Introduction to spefication and verification
Lecture Notes, autumn 2011

Timo Karvi

UNIVERSITY OF HELSINKI
FINLAND

Contents

1 Introduction **1**

1.1 The starting point . . . 1

1.2 Global state graph . . . 2

1.3 Verification based on equivalences . . . 2

1.4 Model Checking . . . 4

1.5 Practical Experiences . . . 4

2 State Transition Systems **7**

2.1 Alternating Bit Protocol . . . 7

2.2 Client/Server-system . . . 10

2.3 The Definition of a Transition System . . . 11

3 Global State Graph **13**

3.1 Introduction . . . 13

3.2 The Global State Graph of the AB-protocol . . . 13

3.3 Parallel Operator . . . 15

3.4 Properties of the Parallel Operator . . . 17

3.4.1 Multisynchronization . . . 17

3.4.2 The Nature of the Synchronization . . . 17

3.4.3 Associativity of the Parallel Operator . . . 17

3.5 Implementing the Global State Graph . . . 19

3.5.1 Global State Graph as a Data Structure . . . 19

3.5.2 Bit Hashing and Alternatives . . . 20

4 Basic Principles in Modelling **21**

4.1 Non-determinism . . . 21

4.2 Channels and Environments . . . 22

5 Equivalences and Verifications **27**
5.1 Modelling Services 27
5.2 Relations 29
5.3 Trace Equivalence 32
5.4 Weak Bisimulation Equivalence 32
5.5 Examples 37
5.5.1 AB-protocol 37
5.5.2 FE-protocol 38
5.6 Conclusions and Problems 41

6 Basic Lotos **45**
6.1 Introduction 45
6.2 Ready-Made Processes 46
6.3 Action Prefix 46
6.4 Hiding 47
6.5 Choice 48
6.6 Parallel Operator 49
6.7 Sequental Composition 52
6.8 Disabling 52
6.9 The Precedence of the Operators 53
6.10 Process Instantiation 53
6.11 Exit and noexit 56
6.12 Examples 56
6.12.1 Specification of a Producer-Client System 56
6.12.2 Counter 58
6.12.3 A Client Server System 59
6.13 CADP 60
6.14 AB-protocol 60

7 Model Checking **65**
7.1 Introduction 65
7.2 Kripke Structures 65
7.3 Linear Time Logic LTL 67
7.4 Example 71

Chapter 1

Introduction

1.1 The starting point

When starting to verify distributed systems, the first thing to do is to describe the system. For this, there are two main possibilities. First, it is possible to use a high-level specification language. A programming language is too cumbersome because of its many details, if the aim is a mathematical verification. On the other hand, it is possible to use a programming language if special commments are added into the code. These comments are used by a verification software during its correctness analysis. This latter alternative has its supporters, but we concentrate in this course on the first alternative.

First we define *transition systems* with which we describe single processes. A transition system is a directed graph with labels added to its arcs. These labels are called *actions*. An action describes either a communication with another process or an internal computation. It is usual that in the distributed protocols it is not necessary to describe internal computations in a detailed way, but only very abstract descriptions are sufficient.

Transition systems must be represented in one way or another, if they are given as input to the verification software. There are several *specification languages* for this purpose. A simple specification language just enumerates the nodes and the labelled arcs between the nodes. But usually a language contains additional features, as for example a possibility to define internal computations or the descriptions of infinite systems. An example of an older language is *Estelle* which contains a lot of features from Pascal. More modern languages are based on *process algebras*. In these languages, transition systems are represented by an algebraic notation. This leads to compact representations and the syntax and especially semantics of the representations are formally defined. *Lotos* is an example of these languages and we will study it in this course. It is based on the process algebras CCS and CSP.

There are still other possibilities for specification languages. *Petri nets* resemble transition systems, but are richer in their structure. Languages based on *temporal*

logics are also popular in practice.

1.2 Global state graph

A single process in a distributed system is often quite simple. On the other hand, it is hard to analyse a system consisting of several, although simple processes. There are new kind of errors such as *deadlocks* and *livelocks*. Furthermore, often it is necessary to guarantee fairness in the sense that the processes of the system get recources equally. It is not easy to quarantee these properties just by traditional testing, because there are a huge amount of various execution sequences.

It is also possible to model the behavour of a large system using a single transition system. In this case, the state is a vector whose i'th element represents the state of the i'th process. When the process i changes its state, the vector changes at the same time: the i'th element changes. A single process may change its state when sending or receiving information, or when doing internal computations.

The first problem to be solved when modelling a system with a global state graph is how to treat true concurrency. If processes p and q execute their actions at the same time, the items in the global state vector corresponding to p and q should also change at the same step. But it is possible that first p changes and only then q, or vice versa. Should we take all the possibilities into account? If this is done mechanically, there would be a huge number of different execution sequences. That is why so called interleaving semantics is applied. In this approach it is thought that concurrent events happen always in certain order and truely concurrent events do not exist. This principle seems to work well in practice and most of the verification software tools are based on interleaving semantics.

In spite of the interleaving semantics there are still a lot of different execution sequences in the global state graph. It is called a state explosion. In the interleaving semantics, all the execution orders are still visible and most of the these orders lead to the same result. However, it is not easy to determine beforehand, which orders are essential and which not. This is the reason why it is not possible to generate the global state graph completely for the large systems. There are some suggestions how to restrict the size of the global state graph. One possibility is to generate states or rather paths in the global state graph randomly. It is also possible that the true concurrency semantics results, if cleverly applied, in a smaller graph than the interleaving semantics. However, we do not consider these alternatives during this course.

1.3 Verification based on equivalences

By using a global state graph, it is possible to detect many errors in the protocol. Deadlocks may be seen already when generating the global state graph. Livelocks correspond the components of the state graph such that it is no more possible to

go out of the component to the main cycle. These kind of mistakes are found, for example, with the help of depth-first search. There are other kind of mistakes which are difficult to find just by examining the state graph. For example, it may be erroneous to perform some actions in a wrong order. In order to detect mistakes in action orders it is necessary to compare execution sequences in the state graph to real execution sequences. These real sequences must thus be represented in some way or another.

First we introduce verification based on equivalences. In this case the starting point is as follows. The operation of a system can often be described from the viewpoint of an outsider. This outsider sees the actions transmitted between the participants, but it does not see, at least not exactly, internal operations performed by the processes. In other words, the observer sees what happens in the interfaces of the processes. The behaviour of the interfaces can be described with the help of transition systems. We call this description *a service*. It is also often the case that the service can be constructed from the definition of the protocol. Thus the service forms a reference with which the behaviour of the system or protocol can be compared. This comparison is done as follows.

First we specify the protocol using some specification language. This means in practice that every process in the protocol, such as sender, receiver, and timers, is written for example in Lotos. After this, the global state graph is generated. This state graph is itself a transition system and it is constructed automatically using software which understands the specification language in use. The arcs of the state graph show actions which are performed by the protocol.

The same is done for the service. Thus the service is described in Lotos and the global state graph is formed. Usually the state graph of the service is much smaller than the graph of the protocol. Next we hide all the actions in the state graph of the protocol which are not present in the service. This hiding means that we change the name of the action to a special name which represents an internal action.

Now we compare the graphs. There are various equivalences for this purpose. Often it does not matter which of the equivalences we use, but sometimes the choice of the equivalence is important, because some equivalences are coarser than the other. For example, some equivalences do not pay attention on cycles which consist of internal actions. In some situations it may be necessary to consider these kind of cycles, for example in real time applications.

One of the most popular equivalences is weak bisimilarity. Processes P and Q are weakly bisimilar, if the other process can simulate the visible actions of the other. In addition, the simulation order can be changed on the fly. This means that for example P simulates first Q, but in the middle of the simulation Q starts to simulate P from the point that was reached in the previous simulation.

In order to be efficient, the equivalence must be calculated quickly, in polynomial time. Bisimilarity is such an equivalence. There are other equivalences, for example trace-based equivalences, which demand an exponential time in the worst case, but in practice they perform quickly, in linear or polynomial time. Such equivalences

are useful, too.

If it possible to show that a protocol and its service are equivalent, the protocol is consired correct. In this approach the computer makes most of the work. The task of humans is to write the specifications and to choose a right equivalence. Extra work is sometimes needed, if the chosen equivalence does not detect all the possible mistakes. For example, fairness may be verified using other methods, for example temporal logic. Another reason why verification is sometimes difficult is the sizes of global state graphs. These are often too large in practical situations. There are various methods to try to tackle this problem: partitioning the state graph or just performing random walk in the graph.

1.4 Model Checking

Model checking is an alternative approach to the verification instead of equivalence-based approach. It means usually that temporal logic is used to check some properties of the system. Properties are expressed by logical formulas, state graph is formed, and then it is checked if the formulas are valid in the graph, i.e. in the model. In this way it is easy to express only certain properties that are to be verified. On the other hand, the total correctness is not so easily reached as in the equivalence and serviced based approach.

There are many kinds of temporal logics. Two of the most common are linear time logic (LTL) and branching time logic (CTL). In LTL, the validity of formulas depends on all the paths in the graph. In CTL, the validity may depend only on some paths. These logics are independent in the sense that both can express properties which cannot be expressed by the other. That is why stronger logics have been developed, for example CTL*. It contains both LTL and CTL.

Traditionally, the basic model to represent timed systems is a graph, where the arcs do not contain labels, i.e. actions. Instead, the states have properties or structures. If we use graphs generated from process algebraic formulas, the states have no structure, but the arcs contain labels. That is one reason why other kind of formalisms have been developed. There is ACTL which is developed from CTL. Others are Hennessy-Milner logic and μ-calculus. In this course, we concentrate only on LTL and CTL.

1.5 Practical Experiences

Formal methods have been applied for more than 20 years. Much of the applications have been done in academic research projects, but industry has been interested, too. The best results in industry seem to be in hardware design. For example, Intel has a verification team. The verification is done mainly with the help of model checking.

Also protocols have been verified succesfully. The best known examples are as

follows. IEEE Futurebus and cache coherence protocols have been found to contain errors.Verification methods found 122 mistakes in ISDN/SUP protocol. A big error was found in Active structural control system. It could have worsen the effects of vibrations.

The serious problem in practical verification is that the methods demand deep understanding. Companies are not willing to invest such sums for specialists, because it takes a long time to analyse systems and the results are not always complete.

Formal verification methods have been tried to apply in security protocols as well. Some mistakes have been found, but this application area is more difficult to model and analyse than communication protocols or hardware systems. The whole area is still in active reseach phase.

Chapter 2

State Transition Systems

2.1 Alternating Bit Protocol

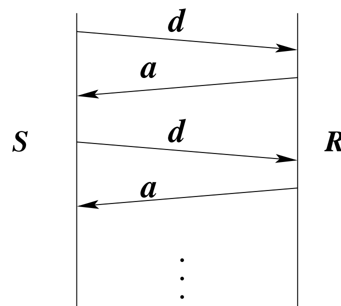
Before defining formally state transition systems, we show an example how a practical data link protocol can be specified with the help of transition systems.

Alternating bit protocol (AB protocol) is one of those simple protocols which have been analysed a lot in scientific literature of this area. It is elegant mathematically and non-trivial logically.

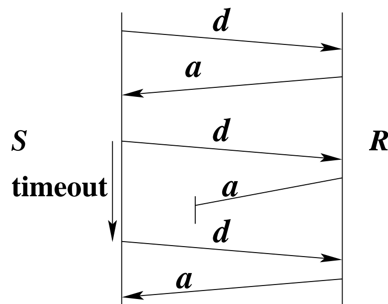
The protocol consists of a sender process S and receiver process R. These processes change messages through a half duplex channel. The channel may drop or distort messages. It is half duplex, which means that the messages can pass from S to R and from R to S, but there can not be messages in the channel which travel in opposite directions at the same time. The order of the messages in the channel cannot be changed. Also, a message cannot duplicate in the channel.

A receiver of a message is able to detect, if the message is distorted. This is a practical assumption which can be implemented with the help of modern codes, for example using the CRC field.

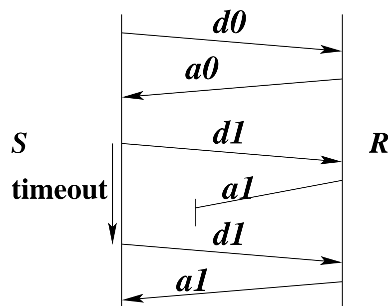
The goal of the protocol is to make sure that the receiver gets the messages sent by S correctly. Clearly S must get some information about the data messages d from R: has R received d correctly or not? This information comes as an acknowledgement a which is sent by R. Only positive acknowledgements are used. If a message is distorted, R does not sent an acknowledgement. The following diagram shows the behaviour of the protocol.



Thus S must wait for an acknowledgement before it can send next message. If the acknowledgement disappears, the sender sends no more, even if there are messages waiting for sending. That is why the protocol has a timer. The timer times out, if an acknowledgement does not come after a certain time. We get the following scenario:



There is now a possibility that the same message is sent two times and the receiver does not recognise this, but considers the messages different. This duplication is the result of acknowledgments. We must add some information to messages and acknowledgements so that different successive messages can be separated. It is enough to add only 0 or 1 to messages. We can now write the previous scenario as follows:



Let us model the sender and receiver as transition systems. We must first solve some details:

- Do we mark, if an action is a send or receive action? For example, if a sender sends a data message d, we have a transition $S_i \xrightarrow{d} S_j$. Should we use only d on the arc or some other notation, like $-d$ to point out that d is sent, not received?
- How is the timer modeled?
- How do we model the distortion or disappearing of packets?

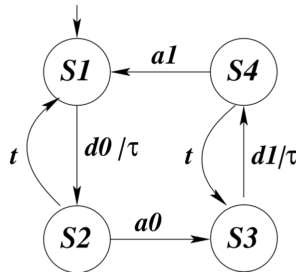
We solve the questions as follows. We do not mark, if an action is a send or receive action. This must follow from the context. Send and receive actions are separated explicitly only, when we start to use a specification language, for example Lotos, instead of transition systems.

In this first example we model the timer as a part of the sender process. Later we consider cases, where timers are modelled separately.

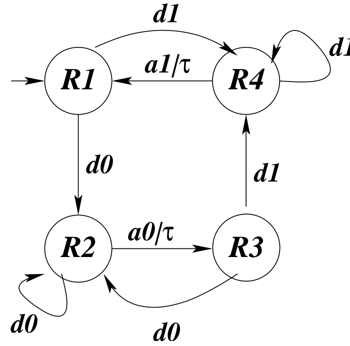
The disappearence of a message is modelled with the help of so called internal or silent action τ. It makes possible for a process to move from one state to another so that other processes do not notice it. The internal action has an important role in other contexts, too.

In addition, we use non-determinism. Non-determinism models situations, where more than one kind of an event may happen and a process cannot alone decide the event that really happens.

We get the following sender system S:



Receiver R:

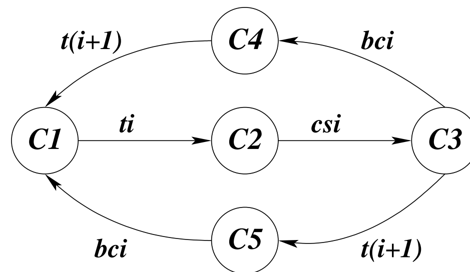


We have now written single processes participating in the system. Later we will see how to analyse the whole system as a single transition system.

2.2 Client/Server-system

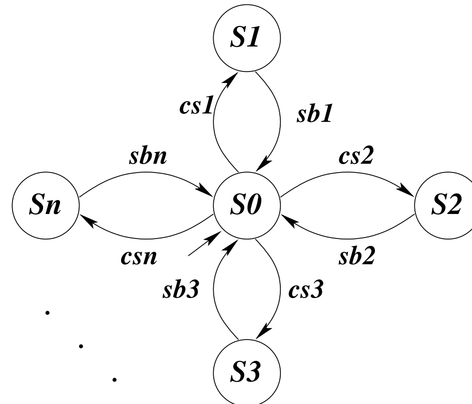
Let us consider one more example which consists of unlimited number of processes. There are a server process S and n client processes C_i, $i = 1, \cdots, n$. A client C_i asks a service from S with a message csi. S sends the answer sbi into the buffer B_i, and the client can take it from there with the help of a message bci. The protocols follows the round robin principle. This means that a token moves from one client to another. Every time a client gets the token, it asks for the service. After this the client gives the token to the other client. The token is modelled using messages. When a client C_i receives the message ti, it can ask for a service. After this it sends the message $t(i+1)$ to C_{i+1}. The addition is interpreted so that $n+1=1$.

Client C_i as a transition system:

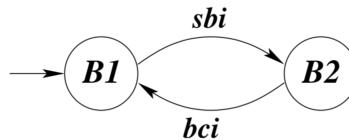


The starting state is C1 except in C_1, where it is C2. Thus round robin starts in C_1.

