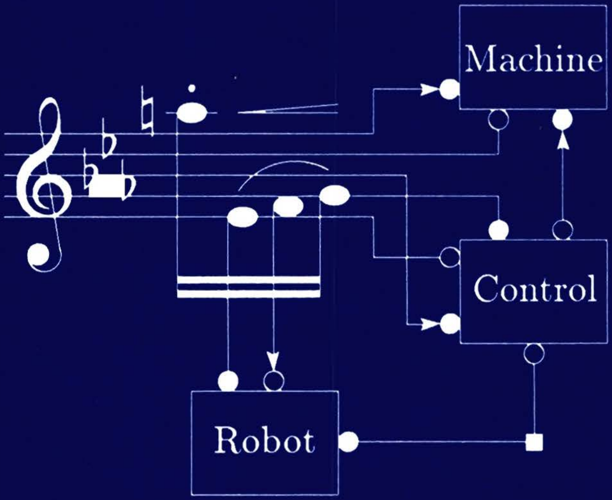
Modelling industrial systems : theory and applications

Modelling Industrial Systems: Theory and Applications



John Koster

Modelling Industrial Systems: Theory and Applications

Modelling Industrial Systems: Theory and Applications

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR AAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF.DR. J.H. VAN LINT, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VAN DEKANEN IN HET OPENBAAR TE VERDEDIGEN OP WOENSDAG 22 DECEMBER 1993 OM 16.00 UUR

DOOR

Gillis Johannes Pieter Koster

GEBOREN TE ZAAMSLAG

Dit proefschrift is goedgekeurd door

de promotoren

prof.dr. M. Rem

en

prof.dr.ir. J.E. Rooda

en de copromotor

dr.ir. W.E.H. Kloosterhuis

The investigations were (partly) supported by the Foundation for Computer Science in the Netherlands (SION) with financial support from the Netherlands Organization for Scientific Research (NWO).

Aan Marie-Anne

Denn das Talent, meine Herren und Damen dort unten, weithin im Parterre, das Talent ist nichts Leichtes, nichts Tändelndes, es ist nicht ohne weiteres ein Können. In der Wurzel ist es Bedürfnis, ein kritisches Wissen um das Ideal, eine Ungenügsamkeit, die sich ihr Können nicht ohne Qual erst schafft und steigert. Und den Größten, den Ungenügsamsten ist ihr Talent die schärfste Geißel ..."

From Thomas Mann's narrative "Schwere Stunde," 1905.

Acknowledgements

First of all, my thanks are due to Martin Rem, who gave me the opportunity to do the research and supervised me throughout the preparation of this thesis.

Many credits are due to Wim Kloosterhuis, who zealously checked the theory and applications, and pointed out various mistakes and omissions. His enthusiasm and expert knowledge of parallel computations have been of paramount importance to me; therefore, my special thanks.

For the implementations made, Johan Lukkien is acknowledged for generously supporting me with his expertise in the field of implementing parallel programs. I very much appreciate his interest and help.

Finally, my grateful thanks go to my wife Marie-Anne, for her never-ceasing encouraging during all stages of the work and, especially, reminding me of much more important things in life than doing research.

Contents

Preface

In 1879 the Hungarian pianist and composer Franz Liszt mentioned in a letter to his friend Baroness von Meyendorff the 9th symphony of Ludwig van Beethoven, the "Choral," as one of the supreme monuments of human culture. The symphony, which consists of four movements, is an immense complex musical structure combining orchestra, chorus, and soloists. It is the result of precise, unwearying work. Of course, conditions like inspiration and creativity had to be satisfied, but above all (like most great works) perseverance was an essential prerequisite. The first ideas of the 9th were already formed when drafting his 7th symphony in 1811. In the end, due to his deafness, Beethoven was no longer able to study at his piano the harmony of the various tunes and accompaniment, though he managed in completing his master-piece. Eventually, on May 7, 1824, the first public performance of the final score of the symphony was given in Vienna.

In order to express his musical thoughts, Beethoven (like most other composers) wrote the score in a staff-notation, a kind of written musical language. The complete score of a symphony consists of such a program for each instrument taking part, whereas each musician in an orchestra gets only that part of the score that applies to the instrument played. A staff-notation tells a musician when, what, and how to play. As a consequence, the concept of time is of paramount importance.

There is a lot of communication going on between musicians in an orchestra. Musicians adjust their playing by listening to and looking at each other. In order to supervise the playing of a somewhat larger orchestra, a conductor is needed. Without supervision a cacophony of sounds will be heard, because each player has a personal interpretation of the music. After some rehearsals a fine orchestra is able to play the piece of music. Nevertheless, the actual performance is still largely influenced by the response from the audience.

Franz Liszt had the profoundest admiration for the symphonies of Beethoven and transcribed them all for the piano. An intimate musical knowledge is needed to succeed in doing the job. Capturing all facets of a symphony into the score of a single instrument is almost impossible, though Liszt managed in doing so. The transcribed pieces of music demand expert musical scholarship of the performing pianist, and as a consequence they are playable only by a happy few.

This thesis addresses the modelling of industrial systems. Although the comparison may be a bit crude, the modelling of industrial systems has some resemblances with the creation of music. An idea is formed and, in order to have a written expression, we need a modelling language. Such a specification of a system enables reasoning about the design. Similar to the creation of music, the specification of a factory will need several revisions of the original scheme until it satisfies the intended idea. A possible, natural modelling approach is formed by separately describing the way in which each mechanism of the system behaves. The collection of all descriptions specifies the whole system. Every now and then, a mechanism has to synchronize with others, for example to exchange information. Like all relevant actions, such an interaction between mechanisms requires an explicit modelling. Another important aspect of the specification language is the incorporation of time. A proper modelling requires a statement about the moment and length of the relevant actions. Hence, a specification states when what action happens.

Often, there is a supervisory control present among the mechanisms in a system. The control acts as a kind of conductor: by means of communication the behaviour of the other mechanisms is supervised. As a result, the behaviour of a mechanism may depend on its environment.

Just like a piece of music, we want to study the harmony in the design of an industrial system, before it is actually built. Repairing a dissonant in a concrete system will be hard and expensive (compare it to a recorded performance). In order to validate whether the specification, the conceptual system, is in accordance with the concrete system required, an executable or prototype is made to simulate the described behaviour in combination to a certain environment (a kind of rehearsal). We might think of building a compiler which translates the specification into a so-called discrete-event model, a single description capturing all aspects of the whole system (like a piano transcription). An advantage of this approach is the ease of obtaining a simulation model, once the correctness of the translation procedure has been shown. We will, however, not do so. Our aim is to devise a scheme in which the distributed nature of the specification is preserved; this results in a distributed discrete-event simulation model (a kind of orchestra approach).

Chapter 1

Introduction

The design of industrial systems, such as machines and factories, requires a *formalism* to specify or model the systems. Suitable formalisms are based upon a mathematical theory, which permits the correctness of the specified system to be validated. Typical correctness criteria are primarily concerned with the functional behaviour, which is the achieved relation between the system's inputs and outputs, and real-time performance requirements. Since more complex systems require an automated validation, we aim for a small distance between the abstract specification and its corresponding implementation in terms of an executable programming language.

We look upon a system as a collection of parallel operating *mechanisms*, also called *processes*. The parallel approach to a system is reflected by its specification, which is formed by the parallel composition of the specifications of the individual mechanisms. The mechanisms cooperate, and together they accomplish the desired functional and real-time performance objectives. The cooperation requires interactions among the mechanisms to exchange the necessary information. This data exchange is achieved by means of *communications* between the participating mechanisms. Unfortunately, the parallel operation of mechanisms often involves complicated performance requirements. The behaviours of the interacting mechanisms should harmonize, otherwise a deadlocking state is likely to occur.

Reasoning about the real-time behaviour of a system requires *time* to be a basic notion of the formalism. A real-time behaviour analysis is based upon the specification, which entails a precise statement about the timing of the activities taking part. For each activity in the system, there is a moment when it begins and a moment when it ends. The interval between these moments corresponds to the time consumption of the activity. Apart from the notion of time, the formalism must be rich in the sense that we can use it to describe a large variety of mechanisms. The expressive power, however, should not make the formalism unmanageable. In fact, a balance between the two is needed: on the one hand the formalism must be kept as simple as possible, and on the other hand it must not exclude on beforehand too many interesting solutions. A basic requirement of the formalism is that we are able to specify mechanisms whose behaviour depends on interactions with its environment. This requirement relates to the presence of deterministic external *choice* to express dependencies on the environment. Apart from deterministic external choice, we need non-deterministic internal choice to describe a certain variety in the behaviour of a mechanism. Internal choice serves, furthermore, as a kind of abstraction technique. The abstraction expresses a certain design freedom in the implementation of the specification. Due to the parallel nature of a system and its proposed partial specification approach, *parallel composition* is needed to define the corresponding system.

Once a system has been specified completely, the question remains whether the specified conceptual system is in accordance with its required concrete equivalent. The specification may contain some parameters that need to be chosen in such a way that the operation of the system is optimized with respect to certain design goals. For complex systems, the value of such parameters will be hard to foretell or to approximate by analytical means. A solution to the validation problem is given by implementing the specified system in terms of an executable programming language. Of course, a prerequisite of such an implementation is that it satisfies its specification. In fact, the implementation serves as a kind of prototype of the system. With the use of the implementation we *simulate* a possible behaviour of the system. On basis of the simulation outcomes we conclude whether the specification is suitable or not. As a consequence, it may be necessary to repeat the procedure a number of times. Typical requirements imposed on the implementation are: it should be easily derived from the specification and take short execution times.

The basic specification formalism we use is the *enabling model* [24], a mathematical theory defining the requirements previously stated. The specification of a mechanism is given by its set of *enabling structures*. The possible behaviours of a mechanism are described in terms of the possible activities which are events or *actions*. These actions are so-called atomic actions, which means that the actions happen instantaneous and have no duration. Behaviours are captured in so-called *schedules*: a schedule gives a mapping of the actions to moments in time. The enabling of an action depends on the schedule of other actions. Once an action has been enabled, it remains enabled until it is performed. An enabling structure distinguishes between internal and external actions. Internal actions happen as soon as they are enabled; external actions can happen only when the environment has enabled the action as well. Consequently, external actions may be delayed by the environment, whereas internal actions cannot.

As a more convenient way of specifying the behaviours of mechanisms we introduce *programs*. In order to define their semantics, programs are related to the enabling model. We define an initial set of primitive programs and, subsequently, more complex programs are built by applying composition rules like sequential and parallel composition, bounded and unbounded recursion, and internal and external choice.

By far the easiest way of obtaining an implementation of the specification is having a compiler do the job [41]. Although such a compiler can be built, we refrain from doing so. Instead, we devise a general scheme to translate a specification into its required implementation format. Unlike other approaches, which choose a single discrete-event implementation [13, 37], we preserve the parallel structure of the specification in its implementation, by which we obtain a *distributed* discrete-event simulation. The target hardware is formed by a parallel computer, a so called MIMD-machine (multiple instructions, multiple data) [14]. The computer is equipped with a number of identical processors that interact by means of communication via hardwired links. In our case, the computer is a reconfigurable Transputer network consisting of 50 Transputers (IMS T800-25) [21, 22, 27]. Typical network configurations are the ring and the torus [36]. An implementation problem of such a distributed approach concerns the mapping of the intended implementation to the available hardware [15]. A related problem is the degree of parallelism. Choosing for maximal parallelism does not necessarily result in the fastest implementation possible: there may be a trade-off in computation and communication times [39, 40]. Apart from the distributed discrete-event approach, we also look at the possibility of using a real-time programming language like Occam [20, 23] or Transputer Pascal [25]. Such a language provides the opportunity of an almost literal translation of a specification into its implementation.

The applicability of the developed theory is tested on some examples of industrial systems that have a close relationship to the area of mechanical engineering. Some applications are concerned with the problem of *factory control* [1, 8]. A rough classification of possible factories is based on the factory layout and results in flow-shop and job-shop factories [38]. A flow-shop factory corresponds to a process-oriented layout: the machines performing the subsequent operations are ordered in a sequence and the product flow between them is fixed, though bypassing a resource is allowed. A functionally oriented layout corresponds to a job-shop factory in which the machines are grouped with respect to their functionality and the product flow is flexible. In case of a job shop, cyclic routes are possible, whereas in a flow shop they are not. Usually, the machines in a job-shop factory are of a more universal kind. Due to the larger degree of freedom, the design of a control architecture for a job-shop factory is more difficult than for a flow-shop factory. Exploiting the potential flexibility of a job shop commonly involves a complex planning strategy.

Overview

The theory in this thesis is described in Chapters 2, 3, 4, and 7. The various applications are presented in Chapters 5, 6, 8, and 9.

In Chapter 2 the enabling model is defined. First, we introduce some basic notions like actions, alphabets, time, and schedules. Afterwards, we add enabling structures to specify a mechanism and parallel composition to describe concurrent operation. The basic model defines the semantics of programs that are introduced next. Programs are added to give a more convenient way of specification.

In Chapter 3 we extend the specification approach that is defined in the previous chapter. Communication between cooperating mechanisms happens via channels, and we distinguish between active and passive communications. In order to describe more complex mechanisms, we allow a kind of parameter passing mechanism in the definition of a mechanism. For asynchronous (buffered) communication, we give an abstraction of the actual description in terms of basic synchronous communication. Since we will often use pictures that give an overview of a system, we explain their meaning.

In Chapter 4 we describe how to transform a specification into an implementation in terms of a programming language. We distinguish between systems with and systems without external choice: the first class being simple, the second one requiring some more effort. We also discuss some optimizations to the basic scheme of implementing a system's specification. In order to give some of the flavour of the approach, an example in which we optimize a buffer size is studied.

In Chapter 5 we describe a case study of a flow-shop factory, whose control is based on the order levels of its buffers: when the number of products in a buffer becomes less than a certain level, it orders new products. We present a general approach and implement a specific instance. It turns out that the implementation is easy to obtain. Various properties of the system are shown, and we also consider some implementation issues.

In Chapter 6 we address a larger example, a lift system. We specify the system and define some control strategies. We show the effect of the strategies in a certain situation and vary the number of parallel operating lifts. We discuss various implementation aspects and study the time-critical behaviour of the lifts.

In Chapter 7 we consider the use of a real-time programming language. Instead of a network of processors, we devise an execution scheme that uses only one processor to execute the implementation in a concurrent fashion. We describe the details and explain how the validity of the simulation outcomes is checked.

In Chapter 8 we use the single processor implementation in the study of a job-shop factory. Just as in the other case studies, we develop the specification and simulate the effect of various control strategies in a specific instance.

In Chapter 9 we give a last case study, the design of a traffic-light system. Starting from a simple specification, we successively add more features to get a satisfactory description. We implement the system and again show the simulation results. The example is used to compare the different implementation techniques.

Notation

Although most of the notations we use are well known, some deviate a bit from the more or less standard notation and are, therefore, shortly explained. Most of the deviations are suggested by [11].

A first remark concerns the use of functions. A function defined in $A \rightarrow B$ has domain A and range B. Besides the traditional notation $f(x)$ for function application, we use $f.x$ as well. Furthermore, functions are also defined by stating the set of relevant pairs. By $f[x := y]$, with x in the domain of f, we denote the function equal to f except that x's image is y:

$$\begin{array}{lllll} f[x:=y](z) = \textbf{if} & z = x & \rightarrow & y \\ [] & z \neq x & \rightarrow & f(z) \\ \textbf{fi} & & & \end{array}$$

For both quantifications and constructions, we use the following general pattern:

$$\langle\text{quantifier} \mid \text{constructor}\rangle \; \langle\text{dummies}\rangle \; : \; \langle\text{predicate}\rangle \; : \; \langle\text{expression}\rangle$$

enclosed by a pair of delimiters. Typical quantifiers are $\forall$, $\exists$, Σ, $\#$, **min**, and **max**, where $\#$ expresses 'the number of,' which are delimited by parentheses. For example, an integer number p, $p > 1$, is called *prime* if it is divisible only by 1 and itself:

$$(\, \# x : 1 < x < p : p \bmod x = 0 \,) = 0$$

Set construction is denoted by $\{\, x : P : E \,\}$, where the constructor is omitted and the delimiters are braces. An example is given by set S consisting of all prime numbers up to and including N, with $N \geq 1$,

$$S = \{\, x : 1 < x \leq N \wedge (\, \forall y : 1 < y < x : x \bmod y \neq 0 \,) : x \,\}$$

The cardinality of a set S is denoted by $|S|$, and the powerset of S, denoted by $\mathcal{P}(S)$, equals:

$$\mathcal{P}(S) = \{\, T : T \subseteq S : T \,\}$$

Apart from the mentioned quantifications and construction, we use others as well; when necessary, their meaning is explained when they are introduced.

Besides sets, another data structure we frequently use is the sequence or list. For set S, S^* denotes the set of all finite-length lists of elements from S. The empty list is denoted by ϵ, and $|L|$ is used to denote the size or length of list L. In order to dissect nonempty lists L, we use $hd.L$ and $tl.L$ to select, respectively, the head and tail of list L. The head of L is the first element, and the tail is the whole list but the head.

A last remark concerns the notation of simple data types. The set of integer numbers is denoted by Int, the set of natural numbers including 0 is denoted by Nat, and the set of boolean values {true, false} is abbreviated to Bool. The interval $\{\, x : a \leq x < b : x \,\}$, where the type of x (integer or real) depends on the context, is denoted by $[a..b)$; the three other variations are $[a..b]$, $(a..b]$, and $(a..b)$, which need no further comment.

