A'_\tau$: if $s'_0 \xrightarrow{a}' s'$, then $s''_0 \xrightarrow{a}'' s'$;
 - for all $s_1, s_2 \in S$ and $a \in A_\tau$: if $s_1 \xrightarrow{a} s_2$, then $s_1 \xrightarrow{a}'' s_2$;
 - for all $s'_1, s'_2 \in S'$ and $a \in A'_\tau$: if $s'_1 \xrightarrow{a}' s'_2$, then $s'_1 \xrightarrow{a}'' s'_2$;
- \downarrow'' is the smallest subset of S'' such that:
 - for all $s \in S$: if $s \downarrow$, then $s \downarrow''$;
 - for all $s' \in S'$: if $s' \downarrow'$, then $s' \downarrow''$;
 - if $s_0 \downarrow$ or $s'_0 \downarrow'$, then $s''_0 \downarrow''$;
- $s''_0 \notin S \cup S'$.

The following things should be noted about the definition of alternative composition. The alternative composition of transition systems T and T' has a

fresh initial state $s_0'' \notin S \cup S'$. This fresh initial state adopts the transitions from the initial state of T and the transitions from the initial state of T' . However, the fresh initial state does not replace the initial states of T and T' . Thus, transitions to the initial state of T or T' do not lead to transitions to the fresh initial state. The latter transitions would imply that the choice, that should be there only initially, could come back later. Here is an example to illustrate that it is quite natural to look at certain real-life processes as the alternative composition of other processes.

Example 3.2.1. We consider a simple railroad crossing controller. An approach signal is sent to the controller as soon as a train passes a detector placed backward from the gate. An exit signal is sent to the controller as soon as the train passes another detector placed forward from the gate. The controller is able to receive approach and exit signals from the train detectors at any time. When the controller receives an approach signal, a lower signal must be sent to the gate. When the controller receives an exit signal, a raise signal must be sent to the gate. Suppose that A and E are the transition systems describing the behaviours of the subprocesses dedicated to receiving and handling an approach signal and an exit signal, respectively, in the case where the signal is received at the beginning of a cycle of the controller, i.e. when there is no previous signal being handled. Then the behaviour of one cycle of the controller is described by $A + E$.

Let us also give an example illustrating the details of alternative composition.

Example 3.2.2. We assume a set of data D , and two input ports k and l . For $d \in D$, let $R_k(d)$ and $R_l(d)$ be the transition systems $(S, A, \rightarrow, \downarrow, s_0)$ and $(S', A', \rightarrow', \downarrow', s_0')$ where

$$\begin{aligned} S &= \{(k, *), (k, d)\}, & S' &= \{(l, *), (l, d)\}, \\ A &= \{r_k(d)\}, & A' &= \{r_l(d)\}, \\ \rightarrow &= \{(k, *) \xrightarrow{r_k(d)} (k, d)\}, & \rightarrow' &= \{(l, *) \xrightarrow{r_l(d)} (l, d)\}, \\ \downarrow &= \{(k, d)\}, & \downarrow' &= \{(l, d)\}, \\ s_0 &= (k, *), & s_0' &= (l, *). \end{aligned}$$

The transition system $R_i(d)$ is capable of receiving d at port i and then terminating successfully ($i = k, l$). The alternative composition $R_k(d) + R_l(d)$ is the transition system $(S'', A'', \rightarrow'', \downarrow'', s_0'')$ where

$$\begin{aligned} S'' &= \{(*, *), (k, *), (k, d), (l, *), (l, d)\}, \\ A'' &= \{r_k(d), r_l(d)\}, \\ \rightarrow'' &= \{(*, *) \xrightarrow{r_k(d)} (k, d), (*, *) \xrightarrow{r_l(d)} (l, d), \\ &\quad (k, *) \xrightarrow{r_k(d)} (k, d), (l, *) \xrightarrow{r_l(d)} (l, d)\}, \\ \downarrow'' &= \{(k, d), (l, d)\}, \\ s_0'' &= (*, *). \end{aligned}$$

This transition system is capable of receiving datum d at port k or l and then terminating successfully. Observe that this transition system has two unreachable states: $(k, *)$ and $(l, *)$. The alternative composition of $R_k(d)$ and $R_l(d)$ is represented graphically in Figure 3.1. As always the unreachable states are not represented graphically.

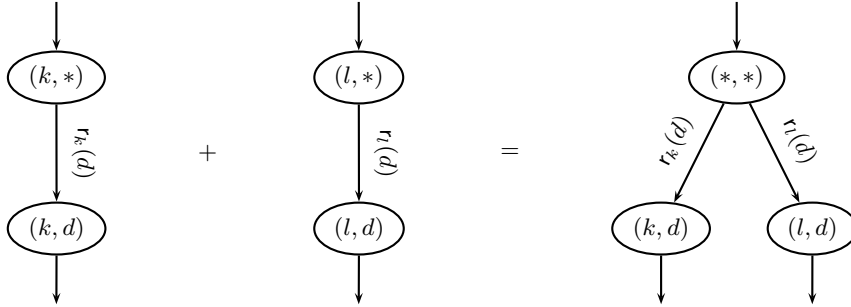


Fig. 3.1. Alternative composition of $R_k(d)$ and $R_l(d)$

Definition 3.2.2 (Sequential composition). Let $T = (S, A, \rightarrow, \downarrow, s_0)$ and $T' = (S', A', \rightarrow', \downarrow', s'_0)$ be transition systems such that $S \cap S' = \emptyset$. The *sequential composition* of T and T' , written $T \cdot T'$, is the transition system $(S'', A'', \rightarrow'', \downarrow'', s''_0)$ where