Server S:



Buffer B_i:



2.3 The Definition of a Transition System

We consider the situation where a system consists of two or more processes. Process means the same as in the operating systems, i.e.

- Process is a program, whose execution has started and not yet finished.
- The same code may generate several processes.
- A process proceeds with discrete steps from one state to another. State is defined using the values of variables and the next instruction or instructions – we allow non-determinism.

A process proceeds by performing actions which can be for example

- internal computation (updating variables etc),
- sending or receiving of a message,
- some other action involving other processes.

One example of the last case is synchronization. Of course, synchronization is possible also in sending or receiving.

Usually we use a high level of abstraction when describing systems with transition systems. It means in practice that the main point is the communication of the processes. Internal events are abstracted as much as possible. On the other hand, some specification languages offer many possibilities to represent internal events, while in transition systems the internal computations must be reduced to the minimum.

Definition 1 A labelled transition system *is a structure* $(S, A, \longrightarrow, s_0)$, *where*

- S *is the set of states;*
- A *is the set of actions which contains the internal or silent actions* τ.
- $\longrightarrow \subset S \times A \times S$ *is the transition relation;*
- s_0 *is the initial state.*

Often S and A are finite, but they could be enumerably infinite in principle. The elements in A represent actions which are involved in the communication of processes. We do not mark in any way, when a message is sent or received in the processes. We could define the sending and receiving separately, but we focus on a formalism which is compatible with the specification language Lotos. (In full Lotos, the actions are ports or gates through which the processes communicate by sending data. In our first formalism we do not distinguish between ports and data messages.)

A transition system shows how the process proceeds. The execution starts from the initial state. The transition relation dictates the alternatives the process can next perform. At the end of this chapter we define some formal concepts needed later.

1. If in a transition system $(s_1, a, s_2) \in \longrightarrow$, then we usually write $s_1 \stackrel{a}{\longrightarrow} s_2$.
2. If a τ-path
$$s_1 \stackrel{\tau}{\longrightarrow} s_2 \stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow} s_n,$$
starts from a state S_1, then we write $s_1 \stackrel{\varepsilon}{\Longrightarrow} s_n$. The notation $\stackrel{\varepsilon}{\Longrightarrow}$ covers also the case when there are no movement from the state. Thus always $s_1 \stackrel{\varepsilon}{\Longrightarrow} s_1$.
3. The notation $s_1 \stackrel{\tau}{\Longrightarrow} s_n$ means that $s_1 \stackrel{\varepsilon}{\Longrightarrow} s_k \stackrel{\tau}{\longrightarrow} s_r \stackrel{\varepsilon}{\Longrightarrow} s_n$.
4. The action sequence $u = a_1 a_2 \cdots a_n$ leads from state r to state s, if there is a path
$$r \stackrel{\varepsilon}{\Longrightarrow} r_1 \stackrel{a_1}{\longrightarrow} s_1 \stackrel{\varepsilon}{\Longrightarrow} r_2 \stackrel{a_2}{\longrightarrow} s_2 \stackrel{\varepsilon}{\Longrightarrow} \cdots \stackrel{\varepsilon}{\Longrightarrow} r_n \stackrel{a_n}{\longrightarrow} s_n \stackrel{\varepsilon}{\Longrightarrow} s.$$
The we use the notation $r \stackrel{u}{\Longrightarrow} s$.

Chapter 3

Global State Graph

3.1 Introduction

In the previous chapter we modelled single processes. We see the messages sent and reveived by every process, but it is impossible to figure out if the whole system behaves correctly. However, it turns out that the behaviour of the whole system can also be modelled as a transition system which is constructed from the graphs of the single processes. This transition system is called *global state graph*.

A state in a global state graph is vector, whose components are states of the single processes. Thus a state in a global state graph describes the states of every process in the system at that particular moment. Before we are able to give an exact definition, we must decide how we will model concurrency. In principle, we have two alternatives.

If the modelling is based on *true concurrency*, the model shows what events happen concurrently and waht sequentially. There are formalisms based on true concurrency, but they are complicated and they demand time-consuming algorithms. True concurrency is, however, an important concept.

The other method is based on *interleaving*. In this approach we assume that two events can always be ordered temporally, i.e. two events can never happen simultaneously. This principle seems at first incredible, but it works well in practice. Nearly all the verification programs are based on this assumption, because it simplifies definitions and algorithms. Our global state graph is based on interleaving.

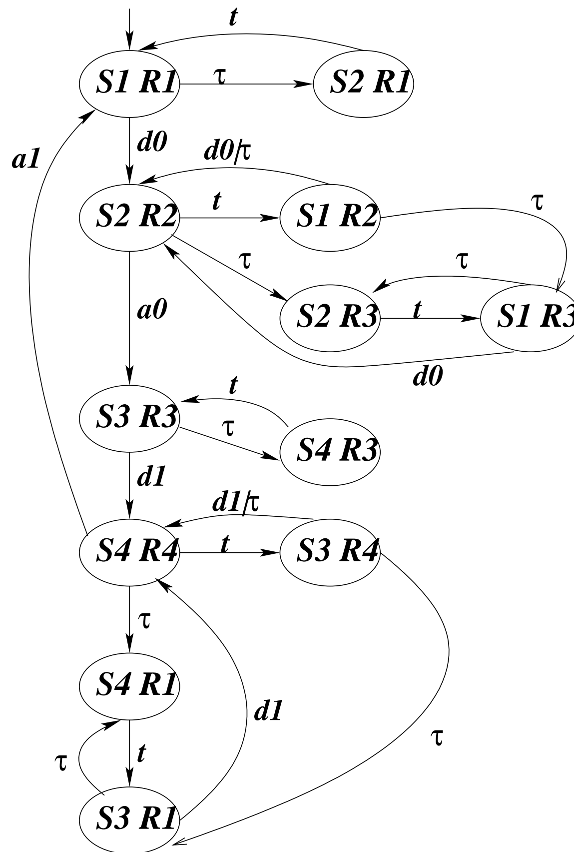
3.2 The Global State Graph of the AB-protocol

Before defining global state graphs formally we show a concrete example based on the AB-protocol. In this protocol, sender S sends messages $d0$ and $d1$ to receiver R. Receiver R sends acknowledgements $a0$ and $a1$ to S. Suppose that the communication between S and R is rendezvous-type. Thus before a sender can send, a

reveiver must be ready to receive. If a receiver is not ready, a sender cannot send. For example, if S sends $d0$, S moves to a next state with the action $d0$ and at the same time R moves to the next state with $d0$.

The internal transition τ does not cause other processes to change their states. Similarly, the timer action t changes only the state of the sender.

A state in the global state graph is a pair (Si, Rj), where Si is a state of the sender and Rj is a state of the receiver. Let us draw now the global state graph. We make no assumptions about the timer. It may timeout too early, but this should cause no problems.



The global state graph shows that there are no deadlocks. Also the basic cycle of the protocol is visible. Communication errors and timer actions lead away from the basic cycle, but in every case it is possible to return to the basic cycle. Thus the global state graph seems to show that the protocol works properly.

3.3 Parallel Operator

Next we define the global state graph formally with the help of the parallel operator. Let P ja Q be transition systems. Suppose that the states in P are $P_1, P_2, \cdots, P_m$ and the states in Q are $Q_1, Q_2, \cdots, Q_n$. The initial state of the whole system is (P_1, Q_1), where P_1 is the initial state in P and Q_1 is the initial state in Q. The states in the global state graph are of the form (P_i, Q_j), $i = 1, \cdots, m$, $j = 1, \cdots, n$. We write $P_i \xrightarrow{a} P_{i'}$, if there is a transition from P_i to $P_{i'}$ with an action a. Similarly in the case of Q.

The global state graph $P|[a_1, \cdots, a_k]|Q$ of the processes P and Q is defined now formally by giving the rules which show how to move from one state to another. The actions $a_1, \cdots, a_k$ are *synchronizing action*. If one process is to perform a_i, then the other synchronizes and performs it, too. If the other cannot perform a_i, then neither the first nor the second can perform it. Other actions can and must be performed without synchronization.

We define now the transitions between the states. This is done with the help of a so called parallel operator $|[a_1, \cdots, a_k]|$. It is demanded that $\tau \neq a_i$ for all $i = 1, \cdots, k$. If we apply the parallel operator to the state pair (P_i, Q_j), the synchronizing actions are marked by writing $P_i|[a_1, a_2, \cdots, a_k]|Q_j$. The following rules define what transitions are possible from a given global state, i.e. from a state pair (P_i, Q_j).

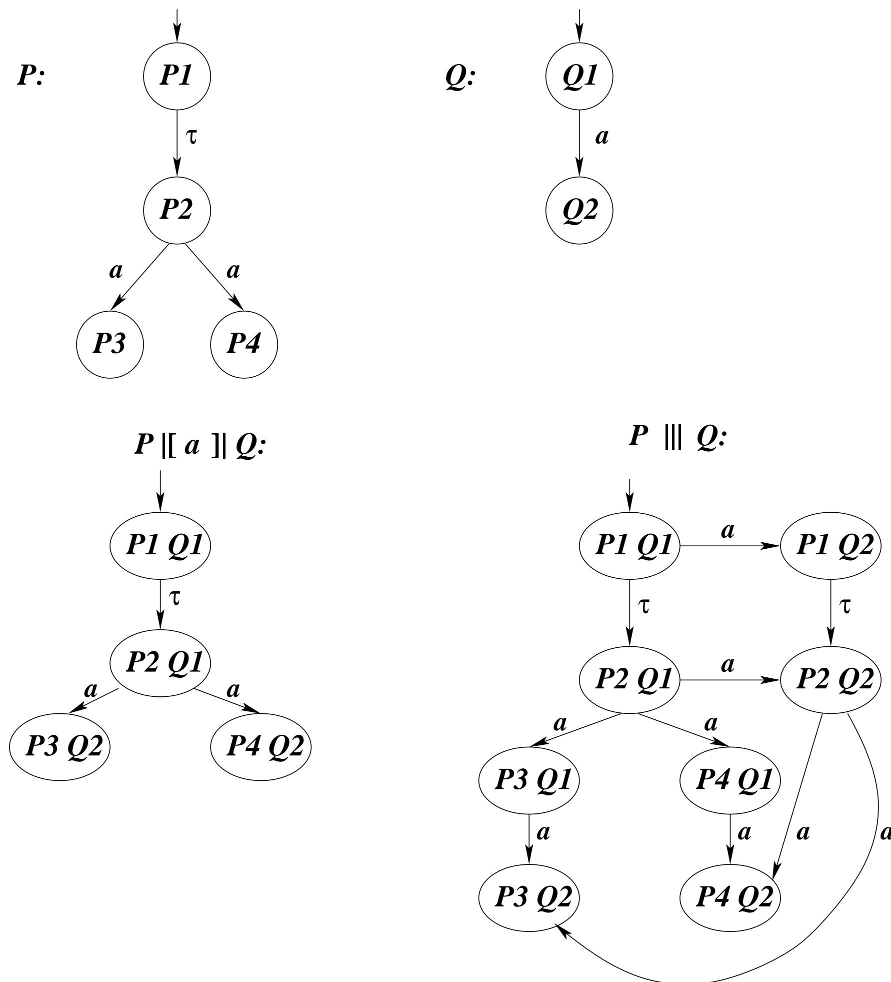
1. If $a \in \{a_1, \cdots, a_k\}$, $P_i \xrightarrow{a} P_{i'}$ and $Q_j \xrightarrow{a} Q_{j'}$, then
$$P_i|[a_1, a_2, \cdots, a_k]|Q_j \xrightarrow{a} P_{i'}|[a_1, a_2, \cdots, a_k]|Q_{j'}.$$
2. If $a \notin \{a_1, \cdots, a_k\}$ and $P_i \xrightarrow{a} P_{i'}$, then
$$P_i|[a_1, a_2, \cdots, a_k]|Q_j \xrightarrow{a} P_{i'}|[a_1, a_2, \cdots, a_k]|Q_j.$$
3. If $a \notin \{a_1, \cdots, a_k\}$ and $Q_j \xrightarrow{a} Q_{j'}$, then
$$P_i|[a_1, a_2, \cdots, a_k]|Q_j \xrightarrow{a} P_i|[a_1, a_2, \cdots, a_k]|Q_{j'}.$$

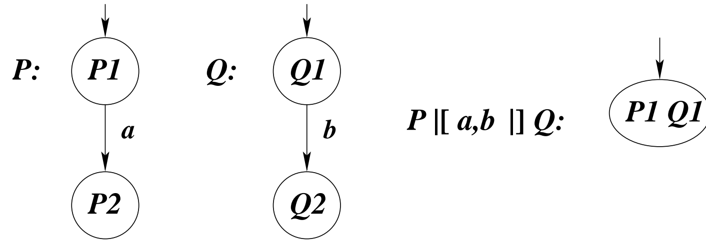
The application determines what is a suitable synchronizing set. If $k = 0$, i.e. no action synchronizes, we speak about complete interleaving and use the notation $|||$. If the synchronizing set consists of all the visible actions, we use the notation $||$.

The final global state graph consists only of those states that can be reached from the initial state. If we use the parallel operator to construct the global state graph of the AB-protocol, we start from the formula

$$S|[d0, d1, a0, a1]|R$$

and deduce all the other states and transitions from this formula. The result as a graph is the same as before. Some more examples:





3.4 Properties of the Parallel Operator

3.4.1 Multisynchronization

It is possible to synchronize several processes at the same time. For example, in the formula

```
P |[a]| (Q |[a]| R)
```

the action a can happen in Q and R only if both processes perform that action at the same time. On the other hand,

```
Q |[a]| R
```

is a process, too, and thus a can happen in it and in P only if all the three processes perform it at the same time.

3.4.2 The Nature of the Synchronization

Synchronization is *symmetric*. We do not distinguished which process starts it. Thus the sender and receiver are equal when a message is sent: both must be ready and perform an action.

Synchronization is *nameless*. If a process offers synchronization, any process with an suitable action may take part in the synchronization. It is not possible for a process to demand that a synchronization is directed to a particular process. This property has advantages and drawbacks in applications.

3.4.3 Associativity of the Parallel Operator

In general, the parallel operator is not associative. Thus it is not that

$$P \,|A_1|\, (Q \,|A_2|\, R) = (P \,|A_1|\, Q) \,|A_2|\, R.$$

In the following cases the parallel operator is, however, associative:

1. *CSP's case.* Let P, Q and R be processes and A_P, A_Q, A_R the action sets of the processes. Then

   $$P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R) \equiv (P\,|A_P \cap A_Q|\,Q)\,|(A_P \cup A_Q) \cap A_R|\,R,$$

   where '$\equiv$ means that the corresponding transition systems are the same, only the names of the states are different (notice that when using the parallel operator state in a global state graph contains that operator).

2. If B is an arbitrary action set, then

   $$P\,|B|\,(Q\,|B|\,R) \equiv (P\,|B|\,Q)\,|B|\,R.$$

3. With synchronizing sets B_1 and B_2 we have

   $$P\,|B_1|\,(Q\,|B_2|\,R) \equiv (P\,|B_1|\,Q)\,|B_2|\,R,$$

   if $A_P \cap B_2 = \emptyset$ and $A_R \cap B_1 = \emptyset$.

Proof of the item 1. It is enough to prove that every transition in

$$P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R)$$

corresponds exactly the same transition in

$$(P\,|A_P \cap A_Q|\,Q)\,|(A_P \cup A_Q) \cap A_R|\,R$$

and vice versa. It is possible to follow two strategies in the proof. In the first strategy we examine a single transition with a and deduce what action sets a belongs to. In the other strategy we we examine what components in the system change their states and deduce what kind of transition takes place in the other system. Let us follow the second strategy.

In what follows we denote by a apostrophe the process that changes in the transition. We have to check several alternatives.

a) $P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R) \xrightarrow{a} P'\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R)$.

b) $P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R) \xrightarrow{a} P\,|A_P \cap (A_Q \cup A_R)|\,(Q'\,|A_Q \cap A_R|\,R)$.

c) $P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R) \xrightarrow{a} P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R')$.

d) $P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R) \xrightarrow{a} P'\,|A_P \cap (A_Q \cup A_R)|\,(Q'\,|A_Q \cap A_R|\,R)$.

e) $P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R) \xrightarrow{a} P'\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R')$.

f) $P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R) \xrightarrow{a} P\,|A_P \cap (A_Q \cup A_R)|\,(Q'\,|A_Q \cap A_R|\,R')$.

g) $P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R) \xrightarrow{a} P'\,|A_P \cap (A_Q \cup A_R)|\,(Q'\,|A_Q \cap A_R|\,R')$.

Case a). In this case the transition takes place only inside P. Thus $a \in A_P$ ja $a \notin A_Q \cup A_R$. Hence $a \notin A_P \cap A_Q$ and $a \notin (A_P \cup A_Q) \cap A_R$, and we can deduce that the transition happens also in $(P\,|A_P \cap A_Q|\,Q)\,|(A_P \cup A_Q) \cap A_R|\,R$ only in P. And the transition is the same.