Chapter 2

Programs and their semantics

In this chapter we define *programs* as a means to specify the behaviours of mechanisms. The semantics of programs are defined in a suitable formalism, the *enabling model*. The basics of the enabling model have been developed by W.E.H. Kloosterhuis [24]. Originally, the purpose of the formalism was to have a mathematical theory for comparing the performance of a restricted class of concurrent mechanisms. In our case, we use the enabling model only as the basic semantic model underlying the programs. Not all the richness of the model is needed; we adopt a subset and change it to meet our needs. We see to it that the modifications do not affect the correctness of the theory.

The enabling model is an appropriate means to describe the timed-behaviour of mechanisms. A mechanism is described by a set of so-called *enabling structures*. The possible behaviours that a mechanism may show are captured in dependence functions. A dependence function expresses the enabling of actions in terms of actions performed in the past. Internal actions are private to a mechanism and happen as soon as they are enabled. The interactions between mechanisms take place via external, shared actions. An external action may be delayed by the environment and happens when both the mechanism and the environment have enabled the action.

In basic form, enabling structures are still rather inconvenient to be used in the specification of mechanisms. Therefore, we add an extra layer on top of the model in which we specify the possible behaviours of mechanisms. This additional layer is formed by programs that are constructed by composing primitive programs. A mechanism is fully specified by its *program structure* which distinguishes between input and output actions. A simplification in the description of mechanisms that are composed of a number of sub-mechanisms is given by the notion of *systems*.

Throughout this chapter we maintain some naming conventions. In order to have a clear denotation, new conventions are preceded by symbol '$\bullet$'.

In Sections 2.1, 2.2, and 2.3 we introduce some basic notions used in the enabling model: for example, we define actions, alphabets, time, and schedules.

In Section 2.4 we define enabling structures to describe mechanisms. Parallel composition of enabling structures is introduced to describe the concurrent operation of mechanisms.

In Section 2.5 actions of the same kind are captured by their generic representative. We introduce program structures as a more convenient means to specify mechanisms and distinguish between input and output actions.

In Section 2.6 abstract specifications are described; these programs make high level specifications possible, in which the notion of time is preserved.

In Section 2.7 we introduce the concept of systems to specify mechanisms built up from a number of smaller mechanisms.

2.1 Actions and alphabets

When modelling a mechanism, we select the events or *actions* that we want to incorporate in the model. Each individual action is given a name, a symbol. We denote the set of all symbols by Ω. A set of actions, which is a subset of Ω, is called an *alphabet*. In order to have a clear distinction between actions and alphabets, we use the following naming conventions.

- a, b, c, and d denote actions.
- A denotes an alphabet.

Note that we allow the same action to be given different names. So, it is possible that names a and b identify the same event, which yields $a = b$. In other words, names are variables. However, in the examples to come we assume that actions with different names denote distinct events. Furthermore, we want to stress that a name is assigned to each individual action instead of kinds of action. For example, actions a and b may denote a similar activity, with a and b denoting the first and the second occurrence of the activity respectively. As a consequence, similar activities of a mechanism are discriminated by their name.

2.2 Time

Actions occur at a certain moment in time and happen instantaneously, which means that they have no duration. When a proper modelling requires that an activity lasts a certain time interval, it is modelled by two actions: the first action denoting the beginning and the second one denoting the end of the activity. The interval between both actions corresponds to the time required to execute the activity. Just ignoring the actual timing of actions has the advantage of simplifying both the modelling approach and the reasoning about the model afterwards. However, we want to make statements about time or speed like 'the mechanism consumes x time units to produce a product' or pose questions like 'how many products will be produced during interval y?' As a consequence, we have to incorporate the notion of *time* into the modelling approach.

Our notion of time is described by a single, conceptual, global clock, with a dense *time-domain*. We choose the non-negative real numbers as the time-domain [2, 9, 32] and add value ∞ to it. When an action happens at moment ∞, it expresses that the action does not happen at all.

Definition 2.1
The time-domain $\mathcal{T}$ is defined by: [1]

$$\mathcal{T} = [0..\infty]$$

□

Taking the non-negative real numbers as our time-domain is no sine qua non: instead of the real numbers, one could consider the natural numbers for example. For the time being, we stick to the real numbers because they correspond with our notion of time; in Section 2.6 we will use the natural numbers.

The addition of ∞ to the real numbers requires an extension of the arithmetic. Without being complete, a clear apprehension of how to use ∞ in a calculation is obtained by the following computation rules.

$$\begin{cases} x + \infty = \infty & \text{if } x \geq 0 \\ x * \infty = \infty & \text{if } x > 0 \end{cases}$$

In the sequel, we will use the following convention.

- M and N denote moments in time.

2.3 Schedules

The actual assignment of actions to a moment in time is described by a *schedule*, a mapping of actions to the time-domain. By giving such a mapping we define a possible operation of the corresponding mechanism.

Definition 2.2
A function s in $A \to \mathcal{T}$ is called a *schedule* over A. We denote the set of all schedules over alphabet A by $\mathcal{S}.A$.
□

For schedules we will use the following naming convention.

- s and t denote schedules.

[1]The time-domain we have chosen is a subset of the time-domain $(-\infty..\infty]$ used in [24]. As a result, we can express only so-called 'initiated enabling.'

Since schedules are functions, we can compare them by using the partial order '$\leq$' that is defined for functions. By $s \leq t$ we express that s schedules all actions at least as early as t does.

Amongst the possible schedules over a certain alphabet, there is one schedule that describes the nonoccurrence of any activity from the alphabet. This particular schedule is called the *empty schedule.*

Definition 2.3
The *empty schedule* over A, which is denoted by ε_A, schedules all actions on ∞:

$$(\forall a : a \in A : \varepsilon_A.a = \infty)$$

□

Usually, not all schedules represent proper operations of a certain mechanism; a mechanism imposes restrictions upon the scheduling of its actions. Therefore, the description of a mechanism is formed by taking only those schedules that apply to its possible operations.

Definition 2.4
A *schedule set* is a set of schedules over the same alphabet. A non-empty schedule set is called a *process.*
□

Note that a mechanism with alphabet A, that does not allow any action to happen, restricts all possible schedules to one, namely ε_A. A proper description of such a mechanism is given by process $\{\varepsilon_A\}$ and not by the empty set $\emptyset$.

We extend the set of naming conventions introduced thus far by adding the following one for processes.

- P and Q denote processes.

Example 2.5
We consider schedule sets P and Q given by:

$$\begin{aligned} P &= \{\{(a,1),(b,2),(c,\infty)\},\{(a,4),(b,7),(c,8)\}\} \\ Q &= \{\, s : s \in \mathcal{S}(\{a,b\}) \wedge s.a \geq 0 \wedge s.b \geq s.a + 1 : s \,\} \end{aligned}$$

It is obvious that P and Q are processes. In order to form an idea about the programs we will use later on, the program that corresponds to process Q is as follows. (In fact, the correspondence in this and some following examples is still incomplete.)

$$a \,;\, \delta(1) \,;\, b$$

The operation of the mechanism starts at moment 0, on which it wants to do action a. Action b can take place only after the scheduling of a and an additional delay of size 1 which is denoted by $\delta(1)$. The order between the actions is expressed by the sequential composition operator '$;$'.
□

As a general concept we introduce the notion '*alphabet of.*' For arbitrary X, $\mathbf{a}X$ denotes the set of actions in X. For example, the alphabet of action a, which is denoted by $\mathbf{a}a$, equals $\{a\}$; for alphabet A we have $\mathbf{a}A = A$; and for process P, $\mathbf{a}P$ is the domain of its schedules.

A kind of abstraction technique is provided by *projection.* The projection of schedule s on alphabet A hides the actions in s that are not in A. The corresponding result is obtained by omitting from s all pairs with an action not in A.

Definition 2.6
The *projection* of schedule s on alphabet A, which is denoted by $s\lceil A$, hides the actions in s that are not in A:

$$s\lceil A = \{\, a : a \in \mathbf{a}s \cap A : (a, s.a)\,\}$$

For process P and alphabet A, process $P\lceil A$ equals:

$$P\lceil A = \{\, s : s \in P : s\lceil A\,\}$$

The projection of set of processes X on alphabet A is defined by:

$$X\lceil A = \{\, P : P \in X : P\lceil A\,\}$$

□

Note that $\mathbf{a}(s\lceil A) = \mathbf{a}s \cap A$ and, for $a \in \mathbf{a}(s\lceil A)$, we have $(s\lceil A).a = s.a$.

Example 2.7
Schedule s is given as follows:

$$s = \{(a,1),(b,\infty),(c,2)\}$$

Possible abstractions are:

$$s\lceil\{a,b\} = \{(a,1),(b,\infty)\}$$
$$s\lceil\mathbf{a}s = s$$
$$s\lceil\emptyset = s\lceil\{d\} = \varepsilon_\emptyset$$

□

2.4 Enabling structures

In the enabling model, a mechanism is described by a set of *enabling structures.* The enabling of the actions is given in terms of causality relations which express the relation or *dependence* between *cause* and subsequent *effect.* A mechanism is willing to do a certain action, an effect, only when the corresponding cause has taken place. Cause and effect happen at different moments in time, in between there elapses a certain (causal) *delay.* With the intention of keeping the descriptions simple, we assume that the delays are positive.

A formal definition of the dependence relation between cause and effect is given in terms of *similarity.* The similarity of two objects is the maximal moment in time up to which they are indistinguishable.

Definition 2.8
For M and N in $\mathcal{T}$, we define

$$\begin{array}{rl} \mathrm{sim}(M,N) = & \mathbf{if}\ M = N \ \rightarrow\ \infty \\ & [\!]\ \ M \neq N \ \rightarrow\ M \ \mathrm{min}\ N \\ & \mathbf{fi} \end{array}$$

For s and t schedules over the same alphabet, and quantifier **glb** meaning the greatest lower bound, we define

$$\mathrm{sim}(s,t) = (\,\mathbf{glb}\, a : s.a \neq t.a : s.a \ \mathrm{min}\ t.a\,)$$

where the greatest lower bound of an empty range equals ∞.
□

Property 2.9 (without proof)

1. $\mathrm{sim}(s,t) \geq 0$
2. $\mathrm{sim}(s,t) = \infty \equiv s = t$
3. $\mathrm{sim}(s,t) = \mathrm{sim}(t,s)$

□

For a moment, consider a mechanism that can perform the two actions a and b. The mechanism is willing to do effect b only when a delay ξ, $\xi > 0$, has elapsed after the scheduling of cause a. Phrased differently, for a given schedule s, b is enabled at moment $s.a + \xi$. When function ϕ expresses the enabling of b—in this case $\phi.s = s.a + \xi$—we call ϕ the *dependence function* of b. Note that the enabling of b depends on a schedule s: different schedules may give distinct enabling times.

Definition 2.10
For functions $\phi \in \mathcal{S}.A \rightarrow \mathcal{T}$ and A non-empty, the *delay* $\mathbf{d}\phi$ of ϕ is defined by:

$$\mathbf{d}\phi = (\,\mathbf{glb}\, s,t : s,t \in \mathcal{S}.A \wedge s \neq t : \mathrm{sim}(\phi.s, \phi.t) - \mathrm{sim}(s,t)\,)$$

A *dependence function* is such a function ϕ with $\mathbf{d}\phi > 0$.
□

Note that restriction $\mathbf{d}\phi > 0$ resembles a kind of physical limitation of mechanisms: between two successive actions there elapses a certain amount of time.

Example 2.11
We consider the following functions in $\mathcal{S}.A \rightarrow \mathcal{T}$.

$$\begin{array}{l} \phi_1.s = s.a - 1 \\ \phi_2.s = s.a + 0.5 \\ \phi_3.s = \mathbf{if}\ s.a < 1 \ \rightarrow\ \infty \\ \qquad\quad [\!]\ \ s.a \geq 1 \ \rightarrow\ s.a + 2 \\ \qquad\quad \mathbf{fi} \end{array}$$

Function ϕ_1 is not a dependence function because $\mathbf{d}\phi_1 = -1$. On the other hand, both ϕ_2 and ϕ_3 are dependence functions, with $\mathbf{d}\phi_2 = 0.5$ and $\mathbf{d}\phi_3 = 2$.
□

Example 2.12
A mechanism performs actions a, b, c, and d. Action a is enabled at moment 0; actions b and c are enabled after the scheduling of a and a delay of size 5; action d is enabled after the scheduling of both b and c, and a delay of size 2. The dependence functions of these actions are:

$$\begin{aligned}
&\phi_a.s = 0\\
&\phi_b.s = s.a + 5\\
&\phi_c.s = s.a + 5\\
&\phi_d.s = (s.b \text{ max } s.c) + 2
\end{aligned}$$

All these functions are dependence functions, because the corresponding delays are greater than 0: $\mathbf{d}\phi_a = \infty$, $\mathbf{d}\phi_b = \mathbf{d}\phi_c = 5$, and $\mathbf{d}\phi_d = 2$. The program that corresponds to this behaviour is:

$$a \; ; \; \delta(5) \; ; \; b \, , c \; ; \; \delta(2) \; ; \; d$$

where operator ' , ' expresses parallel composition of the actions b and c, which binds stronger than sequential composition.
□

Instead of giving the dependence functions separately, we collect them into one compound function in $\mathcal{S}.A \to \mathcal{S}.A$. For the example above, the compound function is:

$$\begin{aligned}
&\phi.s.a = 0\\
&\phi.s.b = s.a + 5\\
&\phi.s.c = s.a + 5\\
&\phi.s.d = (s.b \text{ max } s.c) + 2
\end{aligned}$$

A possible *behaviour* of a mechanism is described by giving the (compound) dependence function. As a result, a mechanism is partly specified by giving its alphabet and a behaviour description. However, we use two alphabets instead of one: we distinguish between *internal* actions which are strictly private and *external* actions which can be shared with other mechanisms as a means to interact. Moreover, mechanisms can be delayed due to the need to interact.

Definition 2.13
A *structure* E is a triple $\langle A_e, A_i, X \rangle$, with

$$\begin{aligned}
&A_e \cap A_i = \emptyset\\
&X \in \mathcal{S}.A \to \mathcal{S}.A, \text{ where } A = A_e \cup A_i
\end{aligned}$$

For structure E, we define its *external alphabet, internal alphabet*, and *behaviour* by:

$$\mathbf{e}E = A_e \ , \ \mathbf{i}E = A_i \ , \ \mathbf{f}E = X$$

The alphabet of E is $\mathbf{a}E = \mathbf{e}E \cup \mathbf{i}E$. For schedules s, the enabling of the actions $\mathbf{f}E.s$ is abbreviated to $E.s$.

□

Example 2.14
Structure E describes a mechanism with only one behaviour and has external alphabet $\mathbf{e}E = \{a, c\}$ and internal alphabet $\mathbf{i}E = \{b\}$. The behaviour of the mechanism reads as follows:

$$E.s.a = 0 \ , \ E.s.b = s.a + 1 \ , \ E.s.c = s.b + 2$$

Expression $E.s.a$ defines that, given an arbitrary schedule s, the mechanism enables action a at moment 0; $E.s.b$ defines that b is enabled after the scheduling of a and a delay of size 1. Note that this single behaviour allows many possible schedules to happen. For example, a possible schedule and enabling of the actions is:

$$s = \{(a,1),(b,2),(c,5)\} \ , \ E.s = \{(a,0),(b,2),(c,4)\}$$

and another possible combination is:

$$s = \{(a,0),(b,1),(c,3)\} \ , \ E.s = \{(a,0),(b,1),(c,3)\}$$

However, not all schedules are allowed: for example, schedule $s = \{(a,0),(b,1),(c,2)\}$ causes c to be scheduled before it is enabled. Another impossible schedule is given by $s = \{(a,0),(b,2),(c,4)\}$: internal actions happen as soon as they are enabled; only external actions can be delayed by the environment.

□

A structure specifies only one behaviour. In general, a mechanism may show many behaviours instead of one. Therefore, we describe a mechanism by a set of structures over the same internal and external alphabet.[2] The choice between the possible structures is made internally and is non-deterministic. In other words, the internal choice is left unspecified and describes a kind of freedom in the design or the operation of the mechanism. However, not each structure is a legal representation of a possible behaviour: we require a *positive delay* between cause and effect.

Definition 2.15
For structures E, the *delay* $\mathbf{d}E$ is defined by:

$$\mathbf{d}E = (\,\mathbf{glb}\, s,t \,:\, s,t \in \mathcal{S}.A \wedge s \neq t \,:\, \mathrm{sim}(E.s, E.t) - \mathrm{sim}(s,t)\,)$$

An *enabling structure* is a structure E with $\mathbf{d}E > 0$. A *closed* enabling structure is an enabling structure with an empty external alphabet. An *enabling function* is an enabling structure with an empty internal alphabet. The set of all enabling structures over (A_e, A_i) is denoted by $\mathcal{E}(A_e, A_i)$.

□

[2]Another possibility is to adapt Definition 2.13 in such a way that a structure describes a set of behaviours. In order to retain the correspondence with [24], we opt for a set of structures.

We introduce the following naming conventions.

- E and F denote enabling structures.
- C and D denote non-empty sets of enabling structures over the same internal and external alphabet.