- $S'' = S \cup S'$;
- $A'' = A \cup A'$;
- \rightarrow'' is the smallest subset of $S'' \times A''_\tau \times S''$ such that:
 - for all $s_1, s_2 \in S$ and $a \in A_\tau$: if $s_1 \xrightarrow{a} s_2$, then $s_1 \xrightarrow{a}'' s_2$;
 - for all $s'_1, s'_2 \in S'$ and $a \in A'_\tau$: if $s'_1 \xrightarrow{a}' s'_2$, then $s'_1 \xrightarrow{a}'' s'_2$;
 - for all $s' \in S'$ and $a \in A'_\tau$: if $s'_0 \xrightarrow{a}' s'$, then $s \xrightarrow{a}'' s'$ for every $s \in \downarrow$;
- \downarrow'' is the smallest subset of S'' such that:
 - for all $s' \in S'$: if $s' \downarrow'$, then $s' \downarrow''$;
 - for all $s \in S$: if $s'_0 \downarrow'$ and $s \downarrow$, then $s \downarrow''$;
- $s''_0 = s_0$.

The definition of sequential composition is the first definition of a way in which transition systems can be composed where successfully terminating states are relevant to the transitions of the resulting transition system. Notice that, in the sequential composition of transition systems T and T' , transitions from the initial state of T' and successful termination options from the initial state of T' are copied to transitions and successful termination options involving the successfully terminating states of T . Here is an example to illustrate that it is quite natural to look at certain real-life processes as the sequential composition of other processes.

Example 3.2.3. We look again at the railroad crossing controller from Example 3.2.1. Suppose that $R(\text{appr})$ and $R(\text{exit})$ are the transition systems describing the behaviours of the subprocesses dedicated to receiving an approach signal and an exit signal, respectively. Suppose that D and U are the transition systems describing the behaviours of the subprocesses dedicated to handling an approach signal and an exit signal, respectively, that is received at the beginning of a cycle of the controller. Then the behaviour of one cycle of the controller is described by $(R(\text{appr}) \cdot D) + (R(\text{exit}) \cdot U)$.

Let us also give an example illustrating the details of sequential composition.

Example 3.2.4. We assume a set of data D and one output port m . For $d \in D$, let $S_m(d)$ be the transition system $(S', A', \rightarrow', \downarrow', s'_0)$ where

$$\begin{aligned} S' &= \{(m, d), (m, *)\}, \\ A' &= \{s_m(d)\}, \\ \rightarrow' &= \{(m, d) \xrightarrow{s_m(d)} (m, *)\}, \\ \downarrow' &= \{(m, *)\}, \\ s'_0 &= (m, d). \end{aligned}$$

The transition system $S_m(d)$ is capable of sending datum d at port m and then terminating successfully. Let $R_k(d) + R_l(d)$ be as defined in Example 3.2.2. The sequential composition $(R_k(d) + R_l(d)) \cdot S_m(d)$ is the transition system $(S'', A'', \rightarrow'', \downarrow'', s''_0)$ where

$$\begin{aligned} S'' &= \{(*, *), (k, *), (l, *), (k, d), (l, d), (m, d), (m, *)\}, \\ A'' &= \{r_k(d), r_l(d), s_m(d)\}, \\ \rightarrow'' &= \{(*, *) \xrightarrow{r_k(d)} (k, d), (*, *) \xrightarrow{r_l(d)} (l, d), \\ &\quad (k, *) \xrightarrow{r_k(d)} (k, d), (l, *) \xrightarrow{r_l(d)} (l, d), \\ &\quad (m, d) \xrightarrow{s_m(d)} (m, *), (k, d) \xrightarrow{s_m(d)} (m, *), (l, d) \xrightarrow{s_m(d)} (m, *)\}, \\ \downarrow'' &= \{(m, *)\}, \\ s''_0 &= (*, *). \end{aligned}$$

This transition system is capable of receiving datum d at port k or l , next sending datum d at port m and then terminating successfully. The sequential composition of $R_k(d) + R_l(d)$ and $S_m(d)$ is represented graphically in Figure 3.2. Observe that the initial state of transition system $S_m(d)$ (i.e., the state (m, d)) is not reachable in the resulting transition system.

Definition 3.2.3 (Iteration). Let $T = (S, A, \rightarrow, \downarrow, s_0)$ be a transition system. The *iteration* of T , written T^* , is the transition system $(S', A', \rightarrow', \downarrow', s'_0)$ where

- $S' = \{s'_0\} \cup S$;
- $A' = A$;
- \rightarrow' is the smallest subset of $S' \times A'_\tau \times S'$ such that:

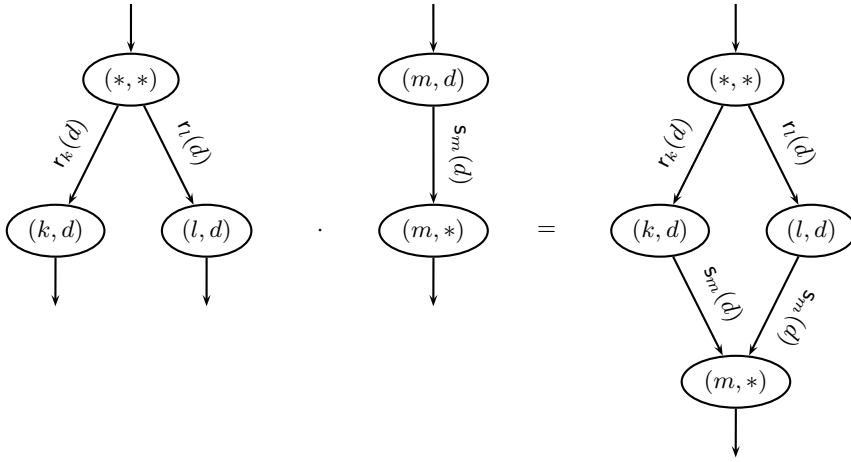


Fig. 3.2. Sequential composition of $R_k(d) + R_l(d)$ and $S_m(d)$

- for all $s_1, s_2 \in S$ and $a \in A_\tau$: if $s_1 \xrightarrow{a} s_2$, then $s_1 \xrightarrow{a'} s_2$;
- for all $s \in S$ and $a \in A_\tau$: if $s_0 \xrightarrow{a} s$, then $s'_0 \xrightarrow{a'} s$;
- for all $s_1 \in S$ and $a \in A_\tau$: if $s_0 \xrightarrow{a} s_1$, then $s \xrightarrow{a'} s_1$ for each $s \in \downarrow$;
- $\downarrow' = \{s'_0\} \cup \downarrow$;
- $s'_0 \notin S$.

Like in the case of sequential composition, successfully terminating states are relevant to the transitions of the resulting transition system. In the case of iteration, the transitions from the initial state are copied to transitions from the successfully terminating states and to transitions from the new initial state. In this way, the choice, that is there initially, will come back after successful termination of T . Here is an example to illustrate that it is quite natural to look at certain real-life processes as the iteration of other processes.

Example 3.2.5. We look once more at the railroad crossing controller from Examples 3.2.1 and 3.2.3. In this example, we take into account that, because of fault tolerance considerations, approach signals should always cause the gate to go down, and exit signals should be ignored while the gate is going down. Suppose that $S(\text{lower})$ is the transition system describing the behaviour of the subprocess dedicated to sending a lower signal. The behaviour of the subprocess dedicated to handling an approach signal that is received at the beginning of a cycle of the controller is described by $(R(\text{appr}) + R(\text{exit}))^* \cdot S(\text{lower})$. This is the transition system D referred to in Example 3.2.3.

Example 3.2.6. The iteration $((R_k(d) + R_l(d)) \cdot S_m(d))^*$ is the transition system $(S''', A''', \rightarrow''', \downarrow''', s_0''')$ where

$$\begin{aligned}
S''' &= \{*, (*, *), (k, *), (l, *), (k, d), (l, d), (m, d), (m, *)\}, \\
A''' &= \{r_k(d), r_l(d), s_m(d)\}, \\
\rightarrow''' &= \{* \xrightarrow{r_k(d)} (k, d), * \xrightarrow{r_l(d)} (l, d), \\
&\quad (*, *) \xrightarrow{r_k(d)} (k, d), (*, *) \xrightarrow{r_l(d)} (l, d), \\
&\quad (k, *) \xrightarrow{r_k(d)} (k, d), (l, *) \xrightarrow{r_l(d)} (l, d), \\
&\quad (m, d) \xrightarrow{s_m(d)} (m, *), (k, d) \xrightarrow{s_m(d)} (m, *), (l, d) \xrightarrow{s_m(d)} (m, *), \\
&\quad (m, *) \xrightarrow{r_k(d)} (k, d), (m, *) \xrightarrow{r_l(d)} (l, d)\}, \\
\downarrow''' &= \{*, (m, *)\}, \\
s_0''' &= *.
\end{aligned}$$

The transition system for $((R_k(d) + R_l(d)) \cdot S_m(d))^*$ is represented graphically in Figure 3.3.

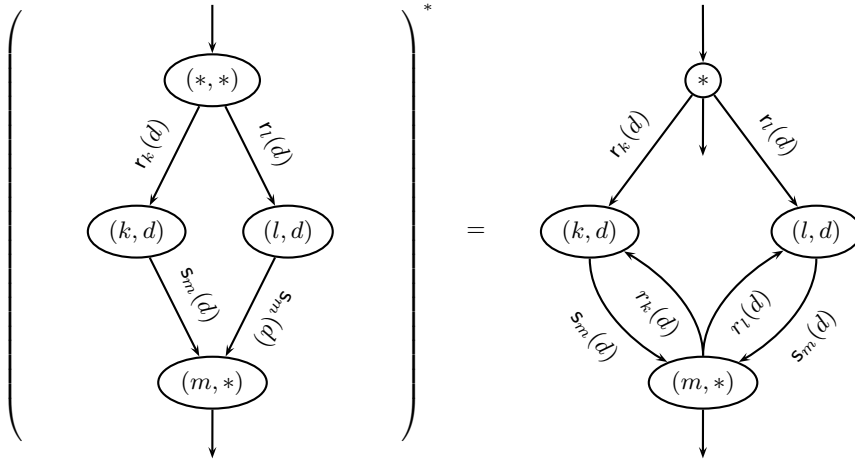


Fig. 3.3. Iteration of $(R_k(d) + R_l(d)) \cdot S_m(d)$

Definition 3.2.4 (No-exit iteration). Let T be a transition system. The *no-exit iteration* of T , written T^ω , is the transition system $T^* \cdot T'$ where T' is the transition system $(\{s_0\}, \emptyset, \emptyset, \emptyset, s_0)$.