Cases b) and c) are proved in the same way as a).

Case d) Now the transition happens both in P and in Q. Thus $a \in A_P$ and $a \in A_Q \cup A_R$, but $a \notin A_Q \cap A_R$, hence $a \in A_Q$ and $a \notin A_R$. We can deduce that $a \in A_P \cap A_Q$, but $a \notin (A_{\cup} A_Q) \cap A_R$. Because of this, the transition with a in the global state graph $(P|A_P \cap A_Q|Q)\,|(A_P \cup A_Q) \cap A_R|\,R$ happens only in P and Q, and they change in the same way as in the graph $P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R)$.

Cases e) ja f) are proved in the same way as d).

Case g). Now all the processes change their states and thus $a \in A_P \cap (A_Q \cup A_R)$ ja $a \in A_Q \cap A_R$. Hence $a \in A_P$, $a \in A_Q$ and $a \in A_R$. Furthermore, $a \in A_P \cap A_Q$ and $a \in (A_P \cup A_Q) \cap A_R$. Thus the transition takes place in the graph $(P\,|A_P \cap A_Q|\,Q)\,|(A_P \cup A_Q) \cap A_R|\,R$ in all the processes and exactly in the same way as in $P\,|A_P \cap (A_Q \cup A_R)|\,(Q\,|A_Q \cap A_R|\,R)$. $\square$

Item 2) is proved in the same way as 1). Item 3) is a modification of item 1).

3.5 Implementing the Global State Graph

3.5.1 Global State Graph as a Data Structure

The global state graph is used in two ways. In some applications, it is enough to traverse the graph without constructing it completely. In some other applications, it is necessary to construct the whole graph explicitly. In this latter alternative there is a problem that the graph may be too large. As matter of fact, many practical protocols and systems lead to such a large graph that it is not possible to generate it completely. For example, it has not been easy to analyse formally the collaboration of several layers in network environments.

The global state graph is usually sparse. Thus there are only few out-going arcs from the states. Hence matrix representations are not promising, but adjajency list representations perform better. Usually the generation and analysis is done depth-first. Thus we start from the initial global state and generate all the states one step away from the initial state. These new states are pushed into a stack. Then continue as follows as long as the stack is non-empty: Take a state from the stack, generate all the neighbours of the chosen state, push the neighbours into the stack and draw the arcs from the chosen state to the neighbours. It is necessary to check always during the generation of new states, if the new states are really new or already generated. Usually this check is done with the help of hashing.

3.5.2 Bit Hashing and Alternatives

Because a global state graph is often large, we need a large hash table, too. It would be temptating to use the virtual memory, but this brings problems. The reason is that during the generation of states new and old states come in unpredictable order. This leads to the fact that it is necessary to fetch pages continually from the disk. This makes the processing too slow. That is why one tries to use only the central memory.

The solution of G. Holzmann was to use *bit hashing*. In this method, a global state is interpreted as a bit string and furthermore as an integer. Reserve now a boolean array, whose size is such that the index space includes the largest integer corresponding a global state. This kind of an array can be used efficently for hashing and one state demands only one bit.

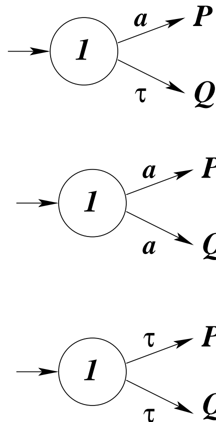
The whole transition system can be represented in an alternative way. Boolean functions can be represented in a compact way and transition system can be represented with the help of boolean functions. This representation is called BDD or binary decision diagram. Neither is this solution universal: in some situations BDD's help to compress the global state graph, but not always. Moreover, many algorithms work easier with adjajency lists than with BDD's.

Chapter 4

Basic Principles in Modelling

4.1 Non-determinism

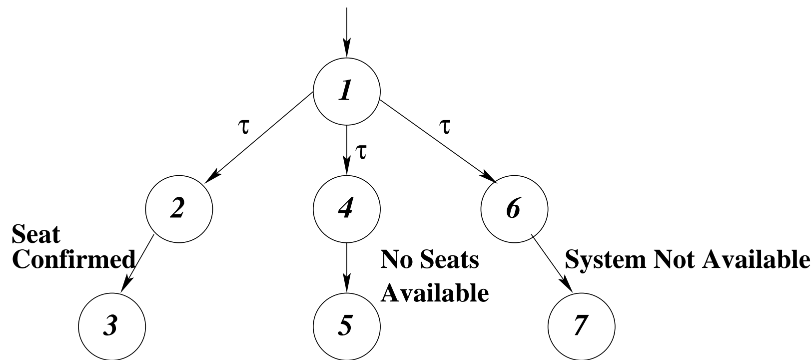
Let P and Q be transition systems and a an action. There are three ways to model non-determinism in transition systems:



In the case 1 the environment has a chance to interact with the system, if a happens before the internal action τ. In the case 2, the system itself decides, at the same time when the environment reacts, which one of a's it chooses. The case 3 says that the system decides internally, if it behaves according to P or Q.

For example when modelling a communication channel it is best to choose the alternative 3, if P represents the delivery of a message and Q the loss of a message. The environment, in this case the sender and receiver, cannot affect the behaviour of the channel. The delivery or loss of a packet is only dependent on the channel. If we used the alternative 1, the environment could in some cases affect the behaviour of the channel, what could not lead to realistic modelling.

The second example is the seat reservation system of airlines, whose one feature is described by the following transition system:



Thus the result of a seat reservation is totally unpredictable from the point of view of a customer, because a normal customer does not know, how the system is built or are there free seats available. The incapability of a customer to affect the system is modelled using internal transitions.

4.2 Channels and Environments

Communication in the computer networks is not ususally synchronized. The parallel operator, however, demands synchronization. It is possible to produce asynchronous communication by specifying a channel as a transition system. A process sends a message into the channel synchronously and continues then its computation. When it is ready, another process takes the message from the channel synchronously. In this way there is an asynchronous communication between the two processes. For this asynchronous communication one has to pay a price: the size of the global state increases, sometimes considerably. This increase depends on the amount of packets which can travel simultaneously in the channel.

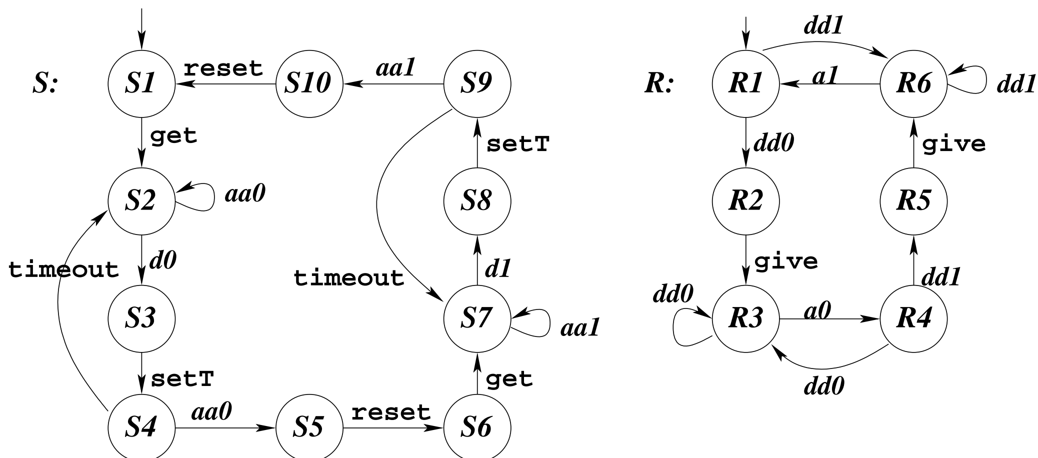
Sometimes it is also necessary to model the environment of a protocol. For example, it is possible to assume in the AB-protocol that the sender gets the data in `get` message from the environment, i.e. from the upper layer or other process, and that the receiver gives the data in `give` message to its own environment.

We model now the AB-protocol in a more orthodox way with the help of channels and environments. In addition, the timer is now a separate process. We make the following agreements:

- The sender sends the messages $d0$ and $d1$ into the channel.
- The receiver takes the messages $dd0$ and $dd1$ from the channel.
- The receiver sends the acknowledgements $a0$ and $a1$ into the channel.
- The sender takes the acknowledgements $aa0$ and $aa1$ from the channel.

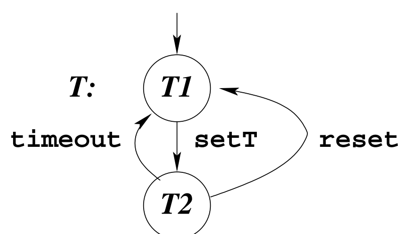
- The sender and environment communicate synchonously with the message `get`.
- The receiver and environment communicate synchronously with a message `give`.
- The sender sets the timer synchronouosly with the help of a message `setT`.
- The timer informs the sender about the timeout synchronously with the help of a message `timeout`.
- The sender informs the timer synchronously with a message `reset` that the timer can return to its initial state.

Below the processes are represented as transition systems.It is assumed of the channel that it can contain only one message at a time. Furthermore, it is the task of the channel to lose or distort messages.

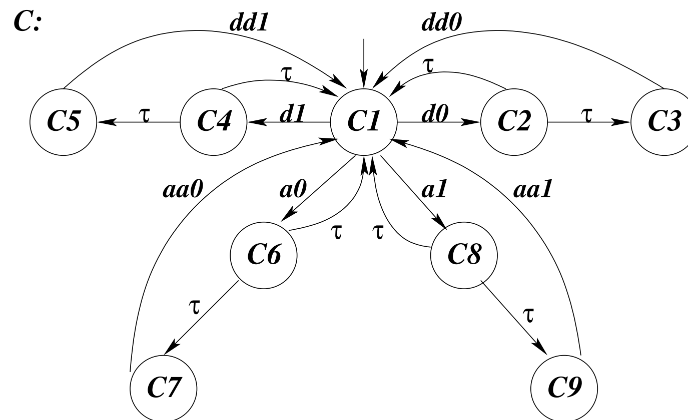


There are some new transitions in the sender process (states S2 and S7), where acknowledgements are received. This is because of the channel and the timer which may send timeout even if an acknowledgement is in the channel. It is necessary to take all the messages from the channel so that the channel can send new messages. If some message stayed in the channel, it would block all the traffic and the result would be a deadlock.

The timer can be represented with the help of two states.



The channel has been modelled in such a way that the environment cannot affect if messages disappear or not. It is thought that the channel takes the data messages $d0$ and $d1$ from the sender and delivers them as messages $dd0$ and $dd1$. Notice that it is not possible to take and deliver exactly the same message forms, because this would disturb the asynchronous communication. The same is true with the acknowledgements: $a0$ and $a1$ are taken by the channel and $aa0$ and $aa1$ are delivered.



AB=((S |[timeout, reset, setT]| T)

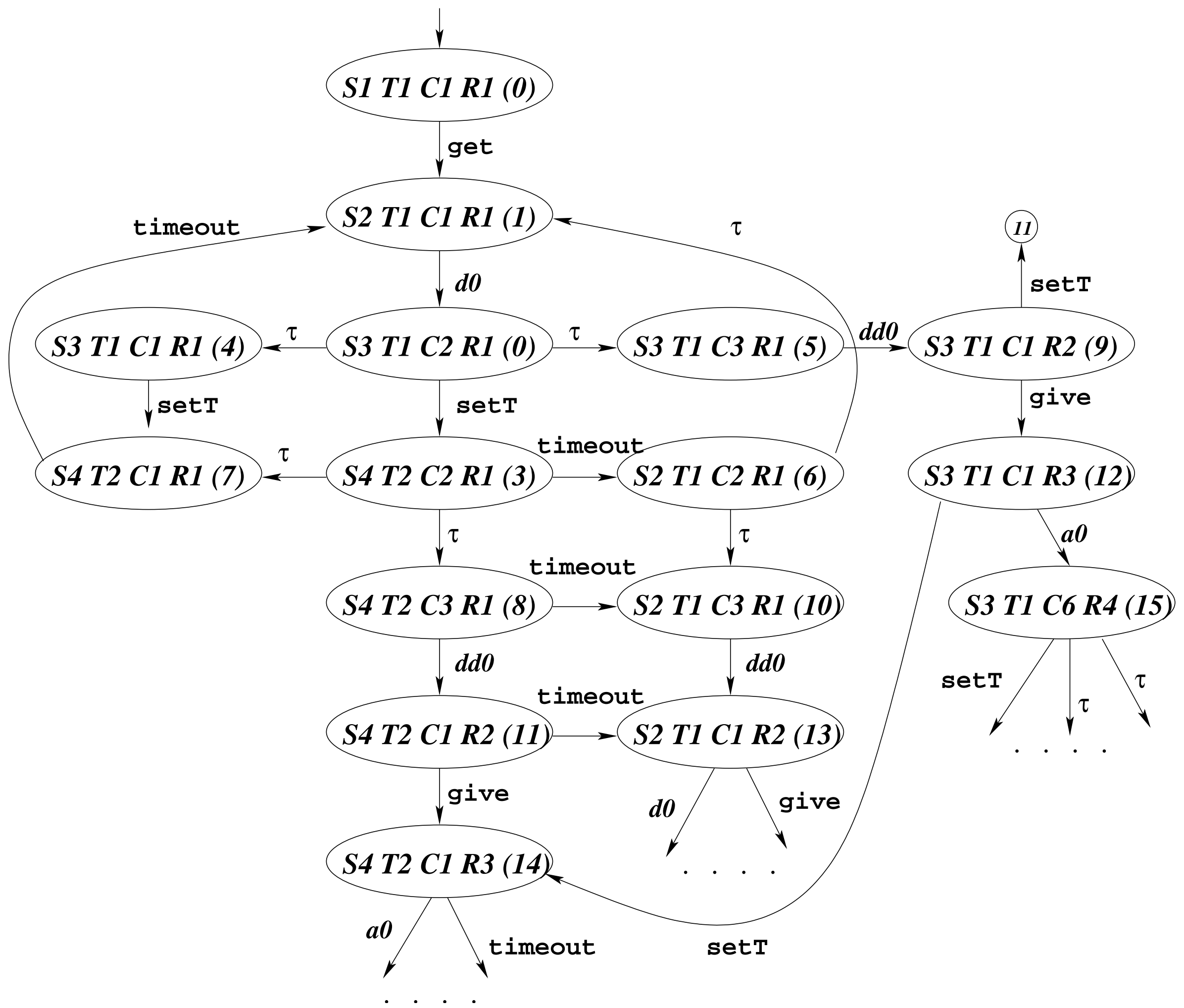
|[*d0,d1,aa0,aa1*]|

C)

|[*dd0,dd1,a0,a1*]|

R

The global state graph is now a combination of four processes. It is essentially more complicated than the earlier global state graph and that is why it is not wise to draw it completely manually. The start of the the graph is seen below.


S1 T1 C1 R1 (0)
get
S2 T1 C1 R1 (1)
timeout
τ
11
d0
S3 T1 C1 R1 (4)
τ
S3 T1 C2 R1 (0)
τ
S3 T1 C3 R1 (5)
dd0
S3 T1 C1 R2 (9)
setT
setT
setT
give
S4 T2 C1 R1 (7)
τ
S4 T2 C2 R1 (3)
timeout
S2 T1 C2 R1 (6)
S3 T1 C1 R3 (12)
τ
τ
a0
timeout
S4 T2 C3 R1 (8)
S2 T1 C3 R1 (10)
S3 T1 C6 R4 (15)
dd0
dd0
setT
τ
τ
timeout
S4 T2 C1 R2 (11)
S2 T1 C1 R2 (13)
give
d0
give
S4 T2 C1 R3 (14)
a0
timeout
setT

Chapter 5

Equivalences and Verifications

5.1 Modelling Services

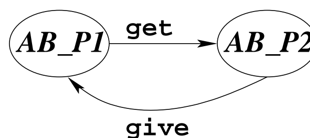
The global state graph can be generated mechanically from the descriptions of the processes in the system. If the graph is small, let us say less than one million states, it is possible to examine the graph systematically and search for mistakes, for example deadlocks, livelocks (it is not possible to back to main cycle) etc. It is difficult, however, to find all the mistakes in this way. Messages may disappear without deadlocks, the same message may be delivered two times to a receiver etc.

If a graph is too large or even infinite, it is still possible to make a random walk in the graph. If there is a mistake in a specification, it turns out often in many places in the global state graph. Even if we examine only 5 percent of the nodes, practical experiments have shown that most of the mistakes are revealed.

If we want to verify the specification more completely, we need a different approach. There are two basic approaches: equivalences and temporal logics. We consider first methods based on equivalences.

The central concept in the methods based on equivalences is *a service description.* This is a transition system which describes the service the protocol gives to its user (environment, observer). Let us examine two example.