Remark 2.16
Enabling structure E is capable of performing only those schedules s in $\mathcal{S}(\mathbf{a}E)$ that satisfy

$$\begin{cases} s.a = E.s.a & \text{if } a \in \mathbf{i}E \\ s.a \geq E.s.a & \text{if } a \in \mathbf{e}E \end{cases}$$

In a sense, this remark is a bit premature and is a consequence of Definition 2.24, in which the possible schedules of enabling structures are defined. Note that it has already been used in Example 2.14. We return to this remark in Property 2.25.
□

Example 2.17
An example of a mechanism with more than one behaviour is described by set $\{E, F\}$, where enabling structures E and F have external alphabet $\{a, c\}$ and internal alphabet $\{b\}$. The behaviours are:

$$\begin{aligned} &E.s.a = 0 \;\;,\;\; E.s.b = s.a + 1 \;\;,\;\; E.s.c = s.b + 1 \\ &F.s.c = 0 \;\;,\;\; F.s.b = s.c + 2 \;\;,\;\; F.s.a = s.b + 1 \end{aligned}$$

Although schedule $\{(a,0),(b,1),(c,2)\}$ is not allowed by F, it is a valid schedule because it is allowed by E. The corresponding program looks like:

$$a \;;\; \delta(1) \;;\; b \;;\; \delta(1) \;;\; c \;\mid\; c \;;\; \delta(2) \;;\; b \;;\; \delta(1) \;;\; a$$

The non-deterministic internal choice is reflected by composition operator '$\mid$', whose binding power is less than that of sequential composition.
□

Example 2.18
The set containing enabling structures E and F, with external alphabet $\{a, c\}$ and internal alphabet $\{b\}$, describes a mechanism that is controlled by the environment. The possible behaviours are as follows:

$$\begin{aligned} E.s.a &= E.s.c = 0 \\ E.s.b &= \textbf{if } s.a \leq s.c \;\rightarrow\; s.a + 1 \\ &\quad\;\, [\!]\;\; s.a > s.c \;\rightarrow\; \infty \\ &\quad\;\, \textbf{fi} \end{aligned}$$

$$\begin{aligned} F.s.a &= F.s.c = 0 \\ F.s.b &= \textbf{if } s.a < s.c \;\rightarrow\; s.a + 1 \\ &\quad\;\, [\!]\;\; s.a \geq s.c \;\rightarrow\; \infty \\ &\quad\;\, \textbf{fi} \end{aligned}$$

The difference between enabling structures E and F involves the choice when actions a and c are scheduled at the same time. Since external actions can be delayed by the environment, the enabling of b is controlled by the environment. Later on we will also introduce a program construct to describe choices that are determined by the environment.

□

As noted before, external actions may be delayed by the environment. For each behaviour there is a schedule that coincides with its enabling, namely the schedule that is not delayed by the environment. In Example 2.14 this delay-free schedule is given by:

$$\{(a,0),(b,1),(c,3)\}$$

Note that there is only one such schedule for each behaviour, because the enabling of an action is fully determined by the scheduling of its cause. We call the schedule that coincides with its enabling the *history* of the enabling structure.

Definition 2.19
For enabling structure E, the *history* of E, which is denoted by $\mathbf{h}E$, is defined as the unique schedule s that satisfies equation

$$s = E.s$$

□

Example 2.20
The histories of the enabling structures in Example 2.17 are given by:

$$\begin{aligned}\mathbf{h}E &= \{(a,0),(b,1),(c,2)\}\\ \mathbf{h}F &= \{(c,0),(b,2),(a,3)\}\end{aligned}$$

For the enabling structures in Example 2.18 we have the following histories:

$$\begin{aligned}\mathbf{h}E &= \{(a,0),(c,0),(b,1)\}\\ \mathbf{h}F &= \{(a,0),(c,0),(b,\infty)\}\end{aligned}$$

□

The definition of the history of an enabling structure is the first step towards the definition of the *process* of an enabling structure. Before we can give that definition, we need to introduce *parallel composition* of enabling structures.

Composing enabling structures in parallel entails the enabling of the actions in the composition. Internal actions are strictly private and cannot be delayed. In case of external actions, an action in the composite is enabled only when the mechanisms that share the action have enabled the action.

Definition 2.21
For schedules s and t, we define schedule s max t over $\mathbf{a}s \cup \mathbf{a}t$ by:

$$\begin{array}{lllll}(s \max t).a = & \textbf{if} & a \in \mathbf{a}s \backslash \mathbf{a}t & \rightarrow & s.a \\ & [] & a \in \mathbf{a}s \cap \mathbf{a}t & \rightarrow & s.a \max t.a \\ & [] & a \in \mathbf{a}t \backslash \mathbf{a}s & \rightarrow & t.a \\ & \textbf{fi} & & & \end{array}$$

□

Not any collection of enabling structures can be composed in parallel: we require the internal actions to be strictly private and the external actions to appear in at most two external alphabets. In the result, the shared actions are made internal.

Definition 2.22
Two enabling structures E and F are *composable* if their internal alphabets are mutually private:

$$\mathbf{i}E \cap \mathbf{a}F = \emptyset \quad \text{and} \quad \mathbf{i}F \cap \mathbf{a}E = \emptyset$$

A set of enabling structures is composable if the enabling structures are pairwise composable and each external action appears in at most two external alphabets.

For composable enabling structures E and F we define the *parallel composition*, which is denoted by $E \bowtie F$, in

$$\mathcal{E}(\mathbf{e}E \div \mathbf{e}F, \mathbf{i}E \cup \mathbf{i}F \cup (\mathbf{e}E \cap \mathbf{e}F))$$

by:[3]

$$(E \bowtie F).s = E.(s\lceil \mathbf{a}E) \max F.(s\lceil \mathbf{a}F)$$

Similarly, sets of enabling structures C and D are composable if their internal alphabets are mutually private. We define their parallel composition as follows:

$$C \bowtie D = \{E, F : E \in C \wedge F \in D : E \bowtie F\}$$

□

When we apply parallel composition, composability of the enabling structures is implicitly assumed. Note that the parallel composition of two enabling structures yields an enabling structure again. For any composable E and F we have:

$$\mathbf{d}(E \bowtie F) \geq \mathbf{d}E \min \mathbf{d}F > 0$$

Furthermore, for enabling structures with a delay of at least ξ, $\xi > 0$, this property generalizes to infinite sets of enabling structures: the result will always have a delay of at least ξ.

[3]In fact, this form of parallel composition '$\bowtie$' is a combination of the parallel composition '$\|$' and masking used in [24].

Example 2.23
Consider enabling structures E in $\mathcal{E}(\{b\},\{a\})$ and F in $\mathcal{E}(\{b\},\{c\})$:

$$\begin{array}{ll} E.s.a = 0 \ , & E.s.b = s.a + 2 \\ F.s.b \ = 3 \ , & F.s.c = s.b + 1 \end{array}$$

The corresponding programs are:

$$\begin{array}{ll} E: & a \ ; \ \delta(2) \ ; \ b \\ F: & \delta(3) \ ; \ b \ ; \ \delta(1) \ ; \ c \end{array}$$

For the parallel composition of these behaviours we easily compute the enabling of the internal actions a and c:

$$(E \bowtie F).s.a = 0 \ , \ (E \bowtie F).s.c = s.b + 1$$

With respect to action b we find:

$$(E \bowtie F).s.b = (s.a + 2) \max 3$$

In the parallel composition, all actions are internal. As a consequence, the possible schedules are restricted to just one, namely the history of $E \bowtie F$ which equals:

$$\{(a,0),(b,3),(c,4)\}$$

□

For enabling structure E, *process* $\mathbf{p}E$ consists of all schedules it is willing to participate in. With respect to the environment, which consists of a set of enabling structures, we assume that there is a closed connection: the external alphabet of the composition is empty.

Definition 2.24
The *process* $\mathbf{p}E$ of enabling structure E is the set of all schedules it may engage in when placed in a closed connection to any composable environment. With construction $(\bowtie F : R : F)$ denoting the parallel composition of all F in range R, $\mathbf{p}E$ is defined by:

$$\mathbf{p}E = \{X : \ : \mathbf{h}(\bowtie F : F \in X \cup \{E\} : F)\lceil \mathbf{a}E\}$$

For set of enabling structures C, the *process set* $\mathbf{p}C$ is defined by:

$$\mathbf{p}C = \{E : E \in C : \mathbf{p}E\}$$

□

Since $\mathbf{p}E$ contains schedule $\mathbf{h}E$, $\mathbf{p}E$ is non-empty and is indeed a process.

Property 2.25 (without proof)

$$\mathbf{p}E = \{\, s : s \in \mathcal{S}(\mathbf{a}E) \wedge s \geq E.s \wedge s \lceil \mathbf{i}E = (E.s) \lceil \mathbf{i}E : s \,\}$$

As a result of this property, Remark 2.16 holds.
□

Example 2.26
A mechanism is described by enabling structures E and F in $\mathcal{E}(\{a,c\},\{b\})$. The behaviours of E and F are:

$$\begin{array}{lll} E.s.a = 0 \ , & E.s.b = s.a + 1 \ , & E.s.c = s.b + 1 \\ F.s.b \ = 1 \ , & F.s.a = s.b + 1 \ , & F.s.c = s.a + 2 \end{array}$$

In program terms, the specification of this mechanism is:

$$a \ ; \ \delta(1) \ ; \ b \ ; \ \delta(1) \ ; \ c \ | \ \delta(1) \ ; \ b \ ; \ \delta(1) \ ; \ a \ ; \ \delta(2) \ ; \ c$$

Process set $\mathbf{p}(\{E,F\})$ is given by:

$$\begin{array}{l} \{ \ \{ s : s \in \mathcal{S}(\{a,b,c\}) \wedge s.a \geq 0 \wedge s.b = s.a + 1 \wedge s.c \geq s.a + 2 : s \}, \\ \ \ \{ s : s \in \mathcal{S}(\{a,b,c\}) \wedge s.a \geq 2 \wedge s.b = 1 \wedge s.c \geq s.a + 2 : s \} \ \} \end{array}$$

□

Example 2.27
Consider enabling functions E and E', with external alphabet $\{a,b\}$, which are defined by:

$$\begin{array}{ll} E.s.a \ = 0 \ , & E.s.b \ = s.a + 1 \\ E'.s.a = 0 \ , & E'.s.b = 0 \end{array}$$

The mechanism described by the set consisting of E and E' has process set

$$\begin{array}{rl} \mathbf{p}(\{E,E'\}) = \{ & \{ s : s \in \mathcal{S}(\{a,b\}) \wedge s.a \geq 0 \wedge s.b \geq s.a + 1 : s \}, \\ & \{ s : s \in \mathcal{S}(\{a,b\}) \wedge s.a \geq 0 \wedge s.b \geq 0 : s \} \ \} \end{array}$$

Suppose the environment is described by enabling function F:

$$F.s.b = 0 \ , \ F.s.a = s.b + 1$$

The process set of the parallel composition equals:

$$\{ \ \{\varepsilon_{\{a,b\}}\}, \{\{(b,0),(a,1)\}\} \ \}$$

Note that the result can deadlock, which is apparent from the process set.
□

As a means to compare mechanisms and their behaviours, we introduce the notion of *equivalence*. Mechanisms are called equivalent when they can be substituted for each other.

Definition 2.28
Enabling structures E and F are *equivalent* if the internal and external alphabets are the same

$$\mathbf{i}E = \mathbf{i}F \ , \ \mathbf{e}E = \mathbf{e}F$$

and if they exhibit the same external process:

$$\mathbf{p}E\lceil\mathbf{e}E = \mathbf{p}F\lceil\mathbf{e}F$$

Similarly, sets of enabling structures C and D are equivalent if the internal and external alphabets are the same and if they have the same external process set:

$$\mathbf{p}C\lceil\mathbf{e}C = \mathbf{p}D\lceil\mathbf{e}D$$

□

Example 2.29
Enabling structure E with external alphabet $\mathbf{e}E = \{a, c\}$, internal alphabet $\mathbf{i}E = \{b\}$, and behaviour

$$E.s.a = 0 \ , \ E.s.b = s.a + 1 \ , \ E.s.c = s.b + 2$$

is equivalent to enabling structure F with external alphabet $\mathbf{e}F = \{a, c\}$, internal alphabet $\mathbf{i}F = \{b\}$, and behaviour

$$F.s.a = 0 \ , \ F.s.b = s.a + 2 \ , \ F.s.c = s.b + 1$$

□

2.5 Generic actions and programs

Describing a mechanism by a set of enabling structures can be very tedious, especially when the mechanism shows an everlasting behaviour. Therefore, we introduce *programs* as a more convenient means to specify all potential behaviours of mechanisms. A program is built up from a number of smaller programs. An action is an example of a simple program. Since in the enabling model actions happen at most once, repetitive behaviours require infinitely many symbols to denote the activity. However, in programs we denote actions of the same kind by the same representative, their *generic action*. As a consequence, a behaviour induced by a program is described in terms of *occurrences* of the generic actions. The use of such generic actions is captured in the following conventions.

- The set containing all occurrences of a generic action a is a subset of Ω, and is given by $\{ i : i \geq 0 : a_i \}$.
- An alphabet contains either all occurrences of a generic action or none.
- An enabling structure respects the order of the occurrences of the same generic action. For enabling structure E, with $a \in \mathbf{a}E$, this implies that every $s \in \mathbf{p}E$ satisfies $s.a_{i+1} > s.a_i$ or $s.a_{i+1} = \infty$.
- We abbreviate the notation of alphabets by mentioning the generic actions only.

We build programs from a set of elementary programs and use composition rules to join them into larger programs. With program S we associate alphabet $\mathbf{a}S$ and *execution-tree set* $\mathbf{t}S$. The alphabet contains the generic actions that occur in the program; a program has external actions only. The execution trees, which can be infinite, describe the possible behaviours of the mechanism, as each execution tree is related to an enabling function. An execution tree is a binary tree $\langle\, u\ x\ v\,\rangle$, with left subtree u, root x, and right subtree v. The possible roots are: elementary programs and composition operators. To reduce the number of angular brackets, x is used as short for $\langle\, \langle\,\rangle\ x\ \langle\,\rangle\,\rangle$. The construction of programs and corresponding execution trees, or shortly trees, is given at the same time.

The elementary programs are called *primitive programs*. The primitive programs capture the basic notions of enabling structures, which are actions and delays. For example, generic action a is a primitive program, and a delay of size M, $M \in \mathcal{T}$, is described by primitive program $\delta(M)$.

Definition 2.30
The *primitive programs* S and their corresponding alphabet $\mathbf{a}S$ and tree set $\mathbf{t}S$ are defined as follows.

- For $M \in \mathcal{T}$, delay $\delta(M)$ is a program, with $\mathbf{a}\delta(M) = \emptyset$ and $\mathbf{t}\delta(M) = \{\delta(M)\}$.
- Action a is a program, with $\mathbf{a}a = \{a\}$ and $\mathbf{t}a = \{a\}$.

□

Two special delays are denoted by ε and $\perp$: program ε, called the *empty program*, equals $\delta(0)$; and program $\perp$, called the *stop program*, equals $\delta(\infty)$.

With respect to programs and trees we use the following conventions.

- S, T, and U denote programs.
- u, v, and w denote execution trees.

The execution order of compound programs is expressed by the composition operators that are used. As a first composition operator we introduce *sequential composition*, which is used to combine programs into a sequence. The sequential composition of programs S and T, which is denoted by

$$S\,;T$$

expresses that T happens after S. When T may not happen immediately after S but only after a delay of size M, $M > 0$, program $\delta(M)$ is added in between, which yields $S\,;\,\delta(M)\,;\,T$. Since in the enabling model we require a positive delay between cause and effect, the sequential composition of actions a and b is always of the form $a\,;\,\delta(M)\,;\,b$. The elements in the tree set of the sequential composition of programs S and T have: root '$;$', a left subtree from $\mathbf{t}S$, and a right subtree from $\mathbf{t}T$.

Definition 2.31
For programs S and T, *sequential composition* $S\,;T$ is a program, with

$$\mathbf{a}(S\,;T) = \mathbf{a}S \cup \mathbf{a}T$$
$$\mathbf{t}(S\,;T) = \{\,u,v : u \in \mathbf{t}S \wedge v \in \mathbf{t}T : \langle\, u\,;\,v\,\rangle\,\}$$

□

The class of programs that can be constructed by using sequential composition only is too restrictive, because a sequence imposes a total order on the execution of the programs. In order to express that some programs are unordered and may happen simultaneously, we introduce *parallel composition* of programs. Parallel composition of programs S and T is denoted by:

$$S\,,T$$

Not all combinations of S and T are allowed: we require that $\mathbf{a}S \cap \mathbf{a}T = \emptyset$. For actions a, b, and c, program $a\,;\,\delta(M)\,;\,(b\,,\,c)$ expresses that after cause a and a delay of size M both b and c can happen simultaneously. Furthermore, the program terminates only when b and c have taken place. The elements in the tree set of a parallel composition are obtained in a way similar to sequential composition.

Definition 2.32
For programs S and T, with $\mathbf{a}S \cap \mathbf{a}T = \emptyset$, *parallel composition* $S\,,T$ is a program, with

$$\mathbf{a}(S\,,T) = \mathbf{a}S \cup \mathbf{a}T$$
$$\mathbf{t}(S\,,T) = \{\,u,v : u \in \mathbf{t}S \wedge v \in \mathbf{t}T : \langle\, u\,,\,v\,\rangle\,\}$$

□

With the composition rules introduced thus far we can describe mechanisms that exhibit just one behaviour. A set of programs suffices to describe multiple behaviours. Instead of using a set, we add *choice* as a composition operator to denote the possibility of multiple behaviours. The non-deterministic *internal choice* between programs S and T is denoted by:

$$S\,|\,T$$

The tree set of $S\,|\,T$ is formed by the union of tree sets $\mathbf{t}S$ and $\mathbf{t}T$. Internal choice cannot be influenced by the environment: the choice reflects freedom in the implementation or operation of the mechanism.