Here is an example to illustrate that it is quite natural to look at certain real-life processes as the no-exit iteration of other processes.

Example 3.2.7. We look again at the railroad crossing controller from Examples 3.2.1, 3.2.3 and 3.2.5. The transition system $(R(\text{appr}) \cdot D) + (R(\text{exit}) \cdot U)$ from Example 3.2.3 describes the behaviour of one cycle of the controller. The behaviour of the controller is described by $((R(\text{appr}) \cdot D) + (R(\text{exit}) \cdot U))^\omega$.

Let us also give an example illustrating the details of no-exit iteration.

Example 3.2.8. Let $(R_k(d)$ and $R_l(d)) \cdot S_m(d)$ be as defined in Example 3.2.4. The no-exit iteration $((R_k(d) + R_l(d)) \cdot S_m(d))^\omega$ is the transition system $(S''''', A''''', \rightarrow''''', \downarrow''''', s_0''''')$ where

$$\begin{aligned}
 S'''' &= \{*, (*, *), (k, *), (l, *), (k, d), (l, d), (m, d), (m, *), (**)\}, \\
 A'''' &= \{r_k(d), r_l(d), s_m(d)\}, \\
 \rightarrow'''' &= \{* \xrightarrow{r_k(d)} (k, d), * \xrightarrow{r_l(d)} (l, d), \\
 &\quad (*, *) \xrightarrow{r_k(d)} (k, d), (*, *) \xrightarrow{r_l(d)} (l, d), \\
 &\quad (k, *) \xrightarrow{r_k(d)} (k, d), (l, *) \xrightarrow{r_l(d)} (l, d), \\
 &\quad (m, d) \xrightarrow{s_m(d)} (m, *), (k, d) \xrightarrow{s_m(d)} (m, *), (l, d) \xrightarrow{s_m(d)} (m, *), \\
 &\quad (m, *) \xrightarrow{r_k(d)} (k, d), (m, *) \xrightarrow{r_l(d)} (l, d)\}, \\
 \downarrow'''' &= \emptyset, \\
 s_0'''' &= *.
 \end{aligned}$$

This transition system is branching bisimulation equivalent (it is even bisimulation equivalent) to the second transition system given for a merge connection in Example 1.5.10 in the case where D is a singleton set. The no-exit iteration of $(R_k(d) + R_l(d)) \cdot S_m(d)$ is represented graphically in Figure 3.4.

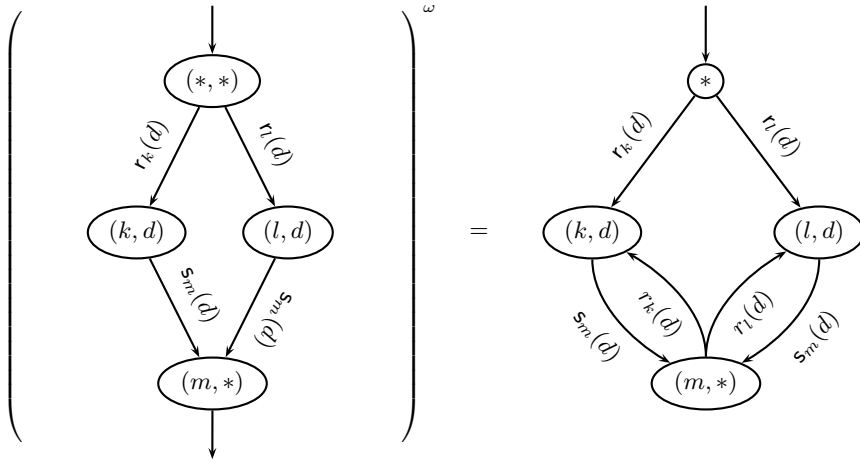


Fig. 3.4. No-exit iteration of $(R_k(d) + R_l(d)) \cdot S_m(d)$

Before we turn to more examples of the use of alternative composition, sequential composition and iteration, we will introduce atomic transition systems, i.e. transition systems that are capable of first performing a single action and then terminating successfully, and the inactive transition system, i.e. the transition system that is not capable of performing any action.

Definition 3.2.5. The *inactive* transition system is the transition system $(\{s_0\}, \emptyset, \emptyset, \emptyset, s_0)$ where s_0 is a fresh state. The inactive transition system is denoted by δ .

The *empty* transition system is the transition system $(\{s_0\}, \emptyset, \emptyset, \{s_0\}, s_0)$ where s_0 is a fresh state. The empty transition system is denoted by ϵ .

Let a be an action. The *atomic* transition system performing a is the transition system $(\{s_0, s_1\}, \{a\}, \{s_0 \xrightarrow{a} s_1\}, \{s_1\}, s_0)$ where s_0 and s_1 are fresh states. If no confusion can arise, the atomic transition system performing a is simply denoted by a .

The *atomic* transition system performing τ is the transition system $(\{s_0, s_1\}, \emptyset, \{s_0 \xrightarrow{\tau} s_1\}, \{s_1\}, s_0)$ where s_0 and s_1 are fresh states. If no confusion can arise, the atomic transition system performing τ is simply denoted by τ .

Bear in mind that it does not matter that arbitrary fresh states are chosen, as up to isomorphism the result is independent of the particular choice. Notice that the inactive transition system δ is used in the definition of no-exit iteration: $T^\omega = T^* \cdot \delta$.

Like for parallel composition, we use the convention of association to the left for alternative composition and sequential composition. The need to use parentheses is further reduced by ranking the precedence of the binary operations on transition systems. We adhere to the following precedence rules:

- the operation $+$ has lower precedence than all others;
- the operation \cdot has higher precedence than all others;
- all other operations have the same precedence.