The AB-protocol offers a data transfer service. The protocol takes data packets from the environment (upper layer) and delivers them to a receiving participant (again upper layer in another computer). Thus it is easy to describe the service of the AB-protocol:



If we base our verification on equivalences, the next step is compare the global state graph of the AB-protocol to the global state graph of the service. If they are same in some respect (specified later in more detail), the AB-protocol can be considered correct. It seems clear that we must abstract away many details from the graph of the AB-protocol. How this is done depends on the equivalence.

Our second example is the client/server system shown in the chapter 2, when there are four clients. The whole system can be described with the help of the parallel operator:

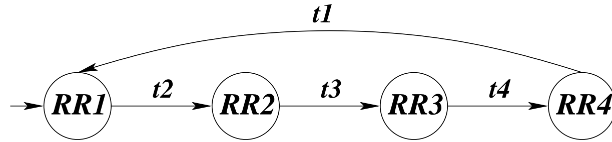
```
SystemRR :=

Server
    | [cs1, sb1, cs2, sb2, cs3, sb3, cs4, sb4] |
((Client1  |[bc1]| Buffer1  )
    | [t1, t2] |
  ((Client2  |[bc2]| Buffer2 )
    | [t3] |
  ((Client3 |[bc3]| Buffer3 )
    | [t4] |
  (Client4 |[bc4]| Buffer4 ))))
```

Suppose now that we are interested in the round robin principle. The principle is realized in the system, if the tokens ti travel in the following order:

$$t2,\ t3,\ t4,\ t1.$$

Thus the round robin principle is described by the following process.



We could now generate the global state graph of the process `SystemRR` and examine, if the token travels in that order. If we do not want to examine the graph manually, we shoul write a program that does the task. This would be time-consuming.

A quicker method is to compare the process `SystemRR` to the process `RR`. The round robin principle is realized in the system, if the functioning of `SystemRR`, abstracted in a suitable way, corresponds the functioning of `RR`. In this case, a suitable abstraction is such that all the actions except $t1$, $t2$, $t3$ ja $t4$ are changed to invisible action τ. If after this change we traverse the paths in the global state graph of `SystemRR`, then the actions ti should appear in the RR-order and all the other actions are invisible.

A strong side of the process algebras is that it is possible to define several equivalences for different purposes. Moreover, these equivalences can be effectively computed, at least in most cases. The simplest equivalence is *the trace equivalence*. One

of the most used equivalence is *the weak bisimulation equivalence* which is sufficient for the most purposes. It can be computed effectively and it is included nearly in all the verification programs. We consider only these two in this course.

5.2 Relations

The Definition of a Relation

Let A and B be sets. Every set $R \subset A \times B$ is called *a relation from the set A to the set B*. The set

$$M_R = \{ x \in A \mid \exists y \in B \text{ such that } (x, y) \in R \}$$

is *the domain* of R and the set

$$A_R = \{ y \in B \mid \exists x \in A \text{ such that } (x, y) \in R \}$$

is its *range or codomain*. If $R \subset A \times A$, i.e. if R is a relation from A to A, then it is said that R is a relation in A. If R a relation in A and if $(x, y) \in R$, then we usually use the notation xRy.

Equivalence Relations

Define first some concepts that are useful when defining equivalence relations.

Sets A and B are *disjoint*, if $A \cap B = \emptyset$. If $\mathcal{I}$ is a set, whose elements are sets, then $\mathcal{I}$ is disjoint, if any two elements in $\mathcal{I}$ are always disjoint.

If $\mathcal{H}$ is a collection of subsets of X, then it is *a partition of* X, if it satisfies the following conditions:

H1. Every $A \in \mathcal{H}$ is non-empty,

H2. the union of the sets in $\mathcal{H}$ is X,

H3. $\mathcal{H}$ is disjoint.

The concept of a partition is closely connected to the equivalence relations which are defined next.

Definition 2 *The relation R in X is* an equivalence, *if it satisfies the following conditions:*

E1. *aRa for every $a \in X$ (reflexive);*

E2. *if aRb, then bRa (symmetric);*

E3. *if aRb and bRc, then aRc (transitive).*

If $a \in X$, then the set $R(a) = \{x \in X \mid aRx\}$ is the equivalence class *of a with respect to the equivalence R.*

Theorem. *If R is an equivalence in X, then the set X/R of the equivalence classes is a partition in X. For elements a and b in X, aRb if and only if a and b belong to the same equivalence class.*

Proof. Let us go through the proof, although the same is done in the elementary courses in mathematics.

Let $a \in X$. Because aRa, then $a \in R(a)$. It follows that every equivalence class is non-empty and that the union of the classes is X. In order to prove condition H3 it is enough to show that given two equivalence classes, they are either identical or disjoint. Suppose that $R(a) \cap R(b) \neq \emptyset$. Then there exits an element $c \in R(a) \cap R(b)$. Let $x \in R(a)$, i.e. aRx. Because $c \in R(a)$, it follows aRc and also cRa, because every equivalence relation is symmetric. Because cRa and aRx, then cRx, because of transitivity. Because $c \in R(b)$, also bRc. This shows that $R(a) \subset R(b)$. Exactly in the same way it is shown that $R(b) \subset R(a)$. Thus $R(a) = R(b)$ and the union of the equivalence classes is a partition of X.

Let aRb. Then $b \in R(a)$, and thus a and b belong to the same equivalence class $R(a)$. Coversely, suppose that a ja b belong to the same class $R(c)$. Then $a \in R(c) \cap R(a)$ and hence $R(c) \cap R(a) \neq \emptyset$. Now the first part of the proof shows that $R(c) = R(a)$. It follows $b \in R(c) = R(a)$ or aRb. Thus the latter claim is true, too □

Theorem. *Let $\mathcal{H}$ be a partition of X. If we define for elements a and b of X that $aR_{\mathcal{H}}b$ if and only if a and b belong to the same set $U \in \mathcal{H}$, then $R_{\mathcal{H}}$ is an equivalence relation of X such that the union of all the $R_{\mathcal{H}}$-equivalence sets is $\mathcal{H}$.*

Proof. Let $a \in X$. Because $\mathcal{H}$ is a partition of X, there exists a set $U \in \mathcal{H}$ such that $a \in U$. Now a and a belong to the same set U so that $aR_{\mathcal{H}}a$. Thus the relation $R_{\mathcal{H}}$ is reflexive.

Let $aR_{\mathcal{H}}b$. Then a and b belong to the same set $U \in \mathcal{H}$ and hence $bR_{\mathcal{H}}a$. Thus the relation is symmetric.

Suppose now $aR_{\mathcal{H}}b$ and $bR_{\mathcal{H}}c$. There exists sets U and $V \in \mathcal{H}$ such that a and $b \in U$, and b and $c \in V$. Because $b \in U \cap V$, we have $U \cap V \neq \emptyset$ and hence $U = V$, because $\mathcal{H}$ is disjoint. Thus a and c belong to the same set $U = V \in \mathcal{H}$, and hence $aR_{\mathcal{H}}c$. The relation $R_{\mathcal{H}}$ is thus transitive and it is an equivalence relation.

Let $R_{\mathcal{H}}(a)$ be an arbitrary $R_{\mathcal{H}}$-equivalence class. Because the union of the sets in $\mathcal{H}$ is X, there exits a $U \in \mathcal{H}$ such that $a \in U$. If $x \in U$, then a and x belong to the same set $U \in \mathcal{H}$, hence $aR_{\mathcal{H}}x$ or $x \in R_{\mathcal{H}}(a)$. Coversely, suppose $x \in R_{\mathcal{H}}(a)$ or $aR_{\mathcal{H}}x$. Then a and x belong to the same set $V \in \mathcal{H}$. Because $a \in U \cap V$, we have $U \cap V \neq \emptyset$, and furthermore $U = V$, because $\mathcal{H}$ disjoint. Hence $x \in V = U$. The results show that $R_{\mathcal{H}}(a) = U$. Conversely, if $U \in \mathcal{H}$, then $U \neq \emptyset$. If $a \in U$, then, because of the previous, $R_{\mathcal{H}}(a) \in \mathcal{H}$. Because $R_{\mathcal{H}}(a) \cap U \neq \emptyset$, we have $U = R_{\mathcal{H}}(a)$. Thus the union of the$R_{\mathcal{H}}$- equivalence classes is the same as $\mathcal{H}$. □

Transitive Closure

Let R be a relation in a set V. Define the powers of a relation as follows:

$$\begin{aligned} R^0 &= \{(a,a) \mid a \in V\}, \\ R^1 &= R, \\ R^2 &= \{(a,c) \mid \exists b \in V: \ aRb \text{ ja } bRc\}, \\ R^n &= R(R^{n-1}), \ \ n > 2. \end{aligned}$$

Transitive closure is now defined with the help of the powers of a relation. *The reflexive transitive closure of a relation R, R^**, is the set

$$R^* = \bigcup_{i=0}^{\infty} R^i,$$

and *transitive closure* R^+ is the set

$$R^+ = \bigcup_{i=1}^{\infty} R^i.$$

Writing out the formulas recursively we see that aR^*b, if there exists elements $a = c_1, c_2, \cdots, c_n = b$ in V such that c_iRc_{i+1}, $i = 1, \cdots, n-1$.

Relation R in a set V can be represented as a directed graph: The node set of the graph is V and if aRb, then (a,b) is an arc in the graph. The transitive closure has an illustrative interpretation in the graph. That is, R^+ means all the pairs $(a,b) \in V \times V$ such that there is a path from a to b in the graph R. Similarly, R^* R^+ with arcs added from every node to itself.

There are many algorithms to compute the transitive closure of a given relation. However, in many situations in verification, it is not necessary to compute a transitive closure beforehand, but *on the fly*. This means that when one needs to traverse the arcs of the transitive closure starting from one node, a depth-first search is started from this node and it finds all the paths from that node. It may seem that this method is very time consuming, but it behaves very well in practice.

5.3 Trace Equivalence

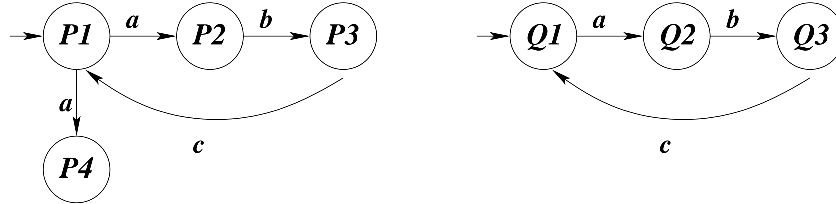
The simplest process equivalence is based on the comparisons of event sequences. Let A be the set of actions. We assume in what follows that the actions of all the processes belong to this set.

Definition 3 *Let $u \in (A\backslash\{\tau\})^*$ be an action sequence. Sequence u is* a trace *of P, if $P \overset{u}{\Longrightarrow} P'$ for some process P'. We denote the set of all the traces of P by* $\mathrm{tr}(P)$.

Definition 4 *Processes P and Q are* trace equivalent*, $P \approx_{tr} Q$, if* $\mathrm{tr}(P) = \mathrm{tr}(Q)$.

Clearly $\approx_{tr}$ is an equivalence relation. It also is compositional with respect to the parallel operator. Thus if $P \approx_{tr} P'$ ja $Q \approx_{tr} Q'$, then $P|[a_1, \cdots, a_n]|Q \approx_{tr} P'|[a_1, \cdots, a_n]|Q'$ (exercise).

If $P \approx_{tr} Q$, then there may be deadlocks in P even if Q is deadlock-free. For example, the processes P and Q below are trace equivalent.



It is generally thought that deadlocks are the most serious errors in distributed systems. That is why the trace equivalence is not always appropriate when comparing a protocol and its service. However, the trace equivalence reveals other types of mistakes quite effectively. In addition, deadlocks are easy to detect already when generating global state graphs. Thus the trace equivalence may be used sometimes. We will see its usefulness when analysing mutual exclusion algorithms. Furthermore, the trace equivalence is a starting point for the whole family of so called decorated trace equivalences which includes failure and test equivalences.

5.4 Weak Bisimulation Equivalence

The bisimulation equivalence was invented by Robin Milner and further developed by David Park at the end of the 70's and early 80's. If P and Q are processes, then

the idea of the equivalence is to simulate the behaviour of visible actions in P by Q and vice versa. If the simulation succeeds all the time, the processes are equivalent. In order to define this precisely, we need auxiliary concepts.

Let a be an action, $a \neq \tau$. Remember the notation

$$P \stackrel{a}{\Longrightarrow} P',$$

which means that there is a transition chain

$$P = P_1 \stackrel{\tau}{\longrightarrow} P_2 \stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow} P_k \stackrel{a}{\longrightarrow} P_{k+1} \stackrel{\tau}{\longrightarrow} P_{k+2} \stackrel{\tau}{\longrightarrow} P_{k+3} \stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow} P_{k+m} = P',$$

$k \geq 1$, $m \geq 0$. In other words, $P \stackrel{a}{\Longrightarrow} P'$, if there is a path from the initial state of P to the in initial state of P' and one of the arcs belonging to the path contains a, and others τ. We can also write

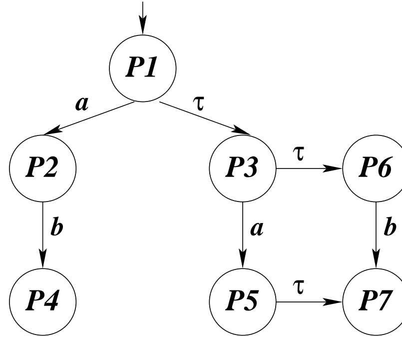
$$P \stackrel{\varepsilon}{\Longrightarrow} P'$$

if $P = P'$ or there is a chain of τ transitions

$$P = P_1 \stackrel{\tau}{\longrightarrow} P_2 \stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow} P_k = P',$$

$k > 1$.

Example. Consider the the following transition system.

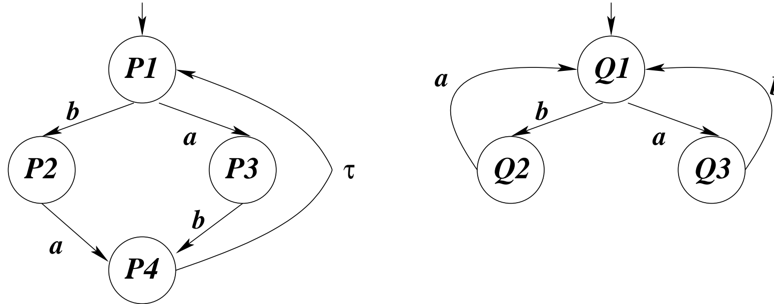


There are the following paths from $P1$: $P1 \stackrel{\varepsilon}{\Longrightarrow} P6$, $P1 \stackrel{a}{\Longrightarrow} P2$, $P1 \stackrel{a}{\Longrightarrow} P5$, $P1 \stackrel{a}{\Longrightarrow} P7$, $P1 \stackrel{b}{\Longrightarrow} P7$, $P1 \stackrel{\varepsilon}{\Longrightarrow} P3$, $P1 \stackrel{\varepsilon}{\Longrightarrow} P1$. □

Definition. *Let P and Q be processes and A the action set of P and Q. Processes P and Q are* weakly bisimilar, *$P \approx_{wbis} Q$, if there is a set $\mathcal{R}$,* weak bisimulation, *consisting of process pairs such that for every action $a \in (A \backslash \{\tau\}) \cup \{\varepsilon\}$:*

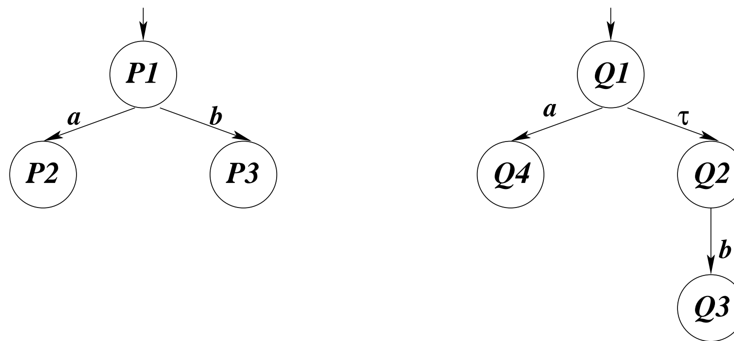
1. *$(P, Q) \in \mathcal{R}$;*
2. *if $(P_1, Q_1) \in \mathcal{R}$ ja $P_1 \stackrel{a}{\Longrightarrow} P_2$, there exists Q_2 such that $Q_1 \stackrel{a}{\Longrightarrow} Q_2$ ja $(P_2, Q_2) \in \mathcal{R}$;*
3. *if $(P_1, Q_1) \in \mathcal{R}$ ja $Q_1 \stackrel{a}{\Longrightarrow} Q_2$,there exists P_2, such that $P_1 \stackrel{a}{\Longrightarrow} P_2$ ja $(P_2, Q_2) \in \mathcal{R}$.*