Definition 2.33
For programs S and T, non-deterministic *internal choice* $S \mid T$ is a program, with

$$\begin{aligned}\mathbf{a}(S \mid T) &= \mathbf{a}S \cup \mathbf{a}T\\ \mathbf{t}(S \mid T) &= \mathbf{t}S \cup \mathbf{t}T\end{aligned}$$

□

Besides internal choice we add a choice construct that is controllable by the environment, namely deterministic *external choice.* Action a that *guards* program S causes S to be delayed until the environment enables action a. The selected alternative of an external choice is determined by the enabling of the actions in the guards. For program S guarded by action a, and program T guarded by action b, we denote the external choice by:

$$(\bar{a} \rightarrow S \;\lhd\; \bar{b} \rightarrow T)$$

The choice results in the execution of either S or T, though S is preferred when the environment enables both actions at the same time. Since the act of passing a guard causes the execution of its alternative, the alternative must start with a delay.

Definition 2.34
For programs S and T, and actions a and b, deterministic *external choice* denoted by $(\bar{a} \rightarrow S \;\lhd\; \bar{b} \rightarrow T)$ is a program, with

$$\begin{aligned}\mathbf{a}(\bar{a} \rightarrow S \;\lhd\; \bar{b} \rightarrow T) &= \{a, b\} \cup \mathbf{a}S \cup \mathbf{a}T\\ \mathbf{t}(\bar{a} \rightarrow S \;\lhd\; \bar{b} \rightarrow T) &= \{\, u, v : u \in \mathbf{t}S \wedge v \in \mathbf{t}T : \langle\, u \;(\bar{a} \lhd \bar{b})\; v \,\rangle \,\}\end{aligned}$$

□

Usually, we do not want to denote a preference for the left alternative in an external choice. For that reason, we introduce a non-deterministic variant which uses symbol ' $[]$ ' instead of symbol ' $\lhd$ ' and means:

$$(\bar{a} \rightarrow S \;[]\; \bar{b} \rightarrow T) = (\bar{a} \rightarrow S \;\lhd\; \bar{b} \rightarrow T) \mid (\bar{b} \rightarrow T \;\lhd\; \bar{a} \rightarrow S)$$

Writing out the program of a mechanism with a repetitive behaviour by using only the introduced composition operators is hard to do; therefore, a kind of abbreviation rule is needed. We use *recursion* for this purpose. A mechanism that performs an *unbounded* number of times program T is recursively specified by program

$$S = T \;;\; S$$

Unfolding the recursion yields:

$$S = T \;;\; T \;;\; T \;;\; \ldots$$

Since we do not want to give a meaning to programs like $S = S$, we require that T is a *productive program*, which means that each tree of T contains at least one action. The alphabet of S equals the alphabet of T, and the tree set is recursively defined as the smallest set X of trees with root ' ; ', left subtrees from $\mathbf{t}T$, and right subtrees from X. In general, recursion looks like $S = U(S)$, where $U(S)$ expresses the dependence of U on S—in the above example, $U(S) = T \;;\; S$. The tree set is the smallest set that satisfies the corresponding recursive tree-set equation.

Example 2.35
For productive program T, S recursively defined by *unbounded recursion* $S = T \; ; \; S$ is a program, with alphabet

$$\mathbf{a}S = \mathbf{a}T$$

and tree set $\mathbf{t}S$ which is defined as the smallest solution of

$$X : \quad X = \{\, u, v : u \in \mathbf{t}T \wedge v \in X : \langle\, u \; ; \; v \,\rangle \,\}$$

□

Apart from unbounded recursion, we will often use a form of *bounded recursion* that is controlled by the environment. For programs T and U, and actions a and b, such a recursive definition looks like:

$$S = (\bar{a} \rightarrow T \, ; \, S \;\; \lhd \;\; \bar{b} \rightarrow U)$$

The corresponding mechanism repeats program T as long as the environment is willing to do an a. When the b-guarded alternative is selected, the recursion finishes.

Example 2.36
For programs T and U, S defined by *bounded recursion* $S = (\bar{a} \rightarrow T \, ; \, S \;\; \lhd \;\; \bar{b} \rightarrow U)$ is a program, with alphabet

$$\mathbf{a}S = \{a, b\} \cup \mathbf{a}T \cup \mathbf{a}U$$

and tree set $\mathbf{t}S$ which is defined as the smallest solution of

$$X : \quad X = \{\, u, v, w : u \in \mathbf{t}T \wedge v \in X \wedge w \in \mathbf{t}U : \langle\, \langle\, u \; ; \; v \,\rangle \; (\bar{a} \lhd \bar{b}) \; w \,\rangle \,\}$$

□

The set of all programs is denoted by Π. In order to limit the necessity to use brackets in our programs, we define that sequence

$$, \quad ; \quad | \quad \lhd \quad []$$

denotes the symbols in decreasing order of binding power. As a rule of thumb, more ink relates to a smaller binding power. Furthermore, the operators associate to the right.

Example 2.37
Program S is given by:

$$S = (a \; ; \; \delta(1)) \, , \, \delta(5) \; ; \; (b \,|\, \varepsilon) \; ; \; \delta(2) \; ; \; c$$

The alphabet of S contains the actions that appear in the program:

$$\mathbf{a}S = \{a, b, c\}$$

Due to the internal choice, the corresponding tree set consists of two elements:

$$\begin{aligned}\mathbf{t}S = \{ & \langle \langle \langle a \,;\, \delta(1) \rangle \,,\, \delta(5) \rangle \,;\, \langle b \,;\, \langle \delta(2) \,;\, c \rangle \rangle \rangle \\ & \langle \langle \langle a \,;\, \delta(1) \rangle \,,\, \delta(5) \rangle \,;\, \langle \varepsilon \,;\, \langle \delta(2) \,;\, c \rangle \rangle \rangle \,\}\end{aligned}$$

The program expresses that action a_0 is enabled at moment 0, and the choice between b_0 and ε is postponed until moment $(s.a_0 + 1)$ max 5. The enabling functions E and F, which correspond to the trees, have external alphabet $\{a, b, c\}$ and behaviours

$$\begin{array}{llll} E.s.a_0 = 0 \;, & E.s.b_0 = (s.a_0 + 1) \text{ max } 5 \;, & E.s.c_0 = s.b_0 + 2 \\ F.s.a_0 = 0 \;, & F.s.b_0 = \infty & , \; F.s.c_0 = ((s.a_0 + 1) \text{ max } 5) + 2 \end{array}$$

The enabling of the actions with an index greater than 0 is defined at ∞:

$$(\, \forall i, x : i > 0 \wedge x \in \{a, b, c\} : E.s.x_i = F.s.x_i = \infty \,)$$

Note that the incompleteness mentioned in Example 2.5 concerns the difference between generic actions used in programs and occurrences used in enabling structures and schedules. Furthermore, the programs we consider have no internal actions.
□

Example 2.38
Consider programs S and T:

$$\begin{aligned} T &= a \,;\, \delta(2) \,;\;\, b \,,\, c \,;\, \delta(1) \\ S &= T \,;\, S \end{aligned}$$

The characteristics of program T are:

$$\begin{aligned} \mathbf{a}T &= \{a, b, c\} \\ \mathbf{t}T &= \{\langle a \,;\, \langle \delta(2) \,;\, \langle \langle b \,,\, c \rangle \,;\, \delta(1) \rangle \rangle \rangle\} \end{aligned}$$

The alphabet and tree set of program S are:

$$\begin{aligned} \mathbf{a}S &= \mathbf{a}T \\ \mathbf{t}S &= \{\langle u \,;\, \langle u \,;\, \langle u \,;\, \ldots \rangle \rangle \rangle\} \end{aligned}$$

where u denotes the tree in $\mathbf{t}T$. Program S is properly defined because T is productive. The alphabet of S is equal to the alphabet of T, and its tree set consists of a single tree which is related to enabling function E, with external alphabet $\{a, b, c\}$ and behaviour

$$\begin{aligned} E.s.a_i &= \textbf{if } i = 0 \;\rightarrow\; 0 \\ &\quad\;\; [\!]\;\; i > 0 \;\rightarrow\; (s.b_{i-1} \text{ max } s.c_{i-1}) + 1 \\ &\quad\;\; \textbf{fi} \\ E.s.b_i &= s.a_i + 2 \\ E.s.c_i &= s.a_i + 2 \end{aligned}$$

□

A program describes the enabling of actions relative to the moment of initiation 0. Between successive actions we require the presence of a positive delay. For example, program $a\,;\,\delta(1)\,;\,b$ maps the enabling of a to moment 0 and the enabling of b to moment $s.a + 1$. Before we can give a formal definition of how a tree relates to a certain enabling function, we need to introduce some new concepts to define the meaning of programs that contain external-choice constructs.

We distinguish between generic *input* and generic *output* actions. Consequently, an action is either input or output. An output action may never be used in a guard, whereas an input action may be. Both alphabets are added to the program that specifies the mechanism, thereby forming a triple, a *program structure.* In a collection of program structures we require shared actions to be of different types.

Definition 2.39
A *program structure* V is a triple $\langle A_O, A_I, X\rangle$, with

$$A_O \cap A_I = \emptyset$$
$$X \in \Pi, \text{ and } \mathbf{a}X = A_O \cup A_I$$

and all guards in X contain only actions from A_I.
For program structure V, the *output alphabet*, *input alphabet*, and *program* are:

$$V_O = A_O \quad , \quad V_I = A_I \quad , \quad V_\pi = X$$

The alphabet of V is $\mathbf{a}V = V_O \cup V_I$.
□

We use the following convention for program structures.

- V and W denote program structures.

Selecting the alternative of an external choice depends on the readiness of the environment to do the actions in the guards. In order to determine this readiness, we associate with each generic action a an extra generic action $\bar{a}$ of the same type, which is called the *probe action* of a. In the notation of alphabets, the probes are omitted. The interpretation of a probe action depends on whether the action is input or output. For output action a, probe $\bar{a}$ is a request to do the action and, for input action b, probe $\bar{b}$ denotes whether the request to do b has been received. As a result, output request $\bar{a}$ is enabled at the same time as action a. Probe action $\bar{b}$ is enabled when it is possibly needed, namely a certain positive delay after the scheduling of the previous occurrence of action b. In order to describe this enabling, we assume a minimum delay ξ, $\xi > 0$, between successive actions. Consequently, enabling functions of programs have a delay of at least ξ. The definition of enabling function E_u, which corresponds to tree u, is now given in two steps: first we define the enabling of probe actions, and second we define the enabling of the other actions.

Definition 2.40
For program structure V, tree u in $\mathbf{t}V_\pi$ is related to enabling function E_u, with

$$\mathbf{e}E_u = \mathbf{a}V \quad , \quad \mathbf{i}E_u = \emptyset$$

The enabling of output probe $\bar{a} \in V_O$ is defined by

$$E_u.s.\bar{a}_i = E_u.s.a_i$$

and for input probe $\bar{b} \in V_I$ we define

$$\begin{aligned} E_u.s.\bar{b}_0 &= 0 \\ E_u.s.\bar{b}_{i+1} &= s.b_i + \xi \end{aligned}$$

□

In order to complete the relation between a tree and its enabling function, we have to define the enabling of the remaining actions. Before the definition is given, we comment on the use of recursion. For example, consider programs S and T:

$$\begin{aligned} S &= a \; ; \; \delta(\xi) \; ; \; S \\ T &= S \; ; \; \delta(\xi) \; ; \; a \end{aligned}$$

An anomaly happens in program T: after all a actions have been used up in program S, program T wants to do another a. Although the extra a action does not take place because of the minimum delay between successive actions, it must be avoided. Furthermore, when an infinite number of actions can happen within a limited amount of time, we want the extra a action to be impossible too; this situation may arise in case of abstract specifications which are discussed in the next section. We choose for the following solution: after the execution of an infinite number of actions, the remaining actions are neglected and do not happen. As a result, the execution of an infinite number of actions does not terminate. In the examples described in following chapters, this anomaly does not occur.

Definition 2.41
For program structure V, with tree $u \in \mathbf{t}V_\pi$, the enabling of the actions in u, say $E_u.s = t_u$, is recursively defined over the structure of u by function *en*. Function application of *en* looks like $en(u, s, (\tau, I, t))$, where τ denotes the current time; function I, the *index function*, gives for each action the number of previous occurrences in the tree; and schedule t denotes the enabling of the actions in the inspected part of the tree. The outcome of $en(u, s, (\tau, I, t))$ is a new triple (τ, I, t). An initial call of function *en* satisfies:

$$\tau = 0 \;\;,\;\; (\forall a : \; : I.a = 0\,) \;\;,\;\; t = \varepsilon_{\mathbf{a}V}$$

When $en(u, s, (0, I_0, \varepsilon_{\mathbf{a}V})) = (\tau_u, I_u, t_u)$, where I_0 satisfies the initial requirement, the enabling of the actions in u is $E_u.s = t_u$. The definition of *en* is as follows, with I such that $(\forall a : \; : I.a < \infty\,)$:

- $en(\delta(M), s, (\tau, I, t)) = (\tau + M, I, t)$
- $en(a, s, (\tau, I, t)) = (s.a_{I.a}, I[a := I.a + 1], t[a_{I.a} := \tau])$

- If $en(u, s, (\tau, I, t)) = (\tau_u, I_u, t_u)$ then

  ```
  $en(\langle u \; ; \; v \rangle, s, (\tau, I, t)) =$
    if  $(\exists a : \; : I_u.a = \infty) \rightarrow (\tau_u, I_u, t_u)$
    ▯  $\neg(\exists a : \; : I_u.a = \infty) \rightarrow en(v, s, (\tau_u, I_u, t_u))$
    fi
  ```

  Note that the unfolding terminates when an infinite number of actions has happened in u.

- If $en(u, s, (\tau, I, t)) = (\tau_u, I_u, t_u)$ and $en(v, s, (\tau, I, t)) = (\tau_v, I_v, t_v)$ and u and v have no common actions, $\mathbf{a}u \cap \mathbf{a}v = \emptyset$, then

  $$en(\langle u \; , \; v \rangle, s, (\tau, I, t)) = (\tau_u \max \tau_v, I_u + I_v, t_u \min t_v)$$

  Since the parallel composition finishes only when both parts have been completed, the maximum of the time values is taken. Due to the absence of common actions in both parts of the composition, the number of occurrences in the composite is obtained by taking the sum of I_u and I_v, and the resulting enabling is got by taking the minimum of t_u and t_v.

- For input actions a and b,

  ```
  $en(\langle u \; (\bar{a} \lhd \bar{b}) \; v \rangle, s, (\tau, I, t)) =$
    if $(\tau \max s.\bar{a}_{I.a}) \leq (\tau \max s.\bar{b}_{I.b}) \rightarrow en(u, s, (\tau \max s.\bar{a}_{I.a}, I, t))$
    ▯  $(\tau \max s.\bar{a}_{I.a}) > (\tau \max s.\bar{b}_{I.b}) \rightarrow en(v, s, (\tau \max s.\bar{b}_{I.b}, I, t))$
    fi
  ```

  As we want to ignore possible orderings in the past, the current time τ is taken into account when comparing the probe actions of the guards. Note that probe action $\bar{a}_{I.a}$ belongs to the next a action.

For a set of trees X, the set consisting of the corresponding enabling functions is denoted as $EF(X)$.
□

The generalization to external-choice constructs with more than two alternatives, or just one, is straightforward. Note that $\bar{a} \rightarrow S$ is short for $(\bar{a} \rightarrow S \;\lhd\; \bar{a} \rightarrow S)$. Guards composed of a conjunction or disjunction of probe actions cause no problems either: in case of a conjunction, the maximum of the scheduling times of the probes is taken, whereas in case of a disjunction the minimum applies. An extension that we illustrate with an example consists of using guards that are pairs formed by an action and a time value. These guards may be passed when the current time is at least the last schedule of the action plus the time value. Consider program S, with programs T and U, and input actions a and b:

$$S = (\bar{a} \rightarrow T \;\lhd\; (b, 5) \rightarrow U)$$

For $v \in \mathbf{t}T$ and $w \in \mathbf{t}U$, tree $u \in \mathbf{t}S$ is of the form:

$$u = \langle v \; (\bar{a} \lhd (b, 5)) \; w \rangle$$

We define the enabling of the actions in u by:

```
$en(u, s, (\tau, I, t)) =$
  if $(\tau \max s.\bar{a}_{I.a}) \le (\tau \max (s.b_{I.b-1} + 5))$
     $\rightarrow en(v, s, (\tau \max s.\bar{a}_{I.a}, I, t))$
  [] $(\tau \max s.\bar{a}_{I.a}) > (\tau \max (s.b_{I.b-1} + 5))$
     $\rightarrow en(w, s, (\tau \max (s.b_{I.b-1} + 5), I, t))$
  fi
```

Similarly, we may add time values to probe actions. When adding positive values, we may drop the requirement that each alternative starts with a delay. Observe the different indexing scheme of actions and probes: an action refers to the last occurrence, whereas a probe belongs to a next occurrence.