For example, we write $x \cdot z + y \cdot z$ for $(x \cdot z) + (y \cdot z)$.

Here are a couple of examples of the composition of transition systems starting from atomic transition systems. These examples show a way to present transition systems that is quite different from the way that we used before. It looks to be a more convenient way.

Example 3.2.9. We consider again the bounded buffer from Example 1.1.3. We restrict ourselves to the case where its capacity is 1 and it can only keep bits, i.e. $D = \{0, 1\}$. Using alternative composition, sequential composition and no-exit iteration, its behaviour can be described as follows:

$$(\text{add}(0) \cdot \text{rem}(0) + \text{add}(1) \cdot \text{rem}(1))^\omega .$$

Example 3.2.10. We consider again the split connection from Example 1.5.9 and the merge connection from Example 1.5.10. We restrict ourselves once more to the case where only bits are involved, i.e. $D = \{0, 1\}$. Using alternative composition, sequential composition and no-exit iteration, the behaviour of the split connection and the merge connection can be described as follows:

$$(r_k(0) \cdot (s_l(0) + s_m(0)) + r_k(1) \cdot (s_l(1) + s_m(1)))^\omega$$

and

$$((r_k(0) + r_l(0)) \cdot s_m(0) + (r_k(1) + r_l(1)) \cdot s_m(1))^\omega .$$

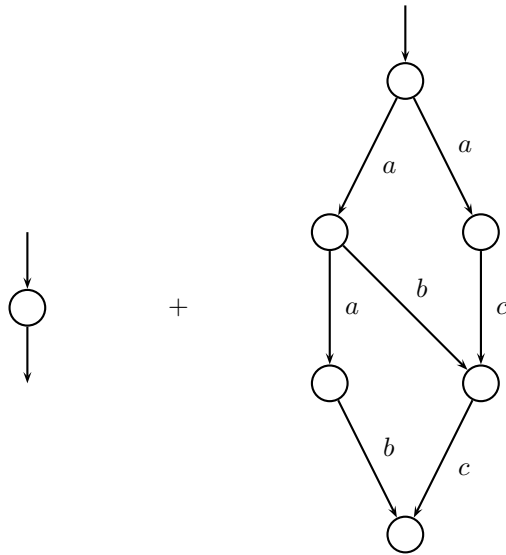
Here is another example, showing that the behaviour of simple PASCAL programs upon execution can also be described using alternative composition, sequential composition and iteration.

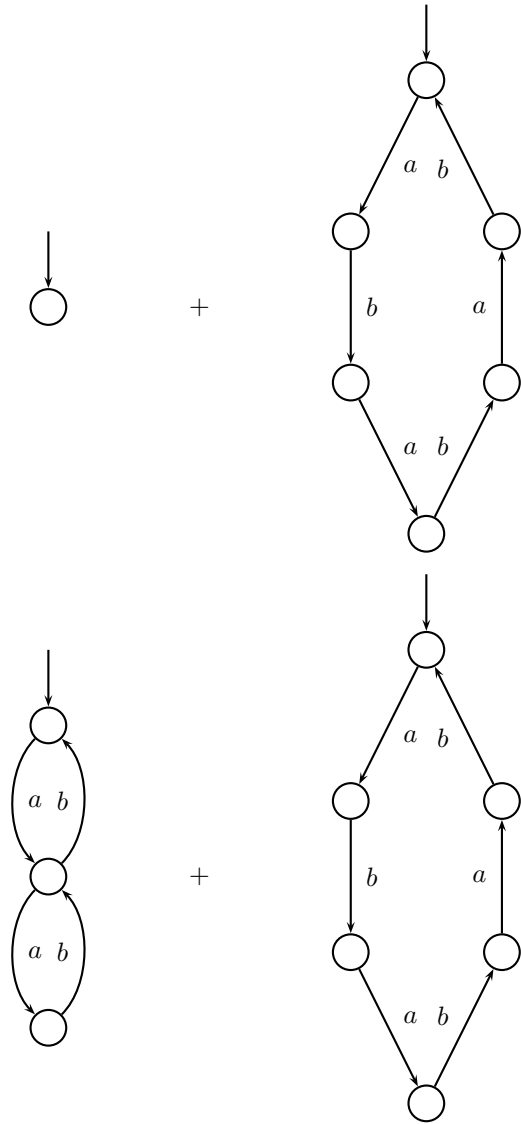
Example 3.2.11. We consider again the PASCAL program to calculate factorials from Example 1.3.1. Using alternative composition, sequential composition and iteration, the behaviour of this program upon abstract execution can be described as follows:

$$\begin{aligned} &(\text{read}(n)) \cdot (i := 0) \cdot (f := 1) \cdot \\ &(((i < n) \cdot (i := i + 1) \cdot (f := f * i))^* \cdot (\text{NOT } i < n)) \cdot (\text{write}(f)) . \end{aligned}$$