Example. The following processes are weakly bisimilar, $P \approx_{wbis} Q$:



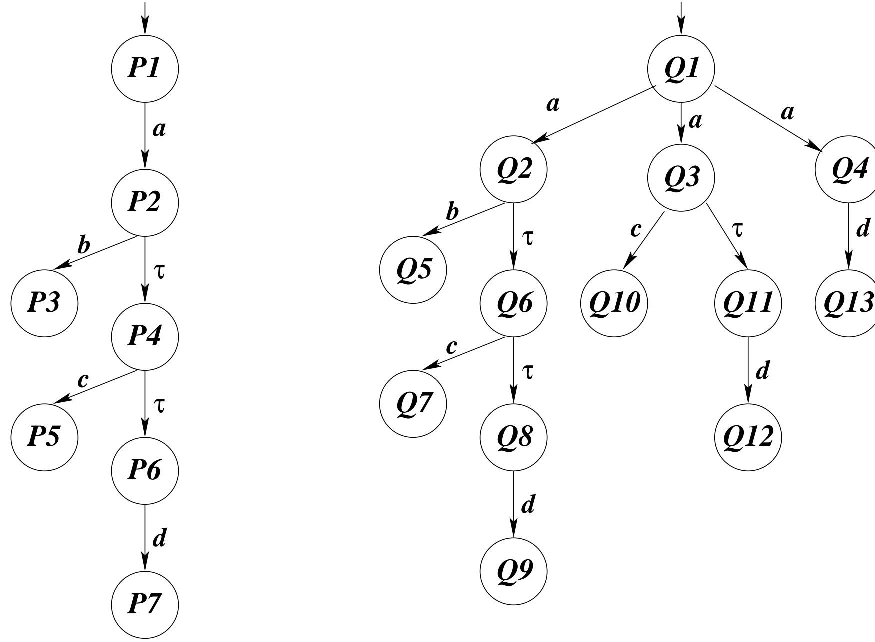
Equivalence follows, because $\mathcal{R} = \{(P1, Q1), (P2, Q2), (P3, Q3), (P4, Q1)\}$ is a weak bisimulation and
$(P, Q) \in \mathcal{R}$ $(P = P1,\ Q = Q1)$. □

Example. The following processes are not weakly bisimilar:



If we try to construct a weak bisimulation $\mathcal{R}$, then the pair $(P1, Q1)$ must belong to that relation . Let us use next the condition 3 in the definition: Q makes a transition from $Q1$ to $Q2$ using an internal action. The only way to simulate this transition in P is such that we must stay in $P1$. Hence the pair $(P1, Q2)$ must belong to the bisimulation relation $\mathcal{R}$. After this, we apply the condition 2 to the pair $(P1, Q2)$: P moves from $P1$ with a to $P2$. Now Q cannot simulate this transition, because there are no a-transitions from $Q2$. Thus there cannot exists a weak bisimulation between P and Q and hence the processes are not bisimilar. □

Example. The following processes are weakly bisimilar:



The weak bisimulation is as follows:

$$\begin{array}{l}(P1,Q1),\ (P2,Q2),\ (P4,Q3),\ (P6,Q4),\\ (P3,Q5),\ (P4,Q6),\ (P5,Q7),\ (P6,Q8),\ (P7,Q9)\\ (P5,Q10),\ (P6,Q11),\ (P7,Q12),\ (P7,Q13)\end{array}$$

□

Theorem. *The relation $\approx_{wbis}$ is an equivalence relation between transition systems.*

Proof. We must show that the relation $\approx_{wbis}$ is reflexive, symmetric, and transitive.

If $\approx_{wbis}$ is reflexive, then we should have $P \approx_{wbis} P$ for all processes P. Symmetry means that from $P \approx_{wbis} Q$ it follows that $Q \approx_{wbis} P$. Both properties follow directly from the definition.

We still must show that the relation is transitive, i.e. from the conditions $P \approx_{wbis} Q$ ja $Q \approx_{wbis} R$ it follows that $P \approx_{wbis} R$. Let $\mathcal{R}$ a bisimulation between P and Q, $\mathcal{S}$ a bisimulaatio between$Q$ and R. Construct the set $\mathcal{T}$ of process pairs as follows:

$$\mathcal{T} = \{(P_1, R_1) |\ \exists Q_1 :\ (P_1, Q_1) \in \mathcal{R},\ (Q_1, R_1) \in \mathcal{S}\}.$$

Let us show that $\mathcal{T}$ is a weak bisimulation between P and R. Because of the definition of $\mathcal{T}$ we have $(P, R) \in \mathcal{T}$. Let $(P_1, R_1) \in \mathcal{T}$ an arbitrary element and $P_1 \overset{a}{\Longrightarrow} P_2$. We know that there exists Q_1 such that $(P_1, Q_1) \in \mathcal{R}$ ja $(Q_1, R_1) \in \mathcal{S}$.

Because $\mathcal{R}$ and $\mathcal{S}$ are weak bisimulations, there exist processes Q_2 and R_2 such that $Q_1 \overset{a}{\Longrightarrow} Q_2$ and $(P_2, Q_2) \in \mathcal{R}$, and further $R_1 \overset{a}{\Longrightarrow} R_2$ ja $(Q_2, R_2) \in \mathcal{S}$. But by the definition of $\mathcal{T}$ we have $(P_2, R_2) \in \mathcal{T}$, and thus the condition 2 has been shown. The condition 3 is shown in the same way. $\Box$

Denote by $\mathcal{P}$ the set of all finite processes or labelled transition systems. Relation $\approx_{wbis}$ thus defines an equivalence relation in $\mathcal{P}$. In mathematics, an equivalence relation is defined as a subset of the cartesian product of the basic set with itself. Thus in our case it should be that $\approx_{wbis} \subset \mathcal{P} \times \mathcal{P}$. We could indeed define that $\approx_{wbis}$ is the maximal bisimulation

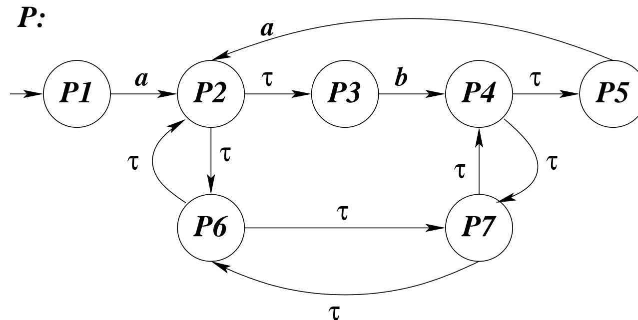
$$\approx_{wbis} = \bigcup \{\mathcal{R} | \ \mathcal{R} \text{ on bisimulaatio}\}.$$

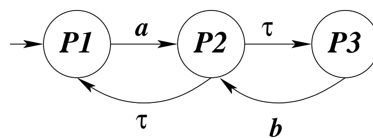
If P is a process, then the equivalence class $[P]_{\approx_{wbis}}$ of P is the set of those processes equivalent with P. The classes $[P]_{\approx_{wbis}}$ form a partition of $\mathcal{P}$.

Minimal Process

If P is a labelled transition system $(S, A, \longrightarrow, s_0)$, then every state $s \in S$ can be considered as a transition system which is the same as P, but the initial state is now s. It is now possible to restrict the equivalence $\approx_{wbis}$ to the set $S \times S$. Then $\approx_{wbis}$ is an equivalence relation in S and it partitions S into equivalence classes. We can now form a new transition system whose states are the equivalence classes. There is an arc with a from one class to another, if there is a state s_1 in the first class and a state s_2 in the second class and an arc $s_1 \overset{a}{\longrightarrow} s_2$ in the original transition system. The transition system constructed in this way is the minimal transition system among those systems that are weak bisimulation equivalent with the original one.

For example, the following processes are equivalent and the latter is minimal.





The equivalence classes of P are $E_1 = \{P1, P5\}$, $E_2 = \{P2, P4, P6, P7\}$ ja $E_3 = \{P3\}$.

If we draw all the transitions mechanically from one class to another, the result may contain too many useless arcs. However, the minimization of arcs is more complicated than that of states. More details can be found in the dissertation of Jaana Eloranta and in the article Eloranta, Tienari, Valmari: Essential Transitions To Bisimulation Equivalences, Theoretical Computer Science 179 (1997) 397-419.

Weak Bisimilarity and the Parallel Operator

The weak bisimulation equivalence behaves well with respect of the parallel operator.

Theorem. *If $P \approx_{wbis} Q$, then $P\ |B|\ R \approx_{wbis} Q\ |B|\ R$ for all action sets B and processes R.*

Proof. Let

$$\mathcal{R} = \{(P_1|B|P_3, P_2|B|P_3) \mid P_1 \approx_{wbis} P_2\}.$$

Let us show that $\mathcal{R}$ is a weak bisimulation between $P|B|R$ and $Q|B|R$. Because $P \approx_{wbis} Q$, we have $(P|B|R, Q|B|R) \in \mathcal{R}$. Let $P|B|R \stackrel{a}{\Longrightarrow} P'|B|R'$. We have to check two cases.

i) $a \in B$. Now $a \neq i$ and $P \stackrel{a}{\Longrightarrow} P'$, $R \stackrel{a}{\Longrightarrow} R'$. Because $P \approx_{wbis} Q$, there exists a weak bisimulation $\mathcal{E}$ between P and Q. We know, because of the definition of the weak bisimulation, that there exists Q' such that $Q \stackrel{a}{\Longrightarrow} Q'$ and $(P', Q') \in \mathcal{E}$. Hence also $P' \approx_{wbis} Q'$. We can directly see that $Q|B|R \stackrel{a}{\Longrightarrow} Q'|B|R'$. The definition of $\mathcal{R}$ implies now that $(P'|B|R', Q'|B|R') \in \mathcal{R}$.

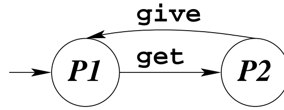
ii) $a \notin B$. Now either $P \stackrel{a}{\Longrightarrow} P'$ and $R \Longrightarrow R'$ or $P \Longrightarrow P'$ and $R \stackrel{a}{\Longrightarrow} R'$. Notice that a can be ε. If $P \stackrel{a}{\Longrightarrow} P'$, then also $Q \stackrel{a}{\Longrightarrow} Q'$ and $P' \approx_{wbis} Q'$, as we saw in the previous case. Thus $Q|B|R \stackrel{a}{\Longrightarrow} Q'|B|R'$ and $(P'|B|R', Q'|B|R') \in \mathcal{R}$. If, on the hand, $R \stackrel{a}{\Longrightarrow} R'$, $P \Longrightarrow P'$, then there also exists Q' such that $Q \Longrightarrow Q'$ and $P' \approx_{wbis} Q'$. Furthermore $Q|B|R \stackrel{a}{\Longrightarrow} Q'|B|R'$. Now it is true that $(P'|B|R', Q'|B|R') \in \mathcal{R}$.

Thus $\mathcal{R}$ satisfies the conditions of the weak bisimulation as for the transitions of in $P|B|R$. The transitions in $Q|B|R$ are handled in the same way and we can conclude that $\mathcal{R}$ is a weak bisimulation. □

5.5 Examples

5.5.1 AB-protocol

Consider the version of the AB-protocol where there are the messages `get` and `give`. This version defines a service which is easily described:



Let us apply the weak bisimulation equivalence to check the correctness of the protocol. Thus we have to show that the global state graph of the protocol is equivalent with the service. Evidently this is not true, if we do not change the global state graph in some way. One usual way is to hide some actions. For this purpose we define the operation `hide`:

```
hide a1, a2, ..., an in P
```

transforms P in such a way that all the actions ai, $i = 1, \cdots, n$, in P are replaced with τ.

Now we can phrase the verification problem of the AB-protocol in the following form. We have to show that

$$\begin{array}{ll} \texttt{hide} & d0, dd0, d1, dd1, a0, aa0, a1, aa1, st, rt, t \text{ in } \texttt{AB-protocol} \\ \approx_{wbis} & \texttt{AB-service}. \end{array}$$

The global state graph of the AB-protocol is a little too large for a manual construct so that it is necessary to use software to generate it. In order to do this, we must first write the protocol and its service in some spesification language. After this the software generates the global state graphs of the protocol and service and finally compares these two to check if they are equivalent or not. We will do all this after we have studied Lotos.

5.5.2 FE-protocol

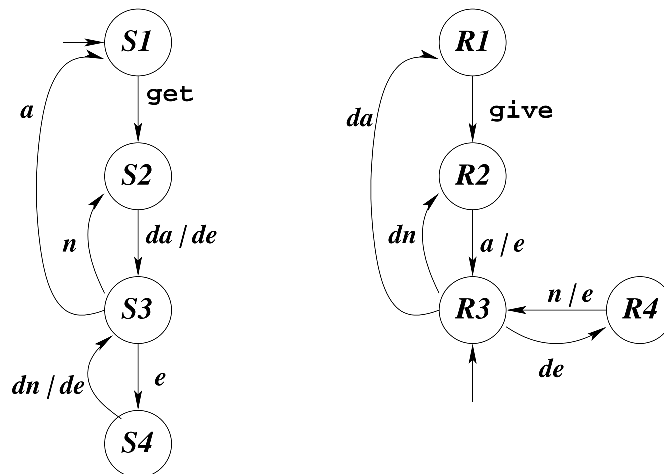
In the old protocol articles there was one erroneous protocol which has been in practical use. (W. C. Lynch: Reliable full-duplex transmission over half-duplex lines, Comm. ACM, Vol. 11, No. 6, pp 362-372, June 1968). Its mistakes appeared so seldom that the errors were not observed in testing, but sometimes they were encountered in the production use. It is a protocol that is illustrative to analyse.

Our protocol, shortly FE-protocol, is a symmetric link layer protocol. Two participants S and R change messages alternatively using a half-duplex channel. Acknowledgements a (positive acknowledgement) are added into every message and this tells that the previous message has arrived correctly. The acknowledgement is n (negative ackknowledgement), if the previous message was distorted in the channel.

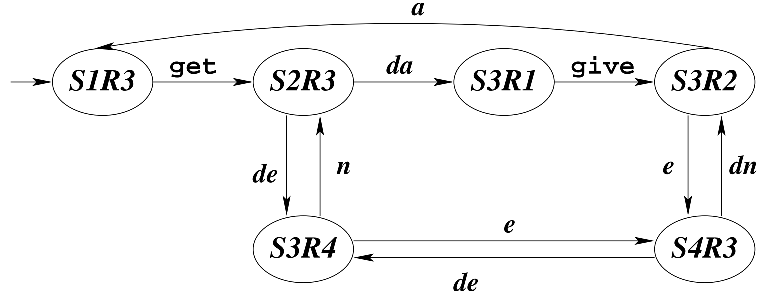
The sending logic was as follows: If the previous message contained n or it was distorted (maybe n was destroyed), send the earlier message again. Otherwise send a new message. In both cases ACK or NAK is added into the message according to the situation.

The receiving logic is not given explicitly in the article. The article shows only that it is impossible to design a receiving logic so that the protocol works correctly. We must use some logic and one possible logic would be as follows: When a message arrives correctly and contains ACK, it is delivered to the client. If a message contains NAK, the message is not delivered further (it is assumed that the message is a repetition of an older message).

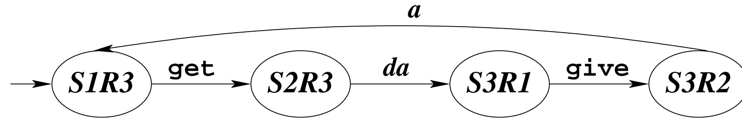
First we build the modell of the protocol in a simplified form. If the protocol behaves incorrectly in this model, it behaves incorrectly in a more general model, too. If, on the other hand, the simplified version works correctly, it is still necessary to check the behaviour of the more general version. In the simplified version the communication is synchronous, we do not use separate channels. Data is sent only from S to R. The notation da means data with a positive ackknowledgement, and dn is data with a negative acknowledgement. R will send only acknowledgements a or n. The notations de and e represent distorted data and acknowledgement, respectively. The labelled transition graphs S and R are given below:



The global state graph is in the following diagram:



We can see the basic cycle of the protocol,



which describes the messages when the communication is error-free. The protocol recovers from a single communication error using an additional path:

$$S2R3 \xrightarrow{de} S3R4 \xrightarrow{n} S2R3$$

or

$$S3R2 \xrightarrow{e} S4R3 \xrightarrow{dn} S3R2$$

Two sequential communication errors may cause that the protocol does not behave correctly. This we can see, if we traverse the following path in the global state graph:

$$S1R3 \xrightarrow{\mathsf{get}} S2R3 \xrightarrow{de} S3R4 \xrightarrow{e} S4R3 \xrightarrow{dn} S3R2 \xrightarrow{a} S1R3 \xrightarrow{\mathsf{get}} S2R3 \xrightarrow{da} S3R1 \xrightarrow{\mathsf{give}} S3R2$$

In this scenario two data messages are sent, but only one of them is really delivered to the receiver. The protocol may thus lose messages. Moreover, it is possible that the protocol delivers the same message to the receiver:

$$S1R3 \xrightarrow{\mathsf{get}} S2R3 \xrightarrow{da} S3R1 \xrightarrow{\mathsf{give}} S3R2 \xrightarrow{e} S4R3 \xrightarrow{de} S3R4 \xrightarrow{n} S2R3 \xrightarrow{da} S3R1 \xrightarrow{\mathsf{give}} S3R2.$$

The former analyses were based on a detailed scrutiny of the global state graph. This erroneous behaviour can also be detected automatically. For this purpose we need the service description which happens to be the same as in the AB-protocol. With the help of a software we can show that

$$\texttt{hide}\ da, de, a, n, e\ \texttt{in}\ \mathrm{FE}-\mathrm{protocol}\ \not\approx_{wbis}\ \mathrm{FE}-\mathrm{service}.$$

It is easy to see this manually, too. Let us try to construct a weak bisimulation:

$$\mathcal{R} = \{(S1R3, P1), (S2R3, P2), (S3R2, P2), (S1R3, P2) \cdots\}.$$

Now `take` can be done in state $S1R3$, but in $P2$ only `give` is possible. Thus it is not possible to construct a weak bisimulation and the processes are not equivalent.