Example 2.42
An example of a program with (non-deterministic) external choice reads:

```
$S = ( \bar{a} \rightarrow \delta(1)\ ;\ a$
    $[]\ \bar{b} \rightarrow \delta(1)\ ;\ b$
    $)$
```

where a and b are input actions. The alphabet and the tree set of program S are:

$$\mathbf{a}S = \{a, b\}$$
$$\mathbf{t}S = \{\langle\langle \delta(1)\ ;\ a\rangle\ (\bar{a} \lhd \bar{b})\ \langle \delta(1)\ ;\ b\rangle\rangle\ ,\ \langle\langle \delta(1)\ ;\ b\rangle\ (\bar{b} \lhd \bar{a})\ \langle \delta(1)\ ;\ a\rangle\rangle\}$$

The trees are related to enabling functions E and F in $\mathcal{E}(\{a, b\}, \emptyset)$:

```
$E.s.\bar{a}_0 = E.s.\bar{b}_0 = 0$
$E.s.a_0 =$ if $s.\bar{a}_0 \le s.\bar{b}_0 \rightarrow s.\bar{a}_0 + 1$
           [] $s.\bar{a}_0 > s.\bar{b}_0 \rightarrow \infty$
           fi
$E.s.b_0 =$ if $s.\bar{a}_0 \le s.\bar{b}_0 \rightarrow \infty$
           [] $s.\bar{a}_0 > s.\bar{b}_0 \rightarrow s.\bar{b}_0 + 1$
           fi

$F.s.\bar{a}_0 = F.s.\bar{b}_0 = 0$
$F.s.a_0 =$ if $s.\bar{a}_0 < s.\bar{b}_0 \rightarrow s.\bar{a}_0 + 1$
           [] $s.\bar{a}_0 \ge s.\bar{b}_0 \rightarrow \infty$
           fi
$F.s.b_0 =$ if $s.\bar{a}_0 < s.\bar{b}_0 \rightarrow \infty$
           [] $s.\bar{a}_0 \ge s.\bar{b}_0 \rightarrow s.\bar{b}_0 + 1$
           fi
```

The non-probe actions with index greater than 0 are enabled at moment ∞.
□

Now that we have defined the meanings of programs, we formulate some properties. The equality used expresses that the corresponding sets of enabling functions are the same.

Property 2.43 (without proof)
Internal choice is symmetric, idempotent, and associative:

1. $S \mid T = T \mid S$
2. $S \mid S = S$
3. $S \mid (T \mid U) = (S \mid T) \mid U$

Consequently, non-deterministic external choice is symmetric:

4. $(\bar{a} \rightarrow S \ [\!] \ \bar{b} \rightarrow T) = (\bar{b} \rightarrow T \ [\!] \ \bar{a} \rightarrow S)$

Parallel composition is symmetric and associative:

5. $S\,,T = T\,,S$
6. $S\,,(T\,,U) = (S\,,T)\,,U$

Sequential composition is associative:

7. $S\,;(T\,;U) = (S\,;T)\,;U$

$\square$

Property 2.44 (without proof)
Sequential composition distributes through internal and external choice:

1. $S\,;(T \mid U) = (S\,;T \mid S\,;U)$
2. $(T \mid U)\,;S = (T\,;S \mid U\,;S)$
3. $(\bar{a} \rightarrow S \ \lhd \ \bar{b} \rightarrow T)\,;U = (\bar{a} \rightarrow S\,;U \ \lhd \ \bar{b} \rightarrow T\,;U)$

Consequently,

4. $(\bar{a} \rightarrow S \ [\!] \ \bar{b} \rightarrow T)\,;U = (\bar{a} \rightarrow S\,;U \ [\!] \ \bar{b} \rightarrow T\,;U)$

$\square$

Property 2.45 (without proof)
For programs S we have:

1. $\varepsilon\,;S \;=\; S\,;\varepsilon \;=\; S$
2. $\varepsilon\,,S \;=\; S\,,\varepsilon \;=\; S$

With respect to delays we find:

4. $\delta(M)\,;\,\delta(N) \;=\; \delta(M+N)$
5. $\delta(M)\,,\,\delta(N) \;=\; \delta(M \text{ max } N)$

□

When using program structures for specifying mechanisms, care should be taken with respect to processes. The environment is now a set of program structures and, hence, only enabling functions that correspond to programs are relevant. For example, consider the possible schedules of an output action and its probe action: in case of programs the probe action cannot happen after the action, whereas in case of enabling functions it can.

Definition 2.46
The process set $\mathbf{p}V$ of program structure V is defined by:

$$\begin{aligned}\mathbf{p}V = \{\,E : E \in EF(\mathbf{t}V)\\ :\{\,X : \;:\mathbf{h}(\bowtie F : F \in EF(X) \cup \{E\} : F\,)\lceil \mathbf{a}V\,\}\,\}\end{aligned}$$

where X is a set of trees which yields a closed connection.
□

The history of the enabling function of a tree is still a valid schedule. To make this clear, we consider the maximally cooperative program (environment) of alphabet A, which is denoted by $MCP(A)$. For actions a, the maximally cooperative program is:

$$MCP(a) = a\;;\;\delta(\xi)\;;\;MCP(a)$$

The maximally cooperative program of alphabet $A = \{a1, a2, \ldots, an\}$, $n \geq 1$, consists of the parallel composition of the maximally cooperative programs of the actions:

$$MCP(A) = MCP(a1)\;,\;MCP(a2)\;,\;\ldots\;,\;MCP(an)$$

We conclude this section by stating the equivalence of program structures.

Definition 2.47
Program structures V and W are called equivalent if the input and the output alphabets are the same

$$V_I = W_I\;\;,\;\;V_O = W_O$$

and if they have the same process set:

$$\mathbf{p}V = \mathbf{p}W$$

□

Determining the equivalence of program structures on basis of the above definition requires all possible environments to be taken into account. In general, verifying the equality of the process sets is therefore difficult. Resolving this problem is beyond the scope of this thesis.

2.6 Abstract specifications

In the enabling model, we assume a positive delay between cause and effect. However, on a higher level of abstraction, we often want to express only the order of actions and refrain from the delay in between. This requires the possibility to schedule ordered actions at the same moment. For example, consider the (still invalid) programs S and T, with a, b, and c being output actions in S and input actions in T.

$$S = (a \;;\; b) \;,\; c$$
$$T = a \;,\; (\bar{b} \rightarrow b \;\triangleleft\; \bar{c} \rightarrow c)$$

In the parallel composition of S and T, actions a, b, and c are internal and, hence, happen as soon as they are enabled. To give a meaning to these programs, we could think of adding implicit delays to enforce the delay required between successive actions. For example, this is achieved by adding minimum delay ξ to each sequential composition and each guard in the programs. When doing so, we find the following scheduling: actions a_0, $\bar{a}_0$, and $\bar{c}_0$ are scheduled at moment 0, and action $\bar{b}_0$ is scheduled at moment ξ. Since action $\bar{b}_0$ is scheduled after action $\bar{c}_0$, alternative c is chosen in program T. However, on a more abstract level, this is undesired: we want alternative b to be possible too, because b and c are not causally related. In fact, we need a *zero delay* after the a action has been scheduled.

A first step in the formal description of zero delays consists of choosing a different time-domain, namely the two-dimensional domain Nat $\times$ Nat extended with value ∞ [19]. Just like before, actions that are scheduled on moment ∞ do not happen at all. These time-moments are called *micro-moments*. For micro-moment (x, y), we refer to the x-value as its *macro-moment* and we call x's domain the *macro-domain*. The idea is that real time steps happen in the macro-domain. Furthermore, on the same macro-moment, any number of actions can happen in sequence. Because of minimum delay ξ, $\xi > 0$, this property cannot be realized in case of a one-dimensional time-domain.

Definition 2.48
The time-domain $\mathcal{T}'$ is defined by:

$$\mathcal{T}' = (\text{Nat} \times \text{Nat}) \cup \{\infty\}$$

In order to compare (micro-) moments (x_0, y_0) and (x_1, y_1) in $\mathcal{T}'$, we use the lexicografical ordering:

$$\begin{aligned}(x_0, y_0) < (x_1, y_1) \equiv\;& x_0 < x_1 \\ & \vee\, (x_0 = x_1 \wedge y_0 < y_1)\end{aligned}$$

□

Note that when the macro-domain is too course grained, a division of the domain by a sufficiently large natural number makes it apt; in other words, the macro-domain is scaled. The reason for choosing the natural numbers as our macro-domain is that the consequences on the enabling model are small; it is straightforwardly generalized, whereas in case of reals it is not. In the new time-domain, the smallest possible time step is $(0,1)$ and, hence, the minimum delay is $\xi = (0,1)$. To illustrate the arithmetic of this time-domain, we give the addition of micro-moments:

$$\begin{cases} (x_0, y_0) + (x_1, y_1) = (x_0 + x_1, y_0 + y_1) \\ (x_0, y_0) + \infty \qquad\quad = \infty \end{cases}$$

Even though we can use zero delays between successive actions, the choice in the example above still persist in choosing the c alternative. In order to enforce the possibility of the b alternative, we could think of evaluating the guards only on basis of the macro-moments of the schedule of the actions. This idea turns out to be wrong because the order between successive actions may be lost. For example, consider the parallel composition of programs $(a \ ; \ b)$ and $(\bar{b} \rightarrow b \ \lhd \ \bar{a} \rightarrow a)$. When a and b are scheduled at the same macro-moment and the choice in T takes only the macro-moment into account, the b alternative is chosen; this is definitely not what we want.

We opt for the following solution: before each action, there happens an arbitrary positive number of implicit ξs. As a result, the order between actions is preserved but the delay is not fixed. Note that the possible delays all have the same macro-moment. A small change concerns the schedule of the probe actions as defined in Definition 2.40. Actions cannot happen before moment ξ because each action is preceded by at least one ξ. To retain the maximally cooperative environment, input probes with index 0 are enabled at moment ξ instead of 0:

- $E_u.s.\bar{b}_0 = \xi$

For the example given at the beginning of this section, possible trees of the programs with explicit delays are as follows:

$$\begin{array}{l} S : \langle\, \langle\, \langle\, \delta(\xi) \ ; \ a \,\rangle \ ; \ \langle\, \delta(\xi) \ ; \ b \,\rangle\,\rangle \ , \ \langle\, \delta(3\xi) \ ; \ c \,\rangle\,\rangle \\ T : \langle\, \langle\, \delta(\xi) \ ; \ a \,\rangle \ , \ \langle\, \langle\, \delta(\xi) \ ; \ b \,\rangle \ (\bar{b} \lhd \bar{c}) \ \langle\, \delta(\xi) \ ; \ c \,\rangle\,\rangle\,\rangle \end{array}$$

Composing their enabling functions in parallel yields the following (partial) history:

$$\{(a_0, \xi), (\bar{a}_0, \xi), (b_0, 3\xi), (\bar{b}_0, 2\xi), (c_0, \infty), (\bar{c}_0, 3\xi)\}$$

As a result of the arbitrary delay before each action, a single action yields infinitely many trees. Therefore, we have to re-define the tree set of actions as given in Definition 2.30: the tree set of action a in case of abstract specifications is

- $\mathsf{t}a = \{\, x : x > 0 : \langle\, \delta(x * \xi) \ ; \ a \,\rangle\,\}$

In the remainder, we stick to abstract specifications. To enhance readability, we write for example $s.a_0 + 3$ instead of $s.a_0 + (3,0)$ and $\delta(5)$ instead of $\delta((5,0))$. Furthermore, minimum delay ξ does not occur in programs; only natural numbers are allowed.

Example 2.49
We consider programs S and T, with a and b being output actions in S and input actions in T.

$$\begin{aligned}&S = a \; ; \; b\\ &T = (\bar{a} \rightarrow a \; [] \; \bar{b} \rightarrow b)\end{aligned}$$

In their parallel composition, the a alternative is always chosen in program T.
□

Example 2.50
Programs S and T are:

$$\begin{aligned}&S = (a \; ; \; \delta(5) \; ; \; c) \; , \; (b \; ; \; \delta(5) \; ; \; d)\\ &T = a \, , \, b \; ; \; (\bar{c} \rightarrow c \; [] \; \bar{d} \rightarrow d)\end{aligned}$$

where T consists of input actions only. In the parallel composition of S and T, both alternatives in the choice in T are possible. To denote the similarity between operators '$[]$' and '$\lhd$' in case of abstract specifications, we also consider program T':

$$T' = a \, , \, b \; ; \; (\bar{c} \rightarrow c \; \lhd \; \bar{d} \rightarrow d)$$

Composing S and T' in parallel reveals the possibility of the c and d alternative, too. However, the operators are discriminated when the probe actions have happened in the past. For example, in the parallel composition of S and T'',

$$T'' = a \, , \, b \; ; \; \delta(6) \; ; \; (\bar{c} \rightarrow c \; \lhd \; \bar{d} \rightarrow d)$$

only the c alternative is possible.
□

The use of guards consisting of an action and a time value needs some further explanation. To illustrate the point of concern, consider programs S and T:

$$\begin{aligned}&S = a \; ; \; b\\ &T = a \; , \; b \; ; \; (\; (a,1) \rightarrow c\\ &\qquad\qquad\quad [] \; (b,1) \rightarrow d\\ &\qquad\qquad\quad)\end{aligned}$$

In their parallel composition, the a and b actions happen at the same macro-moment. As a result, we want both alternatives c and d to be possible. The guards are, however, evaluated on basis of the micro-moments which yields only the c alternative because a happens before b. In fact, an evaluation on basis of macro-moments is required. We reconsider, therefore, the meaning of trees of the form

$$u = \langle \, v \; (\bar{a} \lhd (b,5)) \; w \, \rangle$$

With truncation $\lfloor (x,y) \rfloor = (x,0)$, we define the enabling of the actions in u by:

```
- $en(u, s, (\tau, I, t)) =$
    if $(\tau \max s.\bar{a}_{I.a}) \leq (\tau \max \lfloor(s.b_{I.b-1} + 5)\rfloor)$
        $\rightarrow en(v, s, (\tau \max s.\bar{a}_{I.a}, I, t))$
    $[\!]$ $(\tau \max s.\bar{a}_{I.a}) > (\tau \max \lfloor(s.b_{I.b-1} + 5)\rfloor)$
        $\rightarrow en(w, s, (\tau \max \lfloor(s.b_{I.b-1} + 5)\rfloor, I, t))$
    fi
```

Note that the alternative following guard $(b, 5)$ is preferred when $s.\bar{a}_{I.a}$ and $s.b_{I.b-1}+5$ have the same macro-moment after τ. Because of the truncation, the associated time values must be positive: otherwise, causality-related problems may arise.

A last remark concerns the equivalence of program structures. Defining equivalence of program structures as the equality of the process sets is rather strong: we want to consider differences in the macro-domain only. When defining equivalence as the equality of the process sets restricted to the macro-domain, it is possible that program structures are equivalent, while they can be discriminated by an environment; this violates the idea of equivalence. We take the following: program structures V and W are equivalent when they cannot be distinguished in the macro-domain by an environment; in other words, the sets of histories restricted to the macro-domain are the same in any environment. Of course, the alphabets of V and W have to be the same. With $\lfloor s \rfloor$ denoting schedule s restricted to the macro-domain, and

$$h(X, V) = \{ E : E \in EF(\mathbf{t}V) : \lfloor \mathrm{h}(\bowtie F : F \in EF(X) \cup \{E\} : F) \rfloor \}$$

equivalence of program structures V and W in case of abstract specifications is:

$$(\forall X : : h(X, V) = h(X, W))$$

In fact, the usefulness of this notion of equivalence is limited because of its complexity.

2.7 Systems

The operation of a mechanism that is built up from a number of smaller mechanisms is defined by the parallel composition of the corresponding sets of enabling structures. Instead of writing down this parallel composition, we want to give only the set of programs that describe the sub-mechanisms. Therefore, we introduce the concept of *systems*. Not all mechanisms can be joined into systems: we require that the related sets of enabling functions are composable, and that the pairwise intersection of input and output alphabets is empty. Furthermore, an interaction requires an action to be in both an input and an output alphabet.

Definition 2.51
A *system* X is a set of program structures, which satisfies: each action appears in at most two program structures

$$\begin{aligned} (\forall a, V &: V \in X \wedge a \in \mathbf{a}V \\ &: \neg(\exists W, W' : W, W' \in X \backslash \{V\} \wedge W \neq W' : a \in \mathbf{a}W \cap \mathbf{a}W')) \end{aligned}$$

and the input and output alphabets are disjoint

$$(\forall V, W : V, W \in X \wedge V \neq W : V_I \cap W_I = \emptyset \wedge V_O \cap W_O = \emptyset\,)$$

The external alphabet of system X consists of all input actions that do not appear in any output alphabet and vice versa. The internal alphabet contains all actions that appear in both an input and an output alphabet.

$$\begin{aligned} \mathbf{e}X &= (\cup V : V \in X : V_I\,) \div (\cup V : V \in X : V_O\,) \\ \mathbf{i}X &= (\cup V : V \in X : V_I\,) \cap (\cup V : V \in X : V_O\,) \end{aligned}$$

A *closed system* is a system with an empty external alphabet.
□

The parallel composition of two systems is easy to define. In fact, the composition requires that the union of both sets yields a system again.