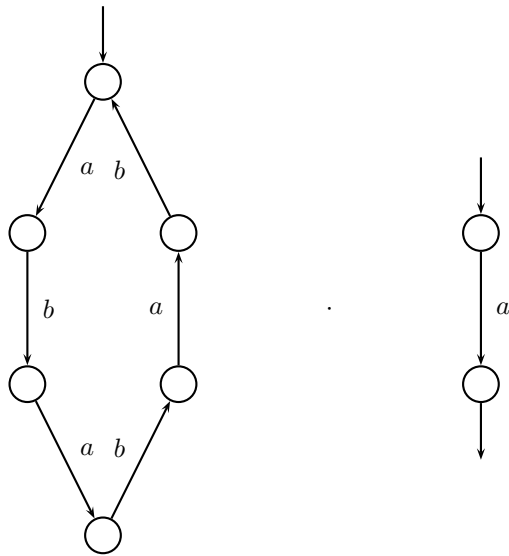
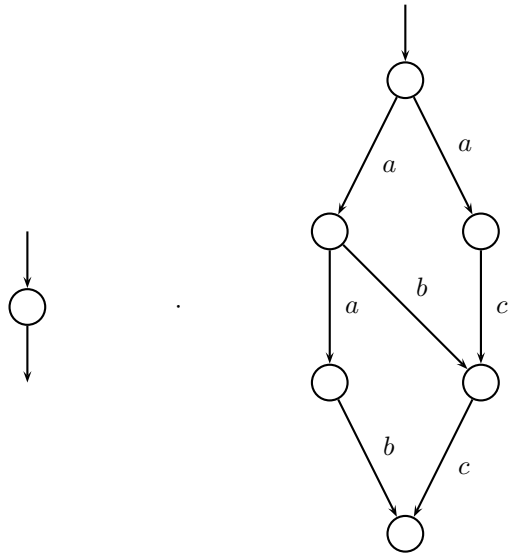
For reasons of readability, we have enclosed all atomic transition systems in parentheses. We cannot directly give a transition system describing the behaviour of a program upon execution on a machine by means of atomic transition systems, alternative composition, sequential composition and iteration. Nor we can give a transition system describing the behaviour of the machine on which the program is executed in this way. For the machine, as well as a category of simple programs, it is possible if we use in addition parallel composition, encapsulation and abstraction. However, it requires special tricks.

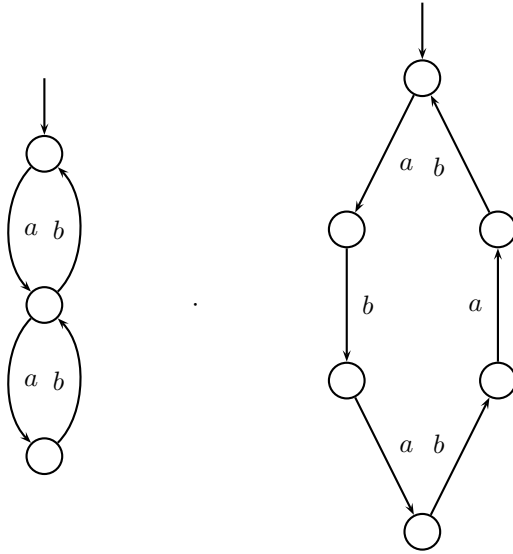
Exercise 3.2.1. Compute the following compositions using the formal definitions. Then give a graphical representation of the resulting transition systems.





Exercise 3.2.2. Compute the following compositions using the formal definitions. Then give a graphical representation of the resulting transition systems.



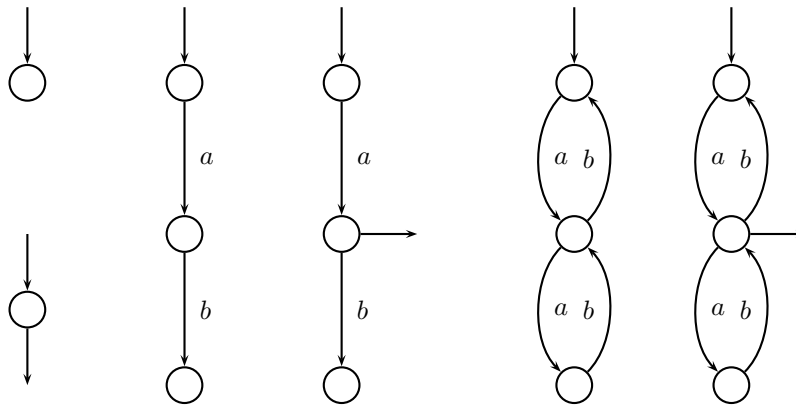


Exercise 3.2.3. Compute the following transition systems:

1. $a \cdot b + a \cdot \delta$
2. $a \cdot (b + \delta)$
3. $a \cdot b \cdot \epsilon \cdot d$
4. $a + b \cdot \epsilon + c \cdot d \cdot \delta + b$