5.6 Conclusions and Problems

It is possible to develop the formalisms based on labelled transition systems further. If we will apply these methods in practice, we must pay attention on the following points:

1. How are labelled transition systems represented? Diagrams are not suitable for computers.
2. How is the collaboration of several processes described? We must decide the following questions:
   - how is time modelled;
   - is the communication synchronous or asynchronous;
   - is multisynchronization allowed;
3. How is data taken into account in the messages?
4. Transitions may depend on the content of data. How is this handled?
5. In the modelling of real time systems we need time constraints. How are these expressed in the formalism?
6. Transitions may have completely different probabilities (for example, the probability of a communication error may be very small). Is it wise to assume that all the transitions happen with the same probability?
7. In ordinary transition systems synchronization points (messages) are known beforehand and their use is fixed when the spesification is written. There are, however, a lot of applications, where the synchronization points are dynamically created. For example, when one needs services, it may be that the address or permit for the service is obtained first from some authority, who answers by giving the port or address of the service. How are these kinds of situations taken into account?

All the previous tasks have been implemented in one form or another. Especially, the items 1-4 have been solved in a generally accepted way.

Labelled transition systems are usually described with the help of process algebras, for example Milner's CCS, Hoare's CSP, and Bergstra's and Klop's ACP. In these, the items 1) and 2) have been solved in a similar way:

- A labelled transition system is described using algebraic expressions.
- Cycles are generated with the help of recursion. Recursion also enables the definitions of an infinite systems. Furthermore, it makes possible to start processes dynamically.
- Processes communicate synchronously. It means that a process cannot finish the sending of a message before the receiver has really received the message. Asynchronous communication is achieved by using separate channel processes as we saw in the AB-protocol.
- Concurrent actions are performed sequentally using interleaving:
  - Events are atomic.
  - If actions a and b are executed concurrently, then we think that either a happens before b or b before a. Thus we have two possible execution sequences ab and ba which are present in the global state graph of a system.
  - Because concurrency is modelled using interleaving, there are a lot of different alternatives for execution sequences. Thus global state graphs tend to be large (abc, acb, cab, bac, bca, cba).
- In CCS, only two processes can synchronize with each other whereas in CSP and ACP multi-synchronization is possible.

In process algebras, it is possible to state conditions for transitions. On the other hand, it is not possible directly to express real time constraints or probabilities in these traditional process algebras. There are newer formalisms which tackle these questions.

Lotos has many features from CCS and CSP. It has

- synchronous communication,
- multi-synchronization,
- interleaving semantics.

In Lotos, it is possible to combine processes in a more general way than in the previous process algebras. This causes both advantages and disadvantages. For example, in the previous process algebras it is possible to prove useful algebraic properties for parallel operators. Lotos has not these properties (for example the operator is not associative).

The representation of data is the most distinct feature in Lotos. In Lotos, data is defined using the algebraic specification of abstract data types. This algebraic specification style was developed since 1970 and it was closely connected to denotational semantics. In algebraic specification, the meaning, or semantics, is created by defining relations the operators satisfy. This is a very powerful technique. For example, it is possible to defined natural numbers without basing the definition on

any other constructions. On the other hand, the data types thus defined are very inefficient (for example, number 3 is $succ(succ(succ(0)))$). In this course, we skip the data part of Lotos.

Lotos is practice-oriented. It is larger than the theoretical languages CCS, CSP, and ACP. Especially the data definition part of Lotos is different. Even if we concentrate solely on basic Lotos, it is not wise to use only that part in practice. In practical verifications, full Lotosa should be used, because in this way the descriptions become clearer and more concise.

There are also extensions of Lotos (E-Lotos). They include real time properties. Furthermore, dynamical synchronization is essential and there have been suggestions to include it into Lotos, but the results have not been successful. Instead, CCS has been extended to include dynamical synchronization and this extension is known by name π-calculus.

There are many more process algebras nowadays. They have been designed for specific purposes. *Ambient calculus* describes concurrent systems with mobility. There can be two kinds of mobility: devices move or code moves. Ambient calculus can handle both. *PEPA* is a stochastic process algebra which has been designed for modelling computer and communication systems. It has probabilistic branching and transitions may have timed conditions. *Fusion calculus* is a modification of π-calculus. *Spi-calculus*, also a modification of π-calculus, has been designed for the verification and analysis of security properties.

Finally, we should mention Robin Milner's latest invention, *bigraphs*. They are designed to be a platform for ubiquitos computing systems. Bigraphs are powerful objects and it is possible to describe various structures, even biological structures, with the help of them.

Chapter 6

Basic Lotos

6.1 Introduction

First, let us clarify the terminology. Assume that a labelled transition system contains a transition $S_i \xrightarrow{a} S_j$. Symbol a is often called *action*. When we take into account the point where the action happens, we speak about *an event*. For example, if a process has an execution sequence *abcaade*, then it contains action a and three different events with a. Sometimes it is difficult to separate events from each other. In these cases it is possible to use extra marks to denote the event, for example subscripts. Thus in the sequence *abcaade* we have events a_1, a_2 ja a_3.

In Lotos, an action is divided into two parts, *gate* and *data*. It is thought that a gate is like a socket and data, i.e. data packet, is sent and received through that socket. Thus in full Lotos a could be $g!2?x$: Boolean, where g is a gate, !2 means that 2 is sent synchronously through g and the last part means that Boolean data, coming at the same time synchronously through g, is received into variable x. However, in basic Lotos we do not use data part at all. Thus it is same, if we speak about a gate or a data.

Simplifying a little, we can say that basic Lotos is an algebraic way to describe labelled transition systems. It resembles CCS and CSP and it contains:

- two ready-made Lotos processes, **stop** and **exit**,
- mechanism how to call a process,
- two ready-made actions:
  - i or *silent* or *internal* action,
  - δ or *succesful termination action*,
- 9 operators.

A specifier can use the internal action i freely. On the contrary, δ is used only when defining the behaviour of the operators. With the help of the operators and

action names it is possible to construct all the processes (what can be constructed in Lotos). Next we define the operators.

6.2 Ready-Made Processes

Lotos has two processes that have been defined beforehand, **stop** and **exit**. **Stop**, inaction, is a stopped or empty process that does nothing. **Exit** is a successful termination. It informs its environment of a successful termination by sending δ and then stops.

We define the behaviour of the operators formally with the help of *transition rules*. The rules give *operational semantics* to the operators. A rule has three parts:

- process expression P,
- action a which can be activated in P,
- a new expression Q which is got after P has done a.

We use the notation $P \xrightarrow{a} Q$.

Expression **stop** has no transition rules. Thus it can not make no actions or transitions.

Prosess **exit** is defined by the rule

$$\mathbf{exit} \xrightarrow{\delta} \mathbf{stop}.$$

6.3 Action Prefix

If P is a process and a an action, then we can construct a new process $a; P$. It makes first s and continues then as P. The transition rule is simple:

$$a; P \xrightarrow{a} P.$$

It is not allowed to write $P; Q$ or $\delta; P$, where P and Q are Lotos prosesses.

We can already specify simple system in Lotos:

Example 1. Let a transition system be as follows:

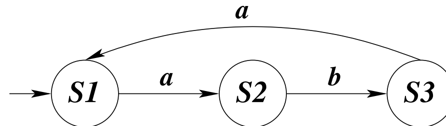
$$\longrightarrow S1 \xrightarrow{a} S2 \xrightarrow{b} S3 \xrightarrow{c} S4.$$

An equivalent Lotos process is

```
process P[a,b,c] := a; b; c; stop endproc
```

We see the formal definition of a Lotos process. The definition is started with the reserved word **process**. Then come the name of the process and the action or gate names in brackets. After := the Lotos expression is written and the reserved word **endproc** is at the end.

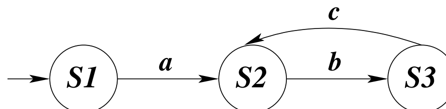
Example 2. A cycle can be realized using recursion. Consider the transition system below:



An equivalent process is

```
process Q[a,b] := a; b; a; Q[a,b] endproc
```

Esimerkki 3. One more example.




```
  process R[a,b,c] := a; S[b,c]
where
  process S[b,c] := b;c; S[b,c] endproc
  endproc
```

Usually it is not wise to draw first a labelled transition system and after this translate it mechanically into Lotos. It is better at once to design a Lotos process. The further operators make this task even more straightforward.

6.4 Hiding

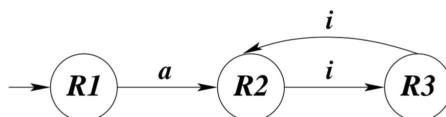
Hiding changes visible actions into internal actions. The syntax of the operator is as follows:

$$\text{hide } a_1, a_2, \cdots, a_n \text{ in } P$$

This process behaves as P, but makes the action i, when P makes some of the actions a_i. The transition rules are easy to understand:

- If $P \xrightarrow{a} Q$ and $a \in \{a_1, \cdots, a_n\}$, then
$$\text{hide } a_1, \cdots, a_n \text{ in } P \xrightarrow{i} \text{hide } a_1, \cdots, a_n \text{ in } Q.$$
- If $P \xrightarrow{a} Q$ and $a \notin \{a_1, \cdots, a_n\}$, then
$$\text{hide } a_1, \cdots, a_n \text{ in } P \xrightarrow{a} \text{hide } a_1, \cdots, a_n \text{ in } Q.$$

Example. If R is as in the example 3 of the previous section, then process hide b, c in R as a transition system is



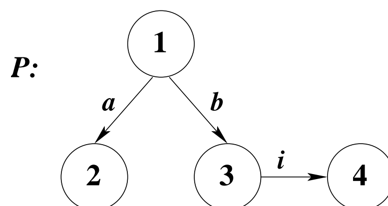
6.5 Choice

The symbol of the choice operator is '[]', for example P [ ] Q. The system composed by using the choice operator is non-deterministic in its initial state. The first event may be either the event of P or Q. After the first event, the process continues as P or Q would continue. This can be expressed using transition rules:

- If $P \xrightarrow{a} P'$, then $P \; [\,] \; Q \xrightarrow{a} P'$.
- If $Q \xrightarrow{a} Q'$, then $P \; [\,] \; Q \xrightarrow{a} Q'$.

Now it is possible to describe any kind of transition systems.

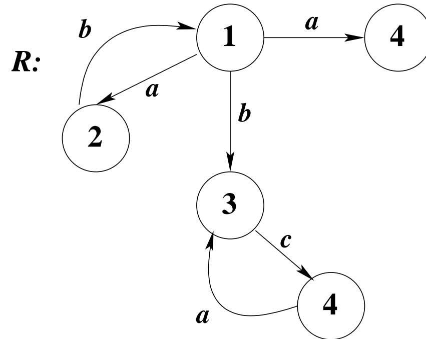
Example 1. Let a transition system be as follows:



The corresponding Lotos expression is

```
process P[a,b] := (a; stop) [] (b; i; stop) endproc
```

Example 2. A labelled transition system is now:



The corresponding Lotos expression is

```
process R[a,b,c] := (a; b; R[a,b,c]) []
                    (b; S[a,c])      []
                    (a; stop)
where

    process S[a,c] := c; a; S[a,c] endproc
endproc
```

We show how the process changes when we execute the actions b,c,a,c:

```
(a; b; R[a,b,c]) [] (b; S[a,c]) [] (a; stop) --b-->
 c; a; S[a,c]   --c-->
 a; S[a,c]      --a-->
 c; a; S[a,c]   --c-->
 a; S[a,c]
```

6.6 Parallel Operator

So far we can create only sequential and non-deterministic processes. With the help of the parallel operator we can also model concurrent behaviours. If P and Q are processes, then the notation

$$P \ |[a_1, \cdots, a_n]| \ Q$$

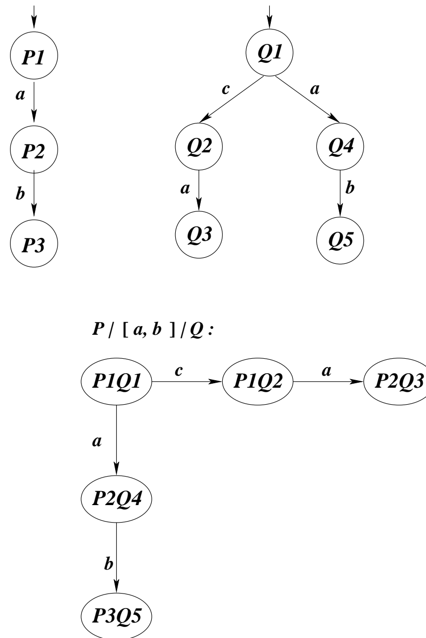
means as follows:

- P and Q proceed concurrently.
- Actions $a_1, \cdots, a_n$ and δ are possible only, if both P and Q make them at the same time ($a_1, \cdots, a_n$ are synchronizing actions or gates).
- P and Q perform other actions independently of each other.

The semantics of the parallel operator is defined by the following rules. It is assumed that $a_i \neq i, \delta,\ i = 1, \cdots, n$.

- If $P \xrightarrow{a} P'$ and $a \notin \{\delta, a_1, \cdots, a_n\}$, then
$$P\ \ |[a_1, \cdots, a_n]|\ \ Q \xrightarrow{a} P'\ \ |[a_1, \cdots, a_n]|\ \ Q.$$
- If $Q \xrightarrow{a} Q'$ and $a \notin \{\delta, a_1, \cdots, a_n\}$, then
$$P\ \ |[a_1, \cdots, a_n]|\ \ Q \xrightarrow{a} P\ \ |[a_1, \cdots, a_n]|\ \ Q'.$$
- If $P \xrightarrow{a} P'$, $Q \xrightarrow{a} Q'$ and $a \in \{\delta, a_1, \cdots, a_n\}$, then
$$P\ \ |[a_1, \cdots, a_n]|\ \ Q \xrightarrow{a} P'\ \ |[a_1, \cdots, a_n]|\ \ Q'.$$

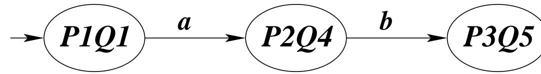
If we applied the above rules, then also $P\ |[a_1, \cdots, a_n]|\ Q$ can be considered as a transition system, the global state graph or reachability graph of the system consisting of P and Q. The next example shows this.



If $[a_1, \cdots, a_n]$ consists of all the actions in P and Q ($\neq i$), then we can use the notation

$$P \;||\; Q.$$

If P and Q are as the previous example, then $P \;||\; Q$ consists only of transitions:

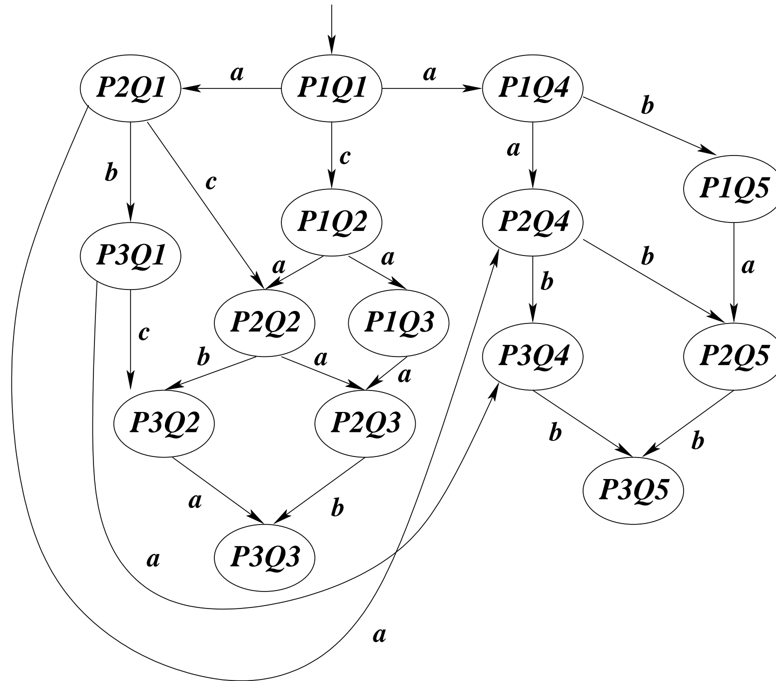


Q can not now to perform c, because c belongs to the synchronization set, but P does not have it.