Definition 2.52
For systems X and Y, with $X \cap Y = \emptyset$ and $X \cup Y$ a system, the parallel composition of X and Y, denoted by $X \bowtie Y$, is defined as the union of X and Y:

$$X \bowtie Y = X \cup Y$$

□

Example 2.53
We consider system X that consists of program structures V and W, with $V_O = \{a, b\}$, $V_I = W_O = \emptyset$, and $W_I = \{b, c\}$. The programs V_π and W_π are:

$$\begin{aligned} V_\pi &= a\ ;\ \delta(3)\ ;\ b\ ;\ V_\pi \\ W_\pi &= b\ ;\ \delta(5)\ ;\ c\ ;\ W_\pi \end{aligned}$$

In the greedy enabling structure E of system X, each action is preceded by a single implicit ξ. The behaviour of E in $\mathcal{E}(\{a, c\}, \{b\})$ is:

```
E.s.a_i = if $i = 0 \rightarrow \xi$
          [] $i > 0 \rightarrow s.b_{i-1} + \xi$
          fi
E.s.b_i = if $i = 0 \rightarrow s.a_i + 3 + \xi$
          [] $i > 0 \rightarrow (s.c_{i-1} + \xi)$ max $(s.a_i + 3 + \xi)$
          fi
E.s.c_i = $s.b_i + 5 + \xi$
```

The enabling of the probe actions needs no further comment.
□

Chapter 3

Communication and values

The specification of a mechanism as presented in the previous chapter describes the behaviours of a mechanism in terms of input and output actions. Later on, we use a mechanism's specification as the starting-point for obtaining its implementation. In order to narrow the gap between specification and subsequent implementation, we extend our programs with so-called *communications.*

The operational meaning of programs becomes clearer when the specification formalism contains constructs that correspond to physical concepts or implementation methods. Programs that are easily understood simplify reasoning about them and, consequently, modelling becomes easier. As a way to clarify their meaning we introduce, for describing the interactions between mechanisms, the concept of '*communication via channels.*' A communication involves the exchange of data between the two participating mechanisms; one partner performs an input and the other an output communication. Sometimes, a communication is coupled with an explicit request to communicate. In order to abstract from these communication requests, we distinguish between *active* and *passive communications.* Detecting pending communications is accomplished by the *probe function*, which is defined on passive channels only. A process can be suspended due to a communication. When this is undesired, buffered or *asynchronous communication* is used. Another abstraction is needed for the possible data transferred in a communication; therefore we use *variables.*

Adding variables to our programs requires a more powerful program notation. For that reason, we introduce the notion of a *parameterized program* which describes a whole range of programs. An *instantiation* of a parameterized program selects a certain program from the range. To specify similarly behaving mechanisms, we add the concept of a *generic program.*

In the remainder, we (ab)use the word 'process' to refer to the mechanism that corresponds to a specification; for example, for program structures V, we mean by process V the specified mechanism.

In Section 3.1 we introduce communication via channels, and distinguish between active and passive communications. Due to the incorporation of data, we have to extend the definition of programs.

In Section 3.2 we introduce parameterized programs, which are functions that range over the set of programs, and generic programs to avoid duplicates in the specification.

In Section 3.3 asynchronous communication is described in terms of the available synchronous communication.

In Section 3.4 we introduce pictures as a means to give the connection diagram of the processes in a system.

Finally, in both Sections 3.5 and 3.6 the specification approach is illustrated by giving an example.

3.1 Communication

All interesting mechanisms interact with their environments, because interactions are necessary to exchange information between the participating mechanisms. We look upon an interaction between mechanisms as a *communication* which happens via a certain medium, a so-called *channel* [16, 17]. The communications are accomplished by sending and receiving messages. In the underlying model, *communication actions* are described by pairs $\langle a, m\rangle$, where a denotes the channel name and m the message. Each channel can convey messages of a certain type only; this is called the channel type. For channel a, we refer to its channel type by $type(a)$. Communication actions are also occurrences of a generic action: for generic action $\langle a, m\rangle$, the ith occurrence is denoted by $\langle a, m\rangle_i$. Consequently, communication actions have a probe action; for example the probe that belongs to action $\langle a, m\rangle$ is $\overline{\langle a, m\rangle}$.[1]

The use of channels is limited by some regulations. We confine ourselves to systems with channels that appear in at most two alphabets, thereby restricting the linking of channels to point-to-point connections. A kind of physical limitation is reflected by the requirement that, along the same channel, at most one communication can happen at the same time. Therefore, we strengthen the precondition in Definition 2.32 to the absence of common channels in the participating programs.

A consequence of the incorporation of data is the inevitable external choice between the possible input actions that belong to a channel. An illustration of this is given in the next example.

Example 3.1
An inverter, with channels a and b of type $\{0, 1\}$, alternates between communications along a and b, and starts with a. The communications via a are input actions, whereas the communications via b are output actions. Hence, the possible inputs are $\langle a, 0\rangle$ and $\langle a, 1\rangle$, and the possible outputs are $\langle b, 0\rangle$ and $\langle b, 1\rangle$. The inverted value of an a action is communicated in the subsequent b action. The behaviours of the mechanism are described by program *Inverter*:

[1]Intuitively, one would expect a single probe action for each channel. This approach is possible but requires more effort.

$$\begin{array}{ll} \mathit{Inverter} = & (\ \overline{\langle a,0\rangle} \rightarrow \langle a,0\rangle\ ;\ \langle b,1\rangle \\ & [\!]\ \overline{\langle a,1\rangle} \rightarrow \langle a,1\rangle\ ;\ \langle b,0\rangle \\ &) \\ & ;\ \mathit{Inverter} \end{array}$$

□

In communication, the information transport is directed from *sender* to *receiver*, where the sender executes an output communication and the receiver performs the matching input communication. The allocation of sender and receiver remains the same for all communications via a channel. As a result, a channel is directed and used by a connected process for either input or output communications. We say that a channel is an input channel for the receiving process and an output channel for sending process.

Instead of explicitly stating the input and output alphabet that belong to a program, we add a notation to discriminate input and output communications: exclamation mark '!' is added to output actions, whereas question mark '?' is added to input actions. Consequently, the input and output alphabet can be determined from the program, as we will do. Furthermore, we assume that the type of each channel is given in the context. The output of the value of expression e on channel a is denoted by $a!e$ and, for a of type $\{0,1\}$, means:

$$\begin{array}{ll} a!e = & \textbf{if}\ e=0\ \rightarrow\ \langle a,0\rangle \\ & [\!]\ \ e=1\ \rightarrow\ \langle a,1\rangle \\ & \textbf{fi} \end{array}$$

The input of a message from a channel entails the external choice between all possible actions that correspond to the input channel. Writing down this choice can be rather elaborate or virtually impossible. For that reason, we abstract from the actual values by using a *variable* to store the received message; for example, we write $a?x$ where x is a variable of type $\mathit{type}(a)$. Program $a?x$ is interpreted as the reception of a value via channel a and the assignment of that value to variable x. For a of type $\{0,1\}$, $a?x$ means:

$$\begin{array}{ll} a?x = & (\ \overline{\langle a,0\rangle}\ \rightarrow\ \langle a,0\rangle\ ;\ x:=0 \\ & [\!]\ \overline{\langle a,1\rangle}\ \rightarrow\ \langle a,1\rangle\ ;\ x:=1 \\ &) \end{array}$$

The use of variables is not restricted to input communications only. When a variable occurs in an output expression, we mean its value and take care of a proper initialization. In general, when a variable is used in an expression, its value is meant. The evaluation of expressions takes zero time. When proper modelling requires that time is spent, a delay δ is added. For simplicity's sake, in the specification of a process we relinquish from an explicit statement about the type of the variables and take care of a consistent type usage. Whenever necessary, we explain the meanings of variables in advance.

The introduction of variables requires an extension of the definition of programs. We do not work it out completely, but roughly indicate the changes involved; a similar extension is thoroughly described in [42]. For input channel a and variable x, $a?x$ is a program and, for expression e and output channel a, $a!e$ is a program. An input $a?x$ contributes the value of x to a substitution function. This substitution function is needed for evaluating expressions that contain variables. Another extension to programs is the *assignment* which for variables x_i, $1 \le i \le N$, and expressions e_i looks like:

$$x_1, ..., x_N := e_1, ..., e_N$$

An assignment is not an action and consumes zero time; it only contributes to the substitution function. We restrict ourselves to the parallel composition of programs that have no shared variables.

Example 3.2
Process *And* has input channels a and b, and output channel c, all of type $\{0,1\}$. The process repeatedly reads an input from both a and b, and afterwards it sends the 'logical and' of the input values to c.

```
And = a?x , b?y
      ; c!(x * y)
      ; And
```

In basic program terms, the corresponding specification is:

```
And = ( ⟨$\overline{a,0}$⟩ → ⟨$a,0$⟩ ; $x := 0$
      ▯ ⟨$\overline{a,1}$⟩ → ⟨$a,1$⟩ ; $x := 1$
      )
      , ( ⟨$\overline{b,0}$⟩ → ⟨$b,0$⟩ ; $y := 0$
      ▯ ⟨$\overline{b,1}$⟩ → ⟨$b,1$⟩ ; $y := 1$
      )
      ; ⟨$c, x * y$⟩
      ; And
```

□

In order to describe possible data-dependent continuations, we need another extension of our programs. We introduce the *selection construct*,

$$\textbf{if } B_1 \rightarrow S_1 \;[\!]\; ... \;[\!]\; B_N \rightarrow S_N \textbf{ fi}$$

in which the guards B_i, $1 \le i \le N$, are boolean expressions and the S_i are programs. We assume that there is no time spent in the evaluation or passage of a guard. When several alternatives are possible, a non-deterministic choice is made.

Example 3.3
In terms of the newly-introduced notations, a possible specification of the inverter in Example 3.1 is:

```
Inverter = a?x
         ; if x = 0 → b!1
           [] x = 1 → b!0
           fi
         ; Inverter
```

The same process is specified by program *Inverter'* as follows:

```
Inverter' = a?x
          ; b!(1 − x)
          ; Inverter'
```

Yet another possible specification of the same inverter is:

```
Inverter'' = a?x
           ; x := 1 − x
           ; b!x
           ; Inverter''
```

□

Example 3.4
Process *Filter*, with input channel a and output channel b, both of type Int, filters out all negative numbers that are received via a. The specification of the process is given by the following program:

```
Filter = a?x
       ; if x ≥ 0 → b!x
         [] x < 0 → ε
         fi
       ; Filter
```

Using basic program terms only, the specification of this process is a stiff job. Note that the channel use or *communication behaviour* of process *Filter* is influenced by the values read from channel a.
□

It is possible to choose between communications from different input channels. To have a more convenient notation for denoting the possible probe actions, we use the *probe function* which is applicable to input channels only [28]. For input channel a of type $\{0, 1\}$, the probe on a, which is denoted by $\bar{a}$, equals:

$$\bar{a} = \overline{\langle a, 0\rangle} \vee \overline{\langle a, 1\rangle}$$

An illustration is given in the next example.

Example 3.5
Process *Converge* sends values that are read from either input channel a or input channel b to output channel c. We assume that all channels are of type Int.

```
Converge = ( ā → a?x ; c!x
           [] b̄ → b?y ; c!y
           )
           ; Converge
```

Note that because of the guards, there is no extra suspension in the actions $a?x$ and $b?y$. In fact, there is no real need to use two variables, just one suffices. The same process is specified by:

```
Converge' = ( ā → a?x
            [] b̄ → b?x
            )
            ; c!x
            ; Converge'
```

□

Sometimes, we use a channel to transmit only one kind of message. Such a communication serves to *synchronize* the processes. We say that these channels are of type *Signal*, and we omit the message, say '$\surd$', in the input and output expression: $a?$ denotes an input, and $a!$ denotes an output communication. Similarly, communication action $\langle a, \surd \rangle$ is abbreviated to a.

Although we have a controllable choice construct at our disposal, the choice is restricted to input actions only. As an extension we introduce a type of choice on output by extending the communication protocol. For that purpose, we discriminate between *active* and *passive communications* [29], with the restriction that active inputs are connected to passive outputs and vice versa. An active communication starts with a request to communicate and, as soon as the passive partner has read the request, the communication happens; meanwhile, the execution is suspended. Note that only active inputs and passive outputs require an extension of the communication protocol: for passive inputs and active outputs, the probe action belonging to the output action serves as a communication request. For each channel, the allocation of active and passive side remains unchanged during execution. Hence, either the sender or the receiver activates all communications. When the sender takes the lead, the type of communication is called *data driven*, otherwise it is called *demand driven*. With respect to synchronization channels there is no need to add an extra request: we can choose the input and output side of the channels.

In the underlying model, communication requests are explicitly modelled: active input and passive output channels a are paired with an additional channel a' of type Signal, which runs in the opposite direction, from active to passive. Hence, a' is an input channel for the passive side and an output channel for the active side. Since we want to avoid explicit requests in our programs, we introduce an abbreviation rule for the communications: we add symbol '•' as a superscript to the channel name of active communications, whereas the passive equivalents get symbol '∘' [4, 5]. As a result, for channel a with request channel a', we have the following four possibilities:

$$\begin{aligned}
a^{\bullet}?x &= a'! \ ; \ a?x \\
a^{\circ}?x &= a?x \\
a^{\bullet}!e &= a!e \\
a^{\circ}!e &= a'? \ ; \ a!e
\end{aligned}$$

In fact, there is no need for a communication to wait for the scheduling of its request and, hence, request and communication action may be enabled at the same time. Note that request channel a' has a probe action, namely $\bar{a}'$.

Example 3.6
A specification of the inverter, with demand-driven communications, is given by the following program:

$$\begin{aligned}
\mathit{Inverter} = \ & a^{\bullet}?x \\
& ; b^{\circ}!(1-x) \\
& ; \mathit{Inverter}
\end{aligned}$$

The meaning of the program in basic terms, with explicit communication requests, is:

$$\begin{aligned}
\mathit{Inverter} = \ & a'! \ ; \ a?x \\
& ; b'? \ ; \ b!(1-x) \\
& ; \mathit{Inverter}
\end{aligned}$$

□

Example 3.7
A choice on output is performed by process *Diverge*, with demand-driven communications along its channels a, b, and c, all of type Int.

$$\begin{aligned}
\mathit{Diverge} = \ & a^{\bullet}?x \\
& ; (\ \bar{b}' \rightarrow b^{\circ}!x \\
& \quad [] \ \bar{c}' \rightarrow c^{\circ}!x \\
& \quad) \\
& ; \mathit{Diverge}
\end{aligned}$$

□

Instead of using a separate notation for denoting the guards in an external-choice construct, the guards are integrated with those of a selection construct. Since we do not want the request channels to occur in the programs, the applicability of the probe function is extended to passive channels. As a result, the probe on passive output channel a is denoted by $\bar{a}$ instead of $\bar{a}'$. The probe on passive channel a evaluates to true as soon as there is a pending communication on input channel a or a'; otherwise the probe returns false. An operational interpretation of a probe-containing selection is: when none of the guards equals true, the operation is suspended until at least one of the guards evaluates to true. As an illustration, consider the following program with boolean expressions X and Y, no probes in X and Y, precondition $X \vee Y$, and data-driven communications.

```
**if** $X \wedge \bar{a} \rightarrow S$
$[\!]$ $Y \wedge \bar{b} \rightarrow T$
**fi**
```

The meaning of the program in basic terms is:

```
**if** $X \wedge \neg Y \rightarrow (\bar{a} \rightarrow S)$
$[\!]$ $Y \wedge \neg X \rightarrow (\bar{b} \rightarrow T)$
$[\!]$ $X \wedge Y \quad\rightarrow (\bar{a} \rightarrow S \; [\!] \; \bar{b} \rightarrow T)$
**fi**
```

Example 3.8
Process *MixedCopy* performs data-driven communications along input channels a and b, and output channels c and d, all of type Int. The environment chooses between a and b, and the process copies the inputs from a to c and the inputs from b to d.

```
$MixedCopy =$ **if** $\bar{a} \rightarrow a^{\circ}?x \; ; \; c^{\bullet}!x$
              $[\!]$ $\bar{b} \rightarrow b^{\circ}?x \; ; \; d^{\bullet}!x$
              **fi**
             ; $MixedCopy$
```

□

Thus far, we have not given any meaning to the negation of probes. In order to do so, we consider the following program, with data-driven communications:

```
**if** $\bar{a} \wedge \neg\bar{b} \rightarrow S$
$[\!]$ $\bar{b} \rightarrow T$
**fi**
```

In basic terms, the meaning of this program is:

$$(\bar{b} \rightarrow T \;\lhd\; \bar{a} \rightarrow S)$$

As a result, we can use negated probes for assigning priorities to the alternatives in a choice construct.

The suspension period before a probe-containing selection is not bounded upwardly; it is determined by the environment and may last for ever. Sometimes, however, we want a kind of timeout after which an alternative is selected. For that reason, we add τ as a function which gives the value of the global clock. The returned value of τ is the time at which the preceding program has been finished, as defined in Definition 2.41. Before a choice, we look upon the execution as if it were suspended until τ reaches a value that causes a guard to evaluate to true. We do not restrict the use of τ to guards. A timeout happens when the clock reaches a predefined value. Often, this value depends on the schedule of an action. We denote the schedule of the last communication via channel a by $\sigma.a$; if there is no previous communication, $\sigma.a$ evaluates to 0.

Example 3.9
An example of the use of timeouts is given by the following process, with input channel a and output channels b and c, all of type Int and passive. The process copies values from a to b or c: the one being selected is (partly) determined by the environment.

$$\begin{array}{rl} \mathit{TimedCopy} = & a^\circ?x \\ & ;\ \mathbf{if}\ \bar{b} \wedge \tau < \sigma.a + 5 \ \rightarrow\ b^\circ!x \\ & \ \ [\!]\qquad\ \ \tau \geq \sigma.a + 5 \ \rightarrow\ c^\circ!x \\ & \ \ \mathbf{fi} \\ & ;\ \mathit{TimedCopy} \end{array}$$

The meaning of the selection is:

$$\begin{array}{lll} (& (a,5) & \rightarrow c^\circ!x \\ \lhd & \bar{b} & \rightarrow b^\circ!x \\ ) & & \end{array}$$

□

3.2 Parameterized and generic programs

Due to the incorporation of data into programs, we need to generalize our programs to describe processes whose behaviours depend on the past, for example an N-place buffer, a stack, or even a simple variable. With respect to a variable, once it has been initialized, its future behaviour depends on the value it contains. We express such dependencies by *parameterized programs*, programs in which the relevant part of the past is captured in a set of program variables. A parameterized program is a function that ranges over the set of programs Π; the domain depends on the information that we want to record. An application of a parameterized program, which is called an *instantiation*, selects a program from the range.