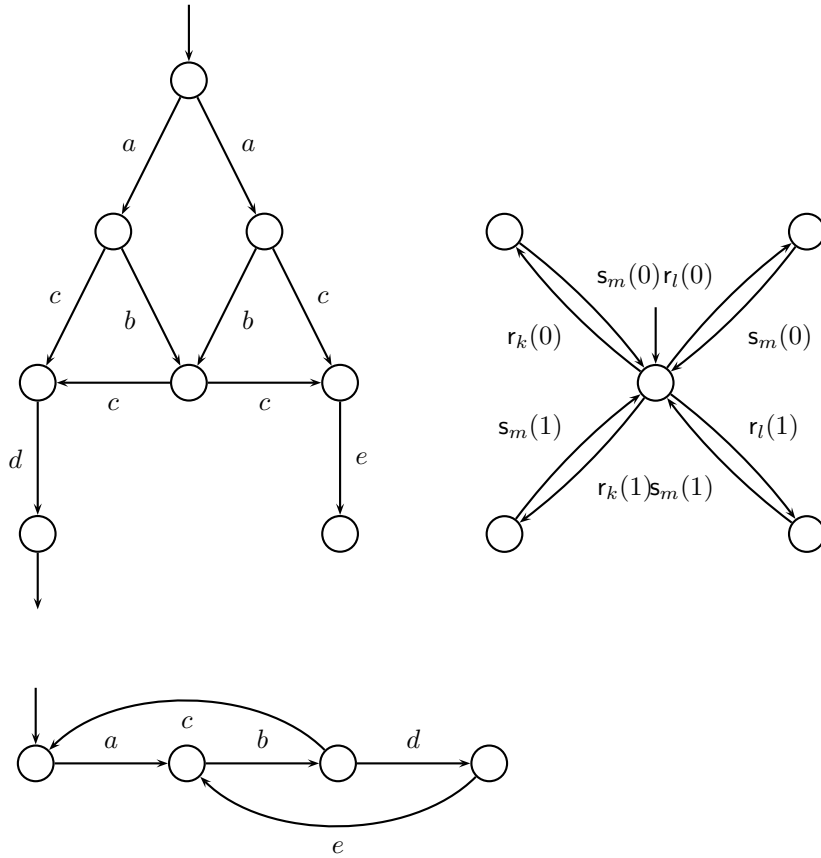
Exercise 3.2.4. Let $T = (S, A, \rightarrow, \downarrow, s_0)$ be an arbitrary transition system. Compute the transition system that results from the composition $T + \delta$. Under which of the equivalences defined so far are these transition systems equivalent?

Exercise 3.2.5. Compute for each of the following transition systems the result of applying iteration to it. Do the same with respect to no-exit iteration.



Exercise 3.2.6. Prove, using the formal definition of iteration, that for any transition system $T = (S, A, \rightarrow, \downarrow, s_0)$ it holds that $(T^*)^* \stackrel{\text{b}}{\cong} T^*$.

Exercise 3.2.7. Give, for the following transition systems, an expression that is bisimulation equivalent with it.



Exercise 3.2.8. Let T_1 , T_2 and T_3 be arbitrary transition systems. Consider the property $T_1 \cdot (T_2 + T_3) \cong T_1 \cdot T_2 + T_1 \cdot T_3$. Give a counterexample that proves that the property does not hold. What about the property $T_1 \cdot (T_2 + T_3) \equiv_{tr} T_1 \cdot T_2 + T_1 \cdot T_3$?

A. Set theoretical preliminaries

In this appendix, we give a brief summary of facts from set theory used in this book. This will at least serve to establish the terminology and notation concerning sets. First of all, we treat elementary sets (Appendix A.1). After that, we look at relations, functions (Appendix A.2) and sequences (Appendix A.3).

A.1 Sets

A *set* is a collection of things which are said to be the *members* of the set. A set is completely determined by its members. That is, if two sets A and A' have the same members, then $A = A'$. We write $a \in A$ to indicate that a is a member of the set A , and $a \notin A$ to indicate that a is not a member of the set A . A set A is a *subset* of a set A' , written $A \subseteq A'$ or $A' \supseteq A$, if for all x , $x \in A$ implies $x \in A'$.

If a set has a finite number of members a_1, \dots, a_n , then the set is written as follows:

$$\{a_1, \dots, a_n\}.$$

Let $P(x)$ be the statement that x has property P . Then the set whose members are exactly the things that have property P , if such a set exists, is written as follows:

$$\{x \mid P(x)\}.$$

If A is a set and $P(x)$ is the statement that x has property P , then there exists a subset of A of which the members are exactly the members of A that have property P . This set is denoted by $\{x \in A \mid P(x)\}$:

$$\{x \in A \mid P(x)\} = \{x \mid x \in A \text{ and } P(x)\}.$$

If A is a set, then there exists a set of which the members are exactly the subsets of A . This set is called the *powerset* of A and is denoted by $\mathcal{P}(A)$:

$$\mathcal{P}(A) = \{x \mid x \subseteq A\}.$$

If \mathcal{A} is a set of sets, then there exists a set of which the members are exactly the members of the subsets of \mathcal{A} . This set is called the *union* of \mathcal{A} and is denoted by $\bigcup \mathcal{A}$:

$$\bigcup \mathcal{A} = \{x \mid \text{for some } A \in \mathcal{A}: x \in A\}.$$

There exists a set with no members. This set is called the *empty set* and is denoted by \emptyset :

$$\emptyset = \{x \mid x \neq x\}.$$

Let A and A' be sets. Then the usual set operations *union* (\cup), *intersection* (\cap) and *difference* (\setminus) are defined as follows:

$$\begin{aligned} A \cup A' &= \{x \mid x \in A \text{ or } x \in A'\}, \\ A \cap A' &= \{x \mid x \in A \text{ and } x \in A'\}, \\ A \setminus A' &= \{x \mid x \in A \text{ and } x \notin A'\}. \end{aligned}$$