If the synchronization set is empty, we can use the notation

$$P \;|||\; Q.$$

This is called *pure interleaving*, because P and Q can make all the actions alone, independently of the other process. If there are a lot of non-synchronizing actions in a system, then the global state graph will often be very large. If we consider the previous examples, then the pure interleaving produces the following graph:



These examples show how concurrency is modelled. First of all, processes proceed independently of each other, interleaving their events. In synchronization events, both processes perform the same action at the same time. A process cannot proceed before the other has reached the same phase where it can perform the action. The synchronization is of the form of rendezvous. This kind of synchronous communication is suitable for applications that take place in the same machine or that are really such that processes must wait for other processes before they proceed.

Often the communication is asynchronous. It is possible to model these kind of systems also as systems with synchronous communication, but then it is not possible to detect all the mistakes. However, if a mistake is found in the synchronouos version, then it is in asynchronous version as well. But if we really want to analyse asynchronous versions, then we must model a channel as a separate process.

Before there was a lot of discussion about the interleaving. There are other approaches, so called true concurrency models. Lotos has, for example, this kind of semantics in addition to the interleaving semantics. True concurrency models seem to be quite complicated and that is why they are seldom applied in concrete verifications.

6.7 Sequental Composition

The notation for sequental composition is '>>'. Process $P >> Q$ acts as P until P terminates successfully, and continues after this as Q. The termination of P is expressed using the internal action. Transitions rules are simple:

- If $P \xrightarrow{a} P'$ and $a \neq \delta$, then
$$P \;>>\; Q \xrightarrow{a} P' \;>>\; Q.$$
- If $P \xrightarrow{\delta} P'$, then
$$P \;>>\; Q \xrightarrow{i} Q.$$

Example.

$$\begin{array}{rl} a;\mathbf{exit} \;>>\; b;\mathbf{exit} & \xrightarrow{a} \\ \mathbf{exit} \;>>\; b;\mathbf{exit} & \xrightarrow{i} \\ b;\mathbf{exit} & \xrightarrow{b} \\ \mathbf{exit} & \xrightarrow{\delta} \\ \mathbf{stop} & \end{array}$$

6.8 Disabling

The notation for disabling is $P\;[>\;Q$. First, the system acts as P. At any time before P has terminated successfully, Q can start. If Q starts, P terminates, but not successfully. Q cannot start, if P has terminated successfully. Transition rules:

- If $P \xrightarrow{a} P'$ and $a \neq \delta$, then
$$P\;[>\;Q \xrightarrow{a} P'\;[>\;Q.$$

- If $P \stackrel{\delta}{\longrightarrow} P'$, then
$$P\,[>\;Q \stackrel{\delta}{\longrightarrow} P'.$$
- If $Q \stackrel{a}{\longrightarrow} Q'$, then
$$P\,[>\;Q \stackrel{a}{\longrightarrow} Q'.$$

Example.

$$\begin{array}{ccc}
a;\; b;\; \mathbf{exit}\,[>\;r;\;\mathbf{stop} & \stackrel{r}{\longrightarrow} & \mathbf{stop} \\
\downarrow a & & \\
b;\; \mathbf{exit}\,[>\;r;\;\mathbf{stop} & \stackrel{r}{\longrightarrow} & \mathbf{stop} \\
\downarrow b & & \\
\mathbf{exit}\,[>\;r;\;\mathbf{stop} & \stackrel{r}{\longrightarrow} & \mathbf{stop} \\
\downarrow \delta & & \\
\mathbf{stop} & &
\end{array}$$

6.9 The Precedence of the Operators

The operator precedences are as follows:

action prefix > choice > parallel composition > disabling > enabling > hiding.

Thus
$$\mathbf{hide}\ a\ \mathbf{in}\ a; P\ [\,]\ Q\ \ >>\ \ R\ ||\ S\ [>\ T$$
is the same as
$$\mathbf{hide}\ a\ \mathbf{in}\ (((a; P)[\,]Q) >> ((R\ ||\ S)[>\ T)).$$
Operators with the same precedence are grouped starting from the right, as in
$$P\ |[a]|\ Q\ |[b]|\ R\ =\ P\ |[a]|\ (Q\ |[b]|\ R).$$

6.10 Process Instantiation

Calling processes is generally the same as the procedure call in programming languages, but in Lotos even this defined formally.

For this formal definition, we need one extra operator *relabelling*. It is denoted by
$$[b_1/a_1, b_2/a_2, \cdots, b_n/a_n].$$

It means that the gates $a_1, \cdots, a_n$ in the process are replaced by the gates $b_1, \cdots, b_n$. Relabelling does not belong to Lotos, but it is used only in the definition of process instantiation. The semantics of the operator is defined by the following transition rules:

- If $P \xrightarrow{a} P'$ and $a = a_i \in \{a_1, \cdots, a_n\}$, then
$$P[b_1/a_1, \cdots, b_n/a_n] \xrightarrow{b_i} P'[b_1/a_1, \cdots, b_n/a_n].$$
- If $P \xrightarrow{a} P'$ and $a \notin \{a_1, \cdots, a_n\}$, then
$$P[b_1/a_1, \cdots, b_n/a_n] \xrightarrow{a} P'[b_1/a_1, \cdots, b_n/a_n].$$

With the help of this operator we can write the rules for process instantiation. Assume that process P is defined by the expression

$$\textbf{process}\ P[a_1, \cdots, a_n] := KL\ \textbf{endproc}$$

where KL is a Lotos process. If there is a transition

$$KL[b_1/a_1, \cdots, b_n/a_n] \xrightarrow{a} Q$$

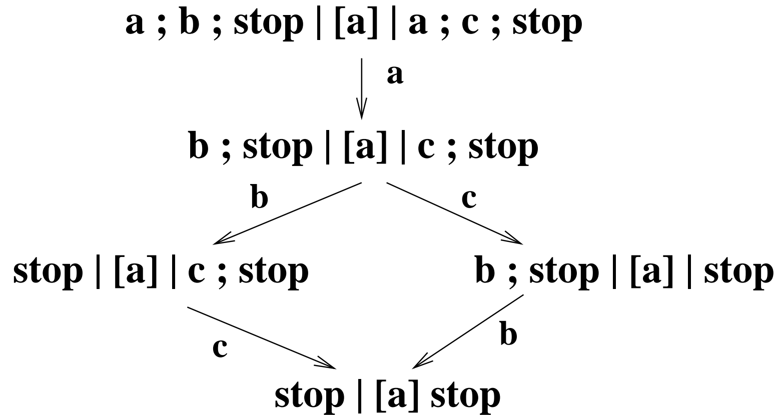
then there is the transition

$$P[b_1, \cdots, b_n] \xrightarrow{a} Q,$$

too. This rule defines process instantiation. In most cases the instantiation behaves as expected. Sometime one must, howver, be careful. We should notice that the process instantiation is dynamic. Relabelings are done continuously during the execution of the process, not statically in the beginning. Consider for example the following process:

```
process P[a,b,c] := a; b; stop |[a]| a;c; stop endproc.
```

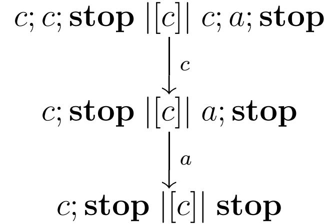
As a transition systen it is of the following form:



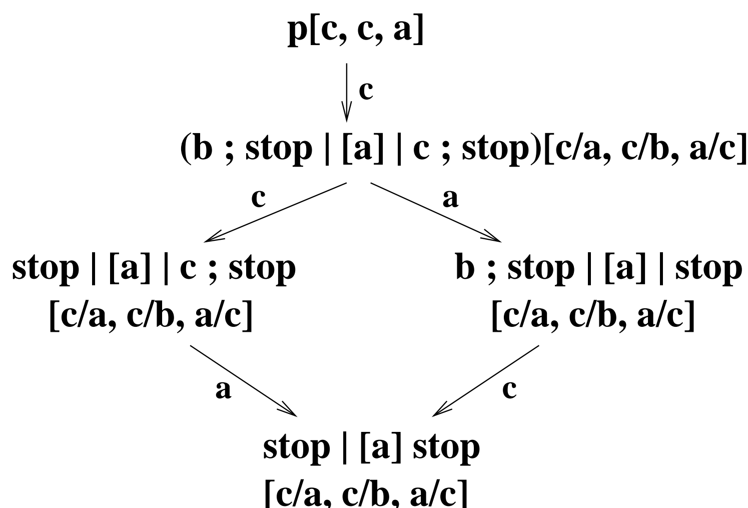
Consider the call $P[c, c, a]$. If the relabeling were done statically, the call would change P into

```
c; c; stop |[c]| c; a; stop
```

and this is the same as the transition system $LTS2$:



The dynamical relabelling in Lotos means that we change action names, but we do not change arcs. We can see in the followin diagram how the relabelling is done step by step.



6.11 Exit and noexit

Usually `exit` or `noexit` is written after the parameter list of a process. `Exit` means that it is possible to connect another process to this process using sequential composition operator. Thus there must be a branch in the process which terminates successfully. `Noexit` says that there is no such branch and that is why sequental composition cannot be used (it is of no use).

Example.

```
process P[a,b]: exit := (a; exit) [] b; stop endproc
process Q[c,d]: noexit := (c; c; d; stop) [] d; c; stop endproc
```

6.12 Examples

6.12.1 Specification of a Producer-Client System

We will write a complete basic Lotos specification. Our system consists of a producer process which sends two messages to a client process using a channel. The channel may lose one or both of the messages. The order of the messages in the channel cannot change. The client is not confused even if messages are lost in the channel, it takes all into account.

```
specification Producer_Consumer[pc1, pc2, cc1, cc2]: exit
behavior
    (Producer[pc1, pc2] ||| Consumer[cc1, cc2])
    ||
```

```
        Channel[pc1, pc2, cc1, cc2]

    where

      process Producer[pc1, pc2]: exit :=
        pc1; pc2; exit
      endproc

      process Consumer[cc1, cc2]: exit :=
        cc1;
          (
            cc2;
              exit
          []
            exit
          )
        []
          cc2;
            exit
        []
          exit
      endproc

      process Channel[pc1, pc2, cc1, cc2]: exit :=
        pc1;
        (
          pc2;
            cc1;
              exit
        []
          cc1;
            pc2;
              exit
        []
          i;
            pc2;
              exit
        )
        >>
        (
          cc2;
            exit
        []
          i;
```

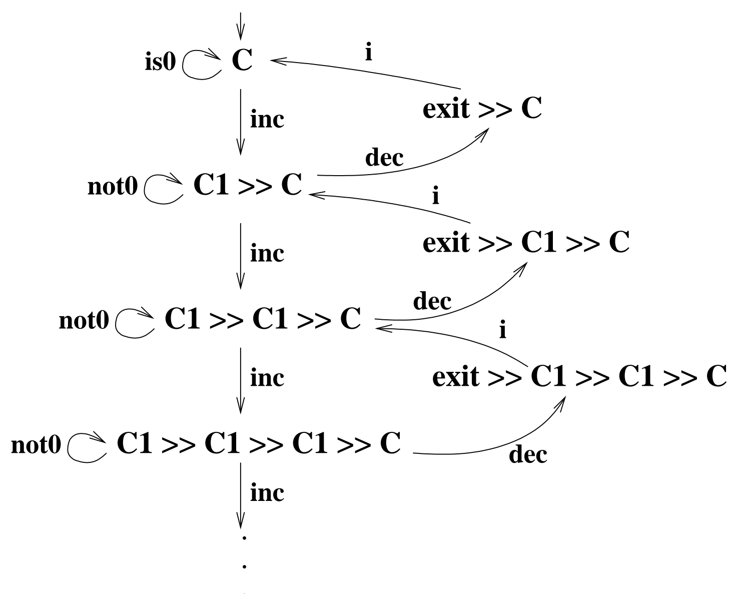
```
      exit
    )
  endproc
endspec
```

6.12.2 Counter

The next example shows how it is possible to define a Lotos specification which generates an infinite transition system.

```
process Counter[is0, not0, inc, dec]: noexit :=
  (inc; C1[not0, inc, dec]
        >>
   Counter[is0, not0, inc, dec])
   []
   (is0; Counter[is0, not0, inc,dec])
where
  process C1[not0, inc, dec]: exit :=
    (dec; exit)
    []
    (inc; C1[not0, inc, dec] >> C1[not0, inc, dec])
    []
    (not0; C1[not0, inc, dec])
  endproc
endproc
```

As a transition system the process is of the following form:



6.12.3 A Client Server System

Let us write the system described 2.2 in basic Lotos. This client-server system is a good example where we will see how the parameter lists can be used in a flexible way. We assume that there are three clients in the system. It is enough to describe two clients, of which one models the starting client and the second the other clients. By changing parameters we can describe the whole system.

```
specification Client_Server_System[t1,t2,t3]: noexit
behavior

 hide cs1, cs2, cs3, sb1, sb2, sb3, bc1, bc2, bc3 in
   Server[cs1, cs2, cs3, sb1, sb2, sb3]
       |[cs1, cs2, cs3, sb1, sb2, sb3]|
   ( (Client1[bc1, cs1, t1, t2] |[bc1]| Buffer[sb1, bc1])
         |[t1, t2]|
     ( (Client[bc2, cs2, t2, t3] |[bc2]| Buffer[sb2, bc2])
         |[t3]|
       (Client[bc3, cs3, t3, t1] |[bc3]| Buffer[sb3, bc3])
     )
   )

where

process Client1[bc,cs,t1, t2]: noexit :=

   cs; (bc; t2; t1; Client1[bc, cs, t1, t2] []
        t2; bc; t1; Client1[bc, cs, t1, t2] )
endproc


process Client[bc, cs, t1, t2]: noexit :=
    t1; cs; (bc; t2; Client[bc, cs, t1, t2] []
             t2; bc; Client[bc, cs, t1,t2] )
endproc


process Server[cs1, cs2, cs3, sb1, sb2, sb3]: noexit :=

   cs1; sb1; Server[cs1, cs2, cs3, sb1, sb2, sb3]
     []
   cs2; sb2; Server[cs1, cs2, cs3, sb1, sb2, sb3]
     []
```

```
   cs3; sb3; Server[cs1, cs2, cs3, sb1, sb2, sb3]
endproc

process Buffer[sb, bc] : noexit :=

   sb; bc; Buffer[sb, bc]
endproc

endspec
```

6.13 CADP

CADP (Caesar/Aldebaran) is a full Lotos software which has been developed mainly in France, but also in Canada ans Spain. Its user interface is graphical and easy to use. The only more difficult task is to make the auxiliary files for data types. But in this course we do not use data types.

The software package is large and it knows many equivalences and representations. There is a separate guide, published on the web page of the course, where you can find all the details necessary to start the program and to use it for simple tasks.

One word of warning: Mistakes in Lotos programs are not always visible to the compiler. For example, if some gate name is missing in the gate list, then this can cause a completely different global state graph compared to the situation where all the gate names are in the list. If the system informs about deadlocks, it is wise to analyse the situation, especially if the graph is smaller or larger than expected.