Example 3.10
We consider variable *Var*, with input channel a and output channel b, both of type Int and passive. New values are read from a, and upon request the current value is written to b.

$$\begin{array}{rl} \mathit{Var} = & a^\circ?x \\ & ;\ \mathit{Var}'(x) \end{array}$$

When process *Var* has been initialized, it behaves like process $Var'(x)$, which is an instantiation of Var' with variable x.

```
$Var' \in \text{Int} \rightarrow \Pi$, with
$Var'(x) =$ if $\bar{a} \rightarrow a^{\circ}?x$
           [] $\bar{b} \rightarrow b^{\circ}!x$
           fi
          ; $Var'(x)$
```

□

As we have seen in the previous example, an instantiation involves a kind of parameter passing. We will use the following convention: a constant in the function call is treated as a *value parameter*, and a variable as a *variable parameter*. As a consequence, a process instantiated with a certain variable may change the value of that variable, whereas a value initializes a new local variable.

In the following two examples, the domain of the parameterized program consists of a list which is used for recording input values.

Example 3.11
An N-place buffer, with $N > 0$, has input channel a and output channel b, both of type Int and passive. The specification of the buffer with contents $L \in \text{Int}^*$ is given by process $Buffer_N(L)$.

```
$Buffer_N \in \text{Int}^* \rightarrow \Pi$, with
$Buffer_N(L) =$ if $\bar{a} \wedge |L| < N \rightarrow a^{\circ}?l \ ; \ L := Ll$
               [] $\bar{b} \wedge |L| > 0 \rightarrow b^{\circ}!(hd.L) \ ; \ L := tl.L$
               fi
              ; $Buffer_N(L)$
```

Instead of explicitly stating the assignments to L, we use the following program notation:

```
$Buffer_N(L) =$ if $\bar{a} \wedge |L| < N \rightarrow a^{\circ}?l \ ; \ Buffer_N(Ll)$
               [] $\bar{b} \wedge |L| > 0 \rightarrow b^{\circ}!(hd.L) \ ; \ Buffer_N(tl.L)$
               fi
```

An initially empty N-place buffer is described by process $Buffer_N(\epsilon)$.
□

The specification of an infinite stack, which is given in the next example, looks very much like the buffer in the previous example. New values are added to the front end of the list instead of the back end, and there is no limit imposed on the number of values it may contain.

Example 3.12
A stack, with input channel a and output channel b, both of type Int and passive, is specified by:

$$\begin{array}{l} Stack \in \text{Int}^* \rightarrow \Pi, \text{ with} \\ Stack(L) = \textbf{if } \bar{a} \rightarrow a^\circ?l \;;\; Stack(lL) \\ \qquad\qquad\;\; [\!]\;\; \bar{b} \wedge |L| > 0 \rightarrow b^\circ!(hd.L) \;;\; Stack(tl.L) \\ \qquad\qquad\;\; \textbf{fi} \end{array}$$

□

Often, a system contains a number of similarly behaving processes. We specify such processes by a generic representative, a *generic program.* An occurrence is obtained by indexing the program name and all its channels. In order to illustrate this, we consider a system which contains M N-place buffers that are initially empty. This system is described by set

$$\{\, i : 0 \leq i < M : Buffer_N[i](\epsilon) \,\}$$

where occurrence $Buffer_N[i](L)$ equals

$$\begin{array}{l} Buffer_N[i](L) = \textbf{if } \bar{a}[i] \wedge |L| < N \rightarrow a^\circ[i]?l \;;\; Buffer_N[i](Ll) \\ \qquad\qquad\qquad\quad [\!]\;\; \bar{b}[i] \wedge |L| > 0 \rightarrow b^\circ[i]!(hd.L) \;;\; Buffer_N[i](tl.L) \\ \qquad\qquad\qquad\quad \textbf{fi} \end{array}$$

Sometimes, we will define generic programs that are connected to a range of channels. For example, consider program $Mixer_N$ which copies values received via passive channel $a[k]$, $0 \leq k < N$, to active channel $b[N-1-k]$. In the program, the N guards and subsequent alternatives are abbreviated to just one.

$$\begin{array}{l} Mixer_N = \textbf{if } k : 0 \leq k < N : \bar{a}[k] \\ \qquad\qquad\quad \rightarrow a^\circ[k]?x \;;\; b^\bullet[N-1-k]!x \\ \qquad\qquad\;\; \textbf{fi} \\ \qquad\qquad\;\; ;\; Mixer_N \end{array}$$

For $N = 2$, unfolding the selection in the program yields:

$$\begin{array}{l} Mixer_2 = \textbf{if } \bar{a}[0] \rightarrow a^\circ[0]?x \;;\; b^\bullet[1]!x \\ \qquad\qquad\;\; [\!]\;\; \bar{a}[1] \rightarrow a^\circ[1]?x \;;\; b^\bullet[0]!x \\ \qquad\qquad\;\; \textbf{fi} \\ \qquad\qquad\;\; ;\; Mixer_2 \end{array}$$

For occurrences of programs like $Mixer_N$, we adopt the following convention: for passive channels, the new index is added before the one already present in the program; and for active channels, the new index follows the other. The usefulness of this convention is illustrated in Figure 3.1. Occurrence i of program $Mixer_N$ reads:

$$\begin{array}{rl} \mathit{Mixer}_N[i] = & \textbf{if } k : 0 \leq k < N : \bar{a}[i,k] \\ & \quad \rightarrow a^{\circ}[i,k]?x \ ; \ b^{\bullet}[N-1-k,i]!x \\ & \textbf{fi} \\ & ; \mathit{Mixer}_N[i] \end{array}$$

If the above ways of indexing do not apply, the substitutions for the channels in the generic program are explicitly denoted between square brackets. For example, the ith occurrence of an initially empty N-place buffer, with explicit substitutions, is denoted by:

$$\mathit{Buffer}_N[a := a[i], b := b[i]](\epsilon)$$

3.3 Asynchronous communication

A communication happens in both participating processes at the same time, namely as soon as both have enabled the action. This type of communication is called *synchronous communication.* As a result of the enabling model underlying our programs, all communications are synchronous. Sometimes, however, we want to define a process that can do an output irrespective of the readiness of the corresponding input, which means that the output may be scheduled before the matching input. We call this more liberal form of communication *asynchronous.* Of course, an asynchronous input cannot happen before the corresponding output has happened. Since a passive output waits for the active input, in case of asynchronous communications, data-driven communication is implicitly assumed.

The way in which we describe asynchronous communication in our synchronous model is accomplished by adding a sufficiently large FIFO-buffer in the 'asynchronous channel' between the inputting and the outputting process. Consequently, all communications via the buffer are asynchronous. The buffer should always be ready to accept another output from the environment and, when not empty, be willing to supply the environment with the first element. We restrict ourselves to asynchronous communication in which a one-place buffer suffices, otherwise we will explicitly model the required processes. Note that, in order to avoid buffer overflow, this form of asynchronous communication requires a kind of feedback from the receiver to the sender.

As an illustration, consider processes P and Q which communicate asynchronously via channel a, where a is directed from P to Q. In fact, the presence of the one-place buffer gives rise to two a channels, say a_P for the connection with P and a_Q for the connection with Q. The buffer is described by process *AsynBuf*:

$$\begin{array}{rl} \mathit{AsynBuf} = & a_P^{\circ}?x \\ & ; a_Q^{\bullet}!x \\ & ; \mathit{AsynBuf} \end{array}$$

In our specifications we abstract from the underlying solution. In order to denote asynchronous communication, we use the same channel name and the same symbols for input and output as in the synchronous case, but put the symbols upside down: symbol '$\iquestion$' for input and symbol '¡' for output. Although we assume data-driven communications, we add the active or passive sign to keep the communication expressions uniform.

Example 3.13
Process *Merge* has input channels a and b, and output channel c, all of type Int. We assume that both sequences of values from a and b are ascending. The specification of *Merge*, with asynchronous communication, is:

$$\begin{array}{l} Merge = a^{\circ}¿x \ , \ b^{\circ}¿y \\ \qquad\quad ; Merge'(x,y) \end{array}$$

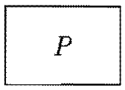
$$\begin{array}{l} Merge' \in \text{Int} \times \text{Int} \rightarrow \Pi, \text{ with} \\ Merge'(x,y) = \textbf{if } x \le y \ \rightarrow \ c^{\bullet}¡x \ , \ a^{\circ}¿z \ ; \ Merge'(z,y) \\ \qquad\qquad\qquad\ \ [] \ \ x \ge y \ \rightarrow \ c^{\bullet}¡y \ , \ b^{\circ}¿z \ ; \ Merge'(x,z) \\ \qquad\qquad\qquad\ \ \textbf{fi} \end{array}$$

□

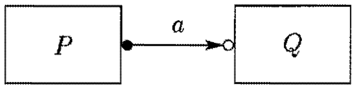
3.4 Pictures

When describing a whole system, a picture that shows the connection diagram of the processes is often indispensable. Such a picture gives an overall view of the structure of the system, without going into too much detail. Therefore, we introduce some graphical symbols and explain their meaning.

We represent processes by rectangles and put their names inside. For example, process P is depicted by:



A channel is represented by an arrow that indicates the direction of the communications. When a channel is used for synchronization only, which means that the channel is of type Signal, we use a line to depict the channel. Close to each channel we put its name. In order to distinguish between the active and passive side of a channel, we add symbol '•' at the active side and symbol '∘' at the passive side. This yields the following picture for processes P and Q which communicate synchronously via channel a, directed from P to Q, with a being active to P and passive to Q.



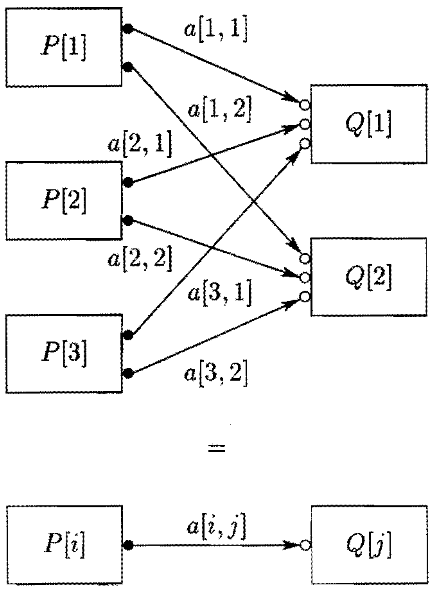
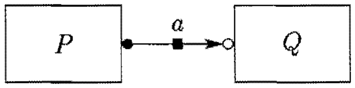


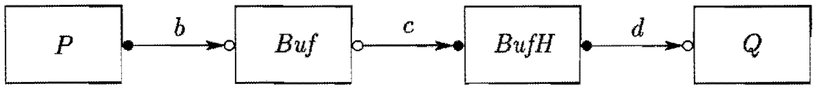
Figure 3.1: An example of a picture and its abbreviation, $1 \le i \le 3$ and $1 \le j \le 2$.

When the communication via a channel happens asynchronously, we indicate this by adding symbol '■' in the arrow or line. Hence, asynchronous communication via channel a between processes P and Q looks like:



When a system contains a number of occurrences of a generic program, we denote, if desired, only a single occurrence. An example is shown in Figure 3.1.

Example 3.14
Suppose a one-place buffer between processes P and Q is insufficient, and a buffer with infinite capacity is required. This is accomplished by adding both an infinite buffer *Buf* and a buffer-handler *BufH*. An overview of the interconnections is:



The buffer *Buf* is: passive in all possible communications; always willing to accept a new value from P; and, when not empty, ready to deliver the first value to the handler. The buffer-handler is active in all communications; it repeatedly asks the buffer for a new value and, afterwards, sends the value to Q. For integer channels, the buffer is specified by:

$$\begin{aligned}
&Buf \in \text{Int}^* \rightarrow \Pi, \text{ with} \\
&Buf(L) = \textbf{if } \bar{b} \rightarrow b^{\circ}?l \; ; \; Buf(Ll) \\
&\qquad\quad [\!] \;\; \bar{c} \wedge |L| > 0 \rightarrow c^{\circ}!(hd.L) \; ; \; Buf(tl.L) \\
&\qquad\quad \textbf{fi}
\end{aligned}$$

The buffer-handler is a special kind of one-place buffer:

$$\begin{aligned}
BufH = \; & c^{\bullet}?x \\
& ; d^{\bullet}!x \\
& ; BufH
\end{aligned}$$

□

3.5 Example: the dining philosophers

A classical control example is the problem of the dining philosophers [10]. Usually, the problem is used to demonstrate the applicability of some (new) synchronization primitives. Unlike prevailing solutions that devise a sort of distributed control scheme, we will simplify matters by giving a solution with a supervisory control.

There are M, $M \geq 2$, philosophers sitting around a table, and each of them alternately thinks and eats. Initially, all philosophers are thinking. In between two neighbouring philosophers there is a fork on the table. A philosopher can eat only if he possesses both his left and his right fork. As a consequence, each fork is shared by two philosophers. After his eating session, a philosopher spends some time thinking. There are no further restrictions imposed on the thinking sessions.

When all philosophers have picked up their left fork and wait for their neighbour to put down the grasped fork, a so-called *deadlock* occurs. In order to avoid the occurrence of such an undesired state, a philosopher asks the supervisory control for permission to pick up both his left and right fork. When receiving the permission, the forks must be available on the table. After the philosopher has finished eating and has put back both forks, he synchronizes with the control again. Philosopher i, $0 \leq i < M$, which is described by process *Phil*$[i]$, interacts with the control process *Control* via channel $a[i]$; the interactions with his left fork *Fork*$[i]$ and his right fork *Fork*$[i+1]$ happen via channels $l[i]$ and $r[i]$ respectively. A picture of the system is given in Figure 3.2. Throughout this example, addition and subtraction are in modulo M arithmetic.

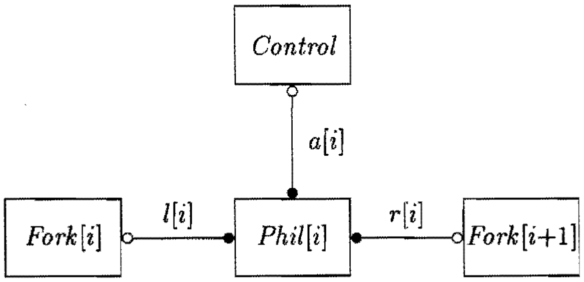


Figure 3.2: The dining philosophers, with $0 \leq i < M$.

3.5.1 Philosophers and forks

A fork is either on the table or possessed by one of its two neighbouring philosophers. Because two philosophers have access, picking up the fork is described by a communication via channel l and by a communication via channel r. Putting down the fork is described similarly. In its initial state, the fork lies on the table.

```
Fork = if $\bar{l}$ → $l^\circ?$ ; $l^\circ?$
       [] $\bar{r}$ → $r^\circ?$ ; $r^\circ?$
       fi
     ; Fork
```

A first impulse to the specification of process *Phil* looks as follows, where *think* and *eat* correspond to a thinking and an eating session:

```
Phil = think
     ; $a^\bullet!$
     ; $l^\bullet!$ , $r^\bullet!$
     ; eat
     ; $l^\bullet!$ , $r^\bullet!$
     ; $a^\bullet!$
     ; Phil
```

Note that in the absence of control, a deadlocking state is likely to occur. When all thinking sessions have the same length, the philosophers are not prohibited from picking up their left fork. This may result in a situation in which all wait for their right fork to become available, which never happens.

Because the philosopher is active in a, the controller to be designed can choose a suspended communication on one of its a channels. We say that the eating and thinking sessions start at the moment on which the control answers the a request. Hence, a communication via a changes the state of the philosopher from eating to thinking or the other way around.

The activities *think* and *eat* need some further elaboration. The only thing about the eating and thinking sessions we want to model is their time consumption. When the required time is constant for a philosopher, say T for the thinking sessions and E for the eating sessions, the activities are properly described by $\delta(T)$ and $\delta(E)$.

The required delay times may, however, vary. We describe such variations by drawing the delay from a certain *statistical distribution*. The actual drawing is described by a distribution function which we will denote by $\mathcal{D}$. We assume that subsequent calls yield a 'quasi-random' sequence of numbers. Moreover, we do not consider the initialization of such functions and assume that the same distribution function yields distinct sequences of numbers in similarly instantiated programs.

In this case, we describe the eating and thinking sessions by a uniform-distribution function over a certain interval, say function $\mathcal{D}_T$ for thinking and $\mathcal{D}_E$ for eating. The resulting specification of process *Phil* reads:

$$\begin{aligned}
\mathit{Phil} = \;& \delta(\mathcal{D}_T) \\
& ;\, a^{\bullet}! \\
& ;\, l^{\bullet}!\,,\; r^{\bullet}! \\
& ;\, \delta(\mathcal{D}_E) \\
& ;\, l^{\bullet}!\,,\; r^{\bullet}! \\
& ;\, a^{\bullet}! \\
& ;\, \mathit{Phil}
\end{aligned}$$

3.5.2 The supervisory control

The task of the control process is to arrange a non-deadlocking scheduling of the eating sessions of neighbouring philosophers. Requests to start eating and to restart thinking arrive via channels $a[i]$. A request from philosopher i to eat may be answered only when both neighbours $i+1$ and $i-1$ are thinking.

In order to record the current states of the philosophers, we introduce variable *think* of type $[0..M) \rightarrow \text{Bool}$, where $\mathit{think}[i]$, $0 \leq i < M$, expresses that philosopher i is thinking. For such an array variable we use $\mathit{think}[i]$ as alias for $\mathit{think}.i$.