If A and A' are sets, then there exists a set of which the members are exactly A and A' . This set is called the *unordered pair* of A and A' and is denoted by $\{A, A'\}$. Let A be a set, $a \in A$ and $a' \in A$. Then the *ordered pair*, or shortly *pair*, with *first element* a and *second element* a' , written (a, a') , is the set defined as follows:

$$(a, a') = \{\{a\}, \{a, a'\}\}.$$

Let A and A' be sets. Then the set operation *cartesian product* (\times) is defined as follows:

$$A \times A' = \{(x, x') \mid x \in A \text{ and } x' \in A'\}.$$

This is extended in the obvious way to the cartesian product of more than two sets. An *ordered n -tuple* ($n > 2$), or shortly *n -tuple*, with *first element* a_1, \dots, n th element a_n , written (a_1, \dots, a_n) , is the set defined as follows:

$$(a_1, \dots, a_n) = ((a_1, \dots, a_{n-1}), a_n).$$

A pair is sometimes also called a 2-tuple. Let A_1, \dots, A_n be sets. Then the *cartesian product* of more than two sets is defined as follows:

$$A_1 \times \dots \times A_n = \{(x_1, \dots, x_n) \mid x_1 \in A_1, \dots, x_n \in A_n\}.$$

If a set has a finite number of members, the set is said to be *finite*. We use the following abbreviation. We write $\mathcal{P}_{\text{fin}}(A)$ for $\{x \in \mathcal{P}(A) \mid x \text{ is finite}\}$, the set of all finite subsets of A .

As usual, we write \mathbb{N} to denote the set of all natural numbers, and \mathbb{B} to denote the set $\{\mathbf{t}, \mathbf{f}\}$ of all boolean values.

A.2 Relations and functions

Let A_1, \dots, A_n be sets. An n -ary relation R between A_1, \dots, A_n is a subset of $A_1 \times \dots \times A_n$. If $A_1 = \dots = A_n$, R is called an n -ary relation on A_1 . We often write $R(a_1, \dots, a_n)$ for $(a_1, \dots, a_n) \in R$.

Let A be a set and R be a binary relation on A . Then we define the following:

- R is *reflexive* if $R(x, x)$ for all $x \in A$;
- R is *symmetric* if $R(x, y)$ implies $R(y, x)$ for all $x, y \in A$;
- R is *transitive* if $R(x, y)$ and $R(y, z)$ implies $R(x, z)$ for all $x, y, z \in A$;
- R is an *equivalence relation* on A if R is reflexive, symmetric and transitive.

Let A be a set and R be an equivalence relation on A . Then, for each $a \in A$, the set $\{x \in A \mid R(a, x)\}$ is called an *equivalence class* with respect to R . The members of an equivalence class are said to be *representatives* of the equivalence class.

Let A and A' be sets. Then a *function from A to A'* is a relation f between A and A' such that for all $x \in A$ there exists a unique $x' \in A'$ with $(x, x') \in f$. This x' is called the *value of f at x* . We write $f : A \rightarrow A'$ to indicate that f is a function from A to A' , and we write $f(x)$ for the value of f at x .

A function $f : A \rightarrow A'$ is called *bijective* if

- for all $x, y \in A$: $f(x) = f(y)$ implies $x = y$;
- for all $x' \in A'$: $x' = f(x)$ for some $x \in A$.

If A, A' and A'' are sets, $A \subseteq A'$ and $f : A' \rightarrow A''$, then there exists a set of which the members are exactly the values of f at the members of A . This set is denoted by $\{f(x) \mid x \in A\}$:

$$\{f(x) \mid x \in A\} = \{x' \mid \text{for some } x \in A: f(x) = x'\}.$$

Let \mathcal{I} be a set, \mathcal{A} be a set of sets. Then a *family* indexed by \mathcal{I} is a function $A : \mathcal{I} \rightarrow \mathcal{A}$. The set \mathcal{I} is called the *index* set of the family. We write A_i for $A(i)$. If A is a family indexed by \mathcal{I} , then we write $\bigcup_{i \in \mathcal{I}} A_i$ for $\bigcup\{A_i \mid i \in \mathcal{I}\}$.

We also use the following abbreviation. We write $\{f(x) \mid x \in A, P(x)\}$ for $\{f(x) \mid x \in \{x' \in A \mid P(x')\}\}$.

A.3 Sequences

Let A be a set and $n \in \mathbb{N}$. Then a (finite) *sequence* over A of *length* n , is a function $\sigma: \{i \in \mathbb{N} \mid 1 \leq i \leq n\} \rightarrow A$. If $n > 0$ and $\sigma(1) = a_1, \dots, \sigma(n) = a_n$, then the sequence is written as follows:

$$a_1 \dots a_n .$$

The sequence of length 0 is called the *empty* sequence and is denoted by ϵ .

Let A be a set. Then the set of all sequences over A is denoted by A^* , and the set of all nonempty sequences over A is denoted by A^+ . For each $\sigma \in A^*$, we write $|\sigma|$ for the length of σ .

Let A be a set, and $\sigma, \sigma' \in A^*$. Then the sequence operation *concatenation* (\frown) is defined as follows. $\sigma \frown \sigma'$ is the unique sequence $\sigma'' \in A^*$ with $|\sigma''| = |\sigma| + |\sigma'|$ such that:

$$\begin{aligned} \sigma''(i) &= \sigma(i) && \text{if } 1 \leq i \leq |\sigma|, \\ \sigma''(i) &= \sigma'(i - |\sigma|) && \text{if } |\sigma| + 1 \leq i \leq |\sigma| + |\sigma'|. \end{aligned}$$

We usually write $\sigma\sigma'$ for $\sigma \frown \sigma'$.

Let A be a set, and $\sigma, \sigma' \in A^*$. Then σ' is a *prefix* of σ , written $\sigma' \preceq \sigma$, if there exists a $\sigma'' \in A^*$ such that $\sigma'\sigma'' = \sigma$; and σ' is a *proper prefix* of σ , written $\sigma' \prec \sigma$, if $\sigma' \preceq \sigma$ and $\sigma' \neq \sigma$.