6.14 AB-protocol

We show a typical scenario of a verification. We verify the version of the AB-protocol, where there are channels and messages `get` and `give`. Let us encode the protocol presented in 4.2. into baseic Lotos:

```
specification AB[get, give]:
  noexit

behavior

hide d0, d1, dd0, dd1, a0, a1, aa0, aa1, st, rt, t in

 ( (Sender[get, d0, d1, aa0, aa1, st,rt,t]
```

```
    |[t,st,rt]|
  Timer[st,t,rt])

    |||

  Receiver[give, dd0, dd1, a0, a1]
 )
    |[d0,d1,dd0,dd1,a0,a1,aa0,aa1]|

  Channel[d0,dd0,d1,dd1,a0,aa0,a1,aa1]

where

process Sender[get,d0,d1,aa0,aa1,st,rt,t]: noexit :=

  get; Transmit[d0, aa0, aa1, st,rt,t] >>
  get; Transmit[d1,aa1, aa0, st,rt,t] >>
  Sender[get,d0,d1,aa0,aa1,st,rt,t]

  where

  process Transmit[d0, aa0, aa1, st,rt,t]: exit :=

    aa0; Transmit[d0, aa0, aa1, st, rt, t]
          []
    d0; st; (t; Transmit[d0, aa0, aa1, st,rt,t]
              []
             aa0; rt; exit
            )
  endproc
endproc


process Timer[st,t,rt] :noexit :=

   st;(t;Timer[st,t,rt] [] rt; Timer[st,t,rt])
endproc


process Receiver[give, dd0, dd1, a0, a1]: noexit :=

   dd0; give; Ack0[give, dd0, dd1, a0, a1]
      []
   dd1; Ack1[give, dd0, dd1, a0, a1]
```

```
   where

         process Ack1[give, dd0, dd1, a0, a1]: noexit :=

            dd1; Ack1[give, dd0, dd1, a0, a1]
               []
            a1; Receiver[give, dd0, dd1, a0, a1]
         endproc



   process Ack0[give, dd0, dd1, a0, a1]: noexit :=

      dd0; Ack0[give, dd0, dd1, a0, a1]
        []
      a0; (dd0; Ack0[give, dd0, dd1, a0, a1]
           []
          dd1; give; Ack1[give, dd0, dd1, a0, a1]
         )


   endproc

endproc



process Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1] : noexit :=

   d0; (i; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
         []
        i; dd0; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
       )
   []
   d1; (i; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
         []
        i; dd1; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
       )
   []
   a0; (i; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
         []
        i; aa0; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
```

```
        )
    []
    a1; (i; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
          []
         i; aa1; Channel[d0,dd0, d1, dd1, a0, aa0, a1, aa1]
        )
endproc

endspec
```

The global state graph consists now of 91 states and 177 transitions. Of the transitions, 156 are τ-transitions. The graph does not contain deadlocks. Now we can compare the global state graph with the service. We do this so that we minimize the global state graph with respect to the bisimulation equivalence. The resulting graph is exactly the same as the service. Thus we have shown that the AB-protocol behaves correctly.

Chapter 7

Model Checking

7.1 Introduction

This last chapter deals with the use of temporal logic in the verification of distributed systems. Temporal logic is appropriate to represent and to verify liveness properties. The other important feature is that with the help of temporal logic it is possible to verify single properties; it is not necessary to model services. Modelling of services is often difficult.

There are many kind of temporal logics. The most fundamental division divides the logics into two classes: *linear time* and *branching time* logics. Traditionally, temporal logics are not based on labelled transition systems but on *Kripke structures*. This enforces us to introduce Kripke structures in this course. These structures resemble transition systems, but transitions are without labels. Instead, states contain information which can be analysed with the help of temporal logic. There is, however, a complete correspondence between transition systems and Kripke structures.

In this chapter, we introduce Kripke structures, their correspondence with transition systems, linear time logic LTL and the basics of branching time logic CTL. There are similar systems which have been developed for transition systems. ACTL is a straightforward modification of CTL for transition systems. One other important formalism is μ-calculus which is also near CTL. We do not deal with these modifications in this course.

7.2 Kripke Structures

Traditionally, temporal logic uses models where processes are represented as graphs in such a way that states contain information. Transitions are without labels. In order to handle temporal logics in a formal way, we must somehow define the data in the states. This is done as follows.

Definition 5 *Kripke structure is a tuple $\mathcal{K} = (S, AP, L, \longrightarrow, S_0)$, where*

- *S is a set of states;*
- *AP is a finite non-empty set of atomic propositions;*
- *$L : S \longrightarrow \mathcal{P}(AP)$ is a function that labels each state with the set of atomic propositions true in that state;*
- *$\longrightarrow \subset S \times S$ is a transition relation that must be total, that is, for every state s there is a state s' such that $(s, s') \in \longrightarrow$; the element $(s, s') \in \longrightarrow$ is called a transition and it is denoted $s \longrightarrow s'$;*
- *S_0 is the set of initial states.*

Example. Consider the following transition system (Dekker's mutual exclusion algorithm)

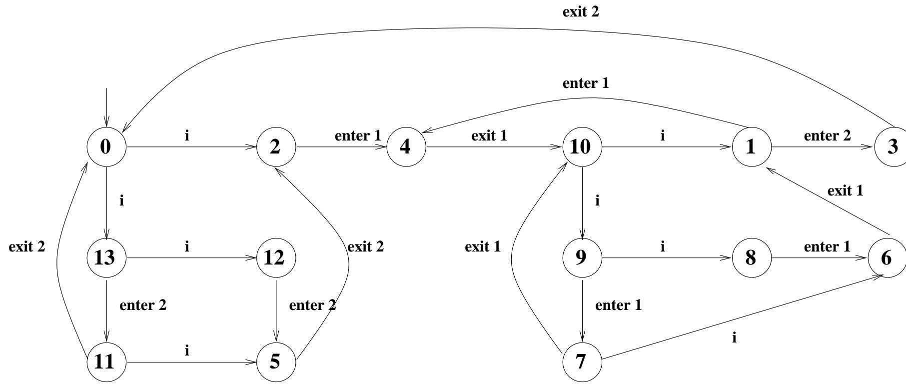


Figure 7.1: Transition system derived from Dekker's mutual exclusion algorithm

Next we show the same as a Kripke structure. As a graph, the Kripke structure is the same as the transition system, see figure 7.2.

We must add the propositions. Let us take two atomic propositions, p_1 and p_2. Proposition p_1 says that process 1 is at the critical area and p_2 says that process 2 is at the critical area. In the original (transition) graph p_1 is true in states 4, 6 and 7. Similarly, p_2 is true in states 3, 5 and 11. In other states p_1 and p_2 are false. These conditions determine the set AP and the function L. The mutual exclusion can now be expressed as a sentence "Both p_1 and p_2 are not true at the same time at any state". $\square$

In the above example the Kripke structure was formed ad hoc. It can be shown that the transformation is always possible.

Proposition 1 *Every labelled transition system can be transformed into an equivalent Kripke structure.*

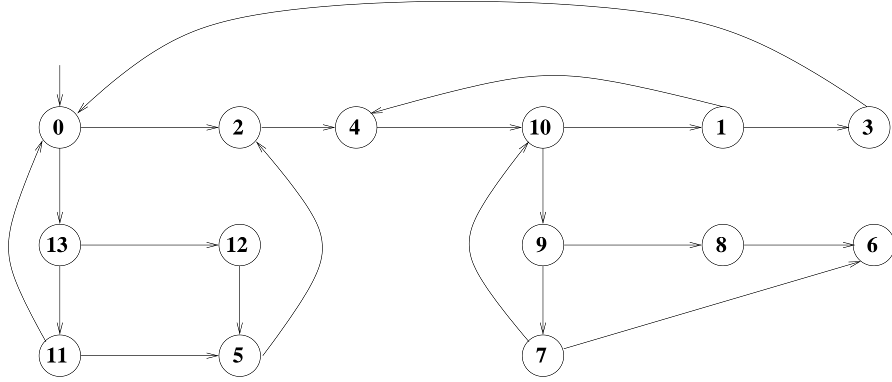


Figure 7.2: States and transitions in the Kripke structure

Proof. Let $LTS = (S, A, T, s_0)$ be a labelled transition system. Let us construct a Kripke structure $\mathcal{K} = (S', AP, L, \longrightarrow, 0_S)$ as follows. The set of states $S' = S \times A$. Between states (s_1, a_1) and (s_2, a_2) there is a transition in $\mathcal{K}$, $(s_1, a_1)\longrightarrow(s_2, a_2)$, if and only if there is a state $s \in S$ such that

$$s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s$$

in LTS. This defines a transition relation $\longrightarrow$ in $\mathcal{K}$. The set of initial states in $\mathcal{K}$ is

$$S_0 = \{(s_0, a_0) \in S' \mid s_0 \xrightarrow{a_0} \text{ in } LTS\}\,.$$

Let η be a variable that can get an action as its value. The propositions are of the form $\eta = a$, where $a \in A$. The interpretation of the proposition is that if $\eta = a$ is true in some state in $\mathcal{K}$, the the system can next perform a in that state. Set now $L : S' \longrightarrow \mathcal{P}(AP)$ by defining $(s, a) \mapsto (\eta = a)$. □

Before we formed the Kripke structure of Decker's algorithm manually ad hoc. If the same is done with the help of the theorem's construction, we will get the structure in figure 7.3 (of which only half is drawn)

The vice versa transformation is possible, too.

Proposition 2 *Every Kripke structure can be transformed into an equivalent transition system.*

Proof. Exercise. Hint: Labels must be sets of propositions. □

7.3 Linear Time Logic LTL

Linear time logic was the first temporal logic that was applied in the verification of distributed computer systems. Its main developers were Manna and Pnueli.

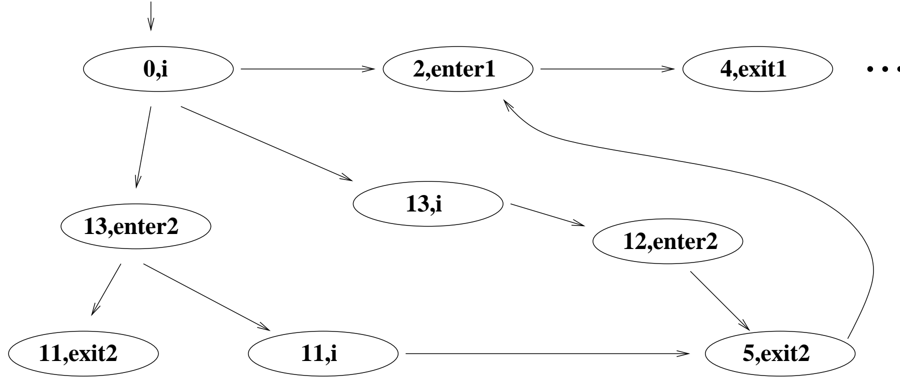


Figure 7.3: Lauseen mukaan johdettu Kripken struktuuri

The starting point of LTL is two operators, X and U. If p is a proposition, then Xp means that next time or in the next state p is true. The formula pUq, on the other hand, claims that p is true until q is true. We define the formulas of LTL formally as follows:

Definition 6 *Let AP the set of atomic propositions. A LTL formula is defined inductively:*

1. *If $\phi \in AP \cup \{\top, \bot\}$, then ϕ is a formula.*
2. *If ϕ and ψ are formulas, then also $(\neg\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$ and $(\phi \equiv \psi)$ are formulas.*
3. *If ϕ and ψ are formulas, then also $X\phi$ and $\phi U \psi$ are.*

The semantics of the formulas are defined formally with the help of Kripke structures and their paths. A path π in a Kripke structure $\mathcal{K}$ is a finite or infinite sequence of states, $s_0 s_1 \cdots s_n$ or $s_0 s_1 \cdots$. State s_0 is always the initial state of a Kripke structure and there is a transition between successive states. A path can be finite only, if it ends at a state where there are no out-going transitions. Sometimes it is demanded that all the states must have out-going transitions. Then only infinite paths are possible. There is not difference between these two agreements in practice. If π is a path $s_0 s_1 \cdots$, then π^k means the part $s_k s_{k+1} \cdots$ of the path.

The truth of a LTL formula is defined first with respect to a path π. If ϕ is a formula and it is true with respect to π, we denote $\pi \models \phi$. In what follows we define the truth of a formula precisely. The truth of the constants $\top$ and $\bot$ is defined in such a way that the former is true in every state and the latter is false in every state.

Definition 7 *The truth of a formula ϕ with respect to a path $\pi = s_0 s_1 \cdots$ is defined as follows:*

- *If $\phi \in A \cup \{\bot, \top\}$ is an atomic proposition or a constant, then $\pi \models \phi$ iff $s_0 \models \phi$ (i.e. $\phi \in L(s_0)$ or ϕ is $\top$).*

- *$\pi \models \phi_1 \vee \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$.*
- *$\pi \models \phi_1 \wedge \phi_2$ iff $\phi \models \phi_1$ and $\pi \models \phi_2$.*
- *$\pi \models \neg\phi$ iff $\pi \not\models \phi$.*
- *$\pi \models X\phi$ iff π^1 exists and $\pi^1 \models \phi$.*
- *$\pi \models \phi_1 U \phi_2$ iff $\pi \models \phi_2$ or there exists $k > 0$ such that π^k is defined, $\pi^k \models \phi_2$ and for all i, $0 \leq i < k$, $\pi^i \models \phi_1$.*

State s satisfies formula ϕ in a Kripke structure $\mathcal{K}$, $\mathcal{K}, s \models \phi$, iff for every path π, starting from s, $\pi \models \phi$.

Formula ϕ is true in $\mathcal{K}$, $\mathcal{K} \models \phi$, iff $\pi \models \phi$ for every path π in $\mathcal{K}$.

Operations F, G and R

Consider the formula $\top U\phi$. Because $\top$ is true in every state, $\top U\phi$ is true on the path π iff $\pi^k \models \phi$ for some $k \geq 0$. Thus $\top U\phi$ is true, if ϕ is true at some point in the future. We use the notation $F\phi$ for this formula.

Formula $G\phi$ says that ϕ is true at every state on the path. It could be expressed with the help of F by writing $\neg F \neg \phi$.

Formula $\phi R \psi$ says that after a finite amount of steps ϕ is true, and before that ψ is true in every state including the first state where ϕ is true.

There are alternative symbols for the operators we have just defined:

- $X = \bigcirc$,
- $G = \Box$,
- $F = \Diamond$,
- $U = \mathcal{U}$.

These symbols are traditional, but the newer symbols are easier to remember. We use both. Next we present some general laws using the traditional symbols:

- Operators $\Box$ and $\Diamond$ are dual:

$$\neg\Box\Phi \equiv \Diamond\neg\Phi.$$

- $\Diamond$ can be written with the help of $\mathcal{U}$ (as we have shown before):

$$\Diamond\Phi \equiv \top\mathcal{U}\Phi.$$

- $\Diamond$ is distributive with respect to $\vee$ and $\Box$ with respect to $\wedge$:

$$\begin{aligned} \Diamond(\Phi \vee \Psi) &\equiv \Diamond\Phi \vee \Diamond\Psi, \\ \Box(\Phi \wedge \Psi) &\equiv \Box\Phi \wedge \Box\Psi. \end{aligned}$$

- In addition,

$$\begin{aligned} \neg \bigcirc \Phi &\equiv \bigcirc \neg \Phi, \\ \neg(\Phi \mathcal{U} \Psi) &\equiv (\neg \Psi \mathcal{U} (\neg \Phi \wedge \neg \Psi)) \vee \Box \neg \Psi. \end{aligned}$$

Typical Formulas

Next we present typical situations where LTL expressions are easy to use:

1. Mutual exclusion:
$$G \neg(\text{critical}_1 \wedge \text{critical}_2)$$
2. At most one request is acknowledged:
$$\bigwedge_{i<j} G \neg(\text{ack}_i \wedge \text{ack}_j)$$
3. Liveness property, according to which my turn is infinitely often:
$$GF\ \text{myturn}$$
4. Liveness property that if succeeding to enter to *try*-area leads finally to the critical area:
$$G(\text{try} \rightarrow F\text{critical})$$
5. After initialization the system stays initialized:
$$FG\ \text{initialized}$$

However, it is not easy to express all the usueful properties in LTL. Take for example the following property of an elevator

Between the times the elevator is asked to a certain floor and it opens its door at that floor, the elevator can pass the floor at most two times:

$$\begin{aligned} G(\quad (\quad & \text{call} \wedge F\text{open}) \rightarrow \\ & ((\neg\text{atfloor} \wedge \neg\text{open})U \\ & (\text{open} \vee ((\text{atfloor} \wedge \text{open})U \\ & (\text{open} \vee ((\neg\text{atfloor} \wedge \neg\text{open})U \\ & (\text{open} \vee ((\text{atfloor} \wedge \neg\text{open})U \\ & (\text{open} \vee ((\neg\text{atfloor} \wedge \text{open})))))))))) \end{aligned}$$

Figure 7.4: Another Kripke strukture, from a mutual exclusion algorithm

7.4 Example

Consider the Kripke structure of another mutual exclusion algorithm in figure 7.4.

Let us examine how various formulas behave with respect to this structure. Formula

$$\Box\neg(C_1 \wedge C_2)$$

is true, because the processes are never at the critical area at the same time. On the other hand, formula

$$\Diamond C_1$$

is not true, because there is an infinite path from the initial state such that User1 has not tried to enter the critical area. But formula

$$\Box(T_1 \Rightarrow \Diamond C_1)$$

is true. It means that if process User1 tries to enter the critical area, it succeeds to enter the area in the future. Thus the algorithm is fair. Formula

$$\Box\Diamond C_1$$

is not true, because without trying a process can never enter the critical area. Furthemore,

$$\Box\Diamond T_1 \Rightarrow \Box\Diamond C_1$$

is true, because at every path where T_1 is performed infinitely many times, the critical area is also entered infinitely many times. Thus the algorithm is fair in a strong sense.