In the description of philosopher i we stated that his state changes after a synchronization with the control. As an invariant property of the variable *think* we state the absence of conflicts by:

$$(\forall i : 0 \leq i < M \wedge \neg \mathit{think}[i] : \mathit{think}[i-1] \wedge \mathit{think}[i+1])$$

The control process *Control* chooses between several alternatives, which are guarded by a probe on input $a[i]$, and maintains the invariant property. Consequently, the a channels have to be passive for the control. Pending requests are considered as soon as possible; when there are several possibilities, a non-deterministic choice is made. In the specification of *Control*, we gather similar guarded alternatives into one by explicitly denoting the dummy and its range.

```
$Control \in ([0..M) \rightarrow \text{Bool}) \rightarrow \Pi$, with
$Control(think) =$
  if $i : 0 \leq i < M : think[i-1] \wedge think[i] \wedge think[i+1] \wedge \bar{a}[i]$
    $\rightarrow a^\circ[i]?$ ; $Control(think[i := \text{false}])$
  [] $i : 0 \leq i < M : \neg think[i] \wedge \bar{a}[i]$
    $\rightarrow a^\circ[i]?$ ; $Control(think[i := \text{true}])$
  fi
```

The corresponding system, say DP_M, is given by the set containing a proper instantiation of all programs. Since initially all philosophers are thinking, we take:

$$\begin{aligned} DP_M = \ & \{Control([0..M) \rightarrow \text{true})\} \\ & \cup \{\, i : 0 \leq i < M : Phil[i] \,\} \\ & \cup \{\, i : 0 \leq i < M : Fork[l := l[i], r := r[i-1]] \,\} \end{aligned}$$

In the resulting system, a deadlock is impossible because a philosopher gets permission to eat only when his neighbours are thinking.

The control strategy suffers from so-called individual starvation: it is not guaranteed that each philosopher gets his turn. A strategy which enforces a fair scheduling scheme is not difficult to design: introduce for each philosopher a variable denoting whether he has had his 'next' eating session.

3.6 Example: a turntable

In this section we specify a system that drills a hole in a block and tests whether the drilling has been done successfully. A similar system is described in [33]. The system consists of five processes: a loader, a driller, a tester, a remover, and a turntable. In order to understand their specific tasks, we describe the processing of a single block. The loader takes a block from the environment and puts it on the turntable. The turntable turns clockwise over ninety degrees and brings the block to the driller that drills a hole in it. After another turn of the table, the block arrives at the tester. The tester checks the drilled hole and reports the outcome of the test to the control process. The turntable turns again, and the block is brought to the remover. The control instructs, on basis of the outcome of the test, where the remover has to put the block: on the pile of the correctly or faultily drilled ones. A sketch of the layout of the concrete system is given in Figure 3.3.

Instead of only one block, there can be at most four blocks on the table. Hence, in order to increase the throughput of the system, we aim for parallel processing of four blocks. The blocks that are simultaneously processed by the loader, driller, tester, and remover process are distinct. A turn of the turntable, however, moves all blocks on the table and requires the other processes to have completed their interaction with the blocks. The control process has the obligation to take care of a proper scheduling of the operation of the processes.

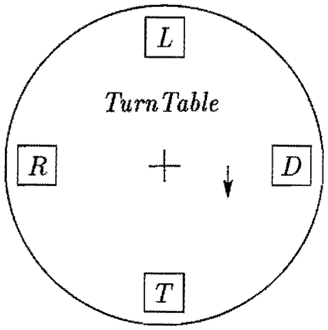


Figure 3.3: A sketch of the turntable and the position of its periphery, where L = *Loader*, D = *Driller*, T = *Tester*, and R = *Remover*.

The connection diagram of the processes in the system is given in Figure 3.4. We close the system by adding a description of the environment, which consists of the programs specifying the input pile and the two output piles. The channels a through d are used to pass the blocks between the processes where it may reside. The others, boolean channels *ct* and *cr*, and synchronization channels *cl*, *ctt*, and *cd*, are needed for the control. We assume the presence of type *Blocks* which gives a proper representation of the blocks. In the current example, the assignment of activeness and passiveness is immaterial; this stems from the fact that there is no external choice in the system.

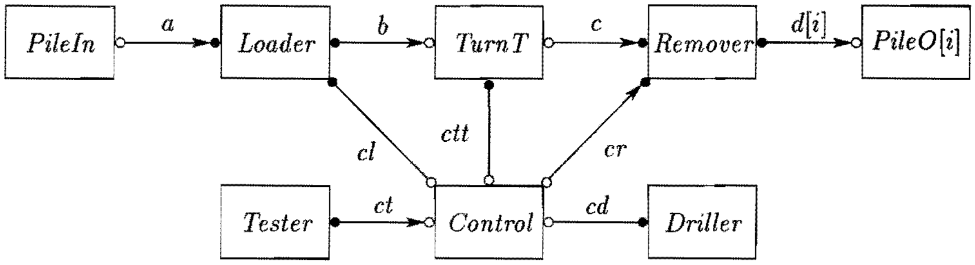


Figure 3.4: The drilling system with input and output piles, with $1 \leq i \leq 2$.

3.6.1 The processes in the system

We continue with a description of the processes. The interaction of a process with the table is controlled by enclosing the corresponding actions between communications with the control. The first communication controls the access, the second serves as a report of its completion. Some interactions consume time; their modelling consists of: an action for denoting the beginning, an action for denoting the end, and a delay for the interval in between. However, if the interval is fixed, the action denoting the end may be omitted, as we will do.

The input pile supplies the loader with new blocks via channel a and has an infinite number of blocks available. In its description *PileIn*, this is described by assuming the presence of function *newBlock* which generates new blocks of type *Blocks*. The specification of the input pile is:

$$\begin{aligned} PileIn = \ & a^{\circ}!newBlock \\ & ;\ PileIn \end{aligned}$$

The loader, which is specified by process *Loader*, gets a new block via a, and moves the block in the right position to put it on the table; this move costs T_L time. After obtaining permission to access the table, which is described by synchronizing via cl, the block is handed over to the turntable via b, which takes T_I time. Next, the loader reports to the control the completion of the interaction and moves back to its original position at the input pile.

$$\begin{aligned} Loader = \ & a^{\bullet}?x \\ & ;\ \delta(T_L) \\ & ;\ cl^{\bullet}!\ ;\ b^{\bullet}!x\ ;\ \delta(T_I)\ ;\ cl^{\bullet}! \\ & ;\ \delta(T_L) \\ & ;\ Loader \end{aligned}$$

The behaviour of the remover resembles that of the loader. The main difference is caused by the dependence on the control which reports the target pile of a block after the interaction with the table.

$$\begin{aligned} Remover = \ & cr^{\bullet}?\ ;\ c^{\bullet}?x\ ;\ \delta(T_I)\ ;\ cr^{\bullet}?target1 \\ & ;\ \delta(T_R) \\ & ;\ \textbf{if}\ \ target1 \ \rightarrow\ d^{\bullet}[1]!x \\ & \quad [\!]\ \neg target1 \ \rightarrow\ d^{\bullet}[2]!x \\ & \quad \textbf{fi} \\ & ;\ \delta(T_R) \\ & ;\ Remover \end{aligned}$$

The tester synchronizes with the control via channel ct. The actual test of a drilled hole consumes T_T time. We assume that the outcome of the test is determined by a distribution function $\mathcal{D}_T$ and a certain threshold C.

$$\begin{aligned} Tester = \ & ct^{\bullet}!\ ;\ \delta(T_T)\ ;\ ct^{\bullet}!(\mathcal{D}_T < C) \\ & ;\ Tester \end{aligned}$$

The driller communicates with the control process via channel cd, and needs T_D time for drilling a hole.

$$\begin{aligned} Driller = \ & cd^{\bullet}!\ ;\ \delta(T_D)\ ;\ cd^{\bullet}! \\ & ;\ Driller \end{aligned}$$

The turntable, which is initially empty, needs to be initialized before it may start its repetitive behaviour. The four blocks that can be present on the table are represented by array variable $x \in [1..4] \rightarrow \mathit{Blocks}$. The loader operates on $x[1]$, the driller on $x[2]$, the tester on $x[3]$, and the remover on $x[4]$. After each turn costing T_{TT} time, the blocks are shifted one place up. The turntable is described by:

$$\begin{aligned}
\mathit{TurnTable} = \ & b^\circ?x[1] \ ; \ \delta(T_I) \\
& ; \ ctt^\bullet! \ ; \ \delta(T_{TT}) \ ; \ x[2] := x[1] \ ; \ ctt^\bullet! \\
& ; \ b^\circ?x[1] \ ; \ \delta(T_I) \\
& ; \ ctt^\bullet! \ ; \ \delta(T_{TT}) \ ; \ x[3], x[2] := x[2], x[1] \ ; \ ctt^\bullet! \\
& ; \ b^\circ?x[1] \ ; \ \delta(T_I) \\
& ; \ \mathit{TurnTable}'(x)
\end{aligned}$$

$$\begin{aligned}
& \mathit{TurnTable}' \in ([1..4] \rightarrow \mathit{Blocks}) \rightarrow \Pi, \text{ with} \\
& \mathit{TurnTable}'(x) = \ ctt^\bullet! \ ; \ \delta(T_{TT}) \ ; \ x[4], x[3], x[2] := x[3], x[2], x[1] \ ; \ ctt^\bullet! \\
& \qquad ; \ b^\circ?x[1] \ , \ c^\circ!x[4] \ ; \ \delta(T_I) \\
& \qquad ; \ \mathit{TurnTable}'(x)
\end{aligned}$$

All processes but the control and the output piles have been described. We omit the description of the output piles, because it is straightforward. A description of the control is given next.

The control process is easy to develop; it takes care of a proper access to the table. As we have seen, all control actions come into pairs, the first one denoting the start, and the second one the completion of the operation. The outcome of the test is kept in variable v of type Bool, which is used in the next cycle. In the specification, we abbreviate for example $(cl^\circ? \ ; \ cl^\circ?)$ to $cl^\circ?^2$. Just like the turntable, the operation of the control begins with a startup phase.

$$\begin{aligned}
\mathit{Control} = \ & cl^\circ?^2 \ ; \ ctt^\circ?^2 \\
& ; \ cl^\circ?^2 \ , \ cd^\circ?^2 \ ; \ ctt^\circ?^2 \\
& ; \ cl^\circ?^2 \ , \ cd^\circ?^2 \ , \ (ct^\circ? \ ; \ ct^\circ?v) \\
& ; \ \mathit{Control}'(v)
\end{aligned}$$

$$\begin{aligned}
& \mathit{Control}' \in \text{Bool} \rightarrow \Pi, \text{ with} \\
& \mathit{Control}'(v) = \ ctt^\circ?^2 \\
& \qquad ; \ cl^\circ?^2 \ , \ cd^\circ?^2 \ , \ (ct^\circ? \ ; \ ct^\circ?w) \ , \ (cr^\circ! \ ; \ cr^\circ!v) \\
& \qquad ; \ \mathit{Control}'(w)
\end{aligned}$$

Now that we have specified all processes, the resulting system *TS* is described by the set of all processes taking part:

$$\begin{aligned}
\mathit{TS} = \ & \{ \mathit{Loader}, \mathit{Remover}, \mathit{Tester}, \mathit{Driller}, \mathit{Turntable}, \mathit{Control}, \\
& \mathit{PileIn}, \mathit{PileOut}[1], \mathit{PileOut}[2] \}
\end{aligned}$$

Chapter 4

Distributed discrete-event simulation

The specification of a complicated system is often hard to fathom and, therefore, reasoning about the design can still be rather difficult. In order to simplify matters, it is rather common practice to build a kind of small-scale model or prototype. We opt for a computer simulation.

The construction of a large industrial system will usually involve a lot of money. Because of the growing complexity, the amounts tend to increase. Therefore, being precise during the development of the specification, the conceptual system, is not enough: certainty is needed that the specification is in accordance with the concrete system required. However, giving a complete specification in advance can be virtually impossible. For example, when the system comprises a complex control process, the resulting consequences may be hard to foresee. Moreover, the flexibility of the system is very important. We want to know how the system will react upon changing market demands, which entails changing the environment. For this purpose, an implementation which simulates the system and its environment is a big help. The implementation enables us to get a profound knowledge about the system's characteristics in relation to possible parameter settings. As a consequence, we are able to validate the specification and, when necessary, change it to meet the requirements that have been imposed.

Typical conditions an implementation must satisfy are: it must be fast and easy to construct. In obtaining an implementation, we benefit from the specification in terms of programs; the program constructs come close to that of existing parallel programming languages. However, implementations do not have the active and passive communication abstractions. Furthermore, implementations are deterministic. The discussion of the implementation approach assumes the *maximal parallelism model*, in which each process is assigned to a processor of its own. In practice, however, a single processor may execute an arbitrary number of processes. Our implementation approach is transparent in the sense that the distinction between processes, which is made in the specification, is also visible in the implementation. We do not choose for the construction of a compiler which transforms the specification into an undistributed, discrete-eventbased simulation, but devise a translation scheme which yields a distributed, event-based network implementation. The reason for doing so is that we try to exploit the capabilities of a processor network. It turns out that the implementation of external choice, or shortly choice, is difficult.

In Section 4.1 we introduce the implementation approach for a subset of systems, namely those without choice.

In Section 4.2 we extend the range of implementable systems with those containing choice, which is achieved by describing how to implement the choice construct.

In Section 4.3 we discuss some optimizations of the implementation that speed up the simulation.

In Section 4.4 our ideas are illustrated by a small example in which we determine the size of a buffer connecting two sub-systems placed in series.

4.1 Absence of choice

Basically, the implementation of a system computes a history up to a specified moment in time. From the resulting history we can derive the statistics required. In general, we are not interested in all actions of the history, but only in those actions that contribute directly to the simulation outcomes. Since we want to know how systems react upon their environment, we assume that the environment is taken into account by incorporating its specification. Consequently, the systems considered are closed.

Due to our aim of a distributed implementation, we exclude the notion of the global clock from the implementation. Instead, each individual process computes its own time τ, which is initially zero. As a result, we obtain an event-based simulation approach. In order to ease the computations on τ, the effect of the implicit unit delays ξ on the value of τ is neglected. As a consequence, τ contains the current macro-moment. In the absence of choice, the primitive programs and the sequential and parallel composition operators remain.

The consequences of a delay are easy to implement. In Definition 2.41 we defined the time consumption of $\delta(M)$ equal to M, where $M \geq 0$. Hence, its implementation consists of a simple increase of τ by M:

$$\tau := \tau + M$$

A communication action happens in both processes at the same time. As a consequence, their times are aligned afterwards. In order to realize this postcondition, the processes have to exchange their local times. Afterwards, the postcondition is validated by assigning to each local τ the maximum of the time values. The time exchange is accomplished by adding a time stamp to each message and the introduction of an additional channel for communication in the reverse direction. The reverse channel associated with channel a is denoted by $\hat{a}$. When a process has performed an input via a, by means of which it obtains the time in the outputting process, the local time is sent via channel $\hat{a}$. For processes P and Q, the annotated implementation of a communication via a looks like:

$$\begin{array}{llll}
P: & \{\tau = X\} & Q: & \{\tau = Y\} \\
 & a!(v,\tau)\ ;\ \hat{a}?\tau' & & a?(w,\tau')\ ;\ \hat{a}!\tau \\
 & ;\ \tau := \tau \max \tau' & & ;\ \tau := \tau \max \tau' \\
 & \{\tau = X \max Y\} & & \{\tau = X \max Y\}
\end{array}$$

As a result of the order imposed on the a and $\hat{a}$ communications, each implemented communication consists of two steps and is completed only when the second step has taken place. In fact, in the absence of choice, the sequential composition of the communications can be replaced by parallel composition. However, in the presence of choice the order is significant.

There remains to describe the effect of sequential and parallel composition on τ. No further attention is paid to these operators, because their effect is given in Definition 2.41.

These minor translations are sufficient to obtain an event-based implementation of systems that are free from external choice. Of course, some further optimizations are possible. For example, in case of passive outputs and active inputs, there is no need to introduce the additional channel because two-way communication is already possible. More interesting systems will be choice holding. Implementing external choice is discussed in the next section.

4.2 Presence of choice

Unfortunately, the implementation of external-choice constructs requires some more effort. From Definition 2.41 follows that the scheduling times of the probe actions are needed to select a proper alternative. These times may be difficult to determine, because it is not guaranteed that the probe actions have been scheduled yet. The exact scheduling time is, however, not always needed. Often, it suffices to know that some actions do not happen before a certain moment. We use this idea in the development of a general strategy to implement choice constructs. The strategy requires the processes to be interdependent: when the execution of a process is locked, all processes become locked eventually. In short, processes are locked before every external choice and before the input step of every communication. After a while, all processes are locked due to the interdependence, and a key is determined to unlock the processes that may continue. To clarify the discussion about the strategy, specifications with data-driven communications are assumed. A further simplification consists of considering sequential processes only.

External choice is restricted to input channels. For simplicity's sake, we assume that the guards consist of: a boolean expression in terms of variables, in conjunction with just one non-negated probed channel. A generalization to more complex guards is relatively easy to obtain. Furthermore, we assume that function τ does not appear in the boolean expressions. As a result, we have so-called stable guards. In order to describe timeouts, we introduce variable *timeout* which contains the moment when the alternative associated with the timeout becomes valid. Variable *timeout* is not used in the boolean expressions and, in the absence of a timeout, *timeout* equals ∞. The alternatives that need to be considered for continuation have boolean expressions that evaluate to true. The names of the channels that belong to these valid boolean expressions are gathered in set F. Note that F contains input channels only.

In order to determine a valid alternative, we need to know the times of the probe actions belonging to the channels in F. Due to the two-phase protocol in the implementation, we can perform the input step of input communications without completing the communications as a whole. As a result of the time stamp added to every message, the input steps give the times required. To record the input communications of which the input step has happened but the subsequent output step has not, each process maintains local set E:

$$E \subseteq \{\, a, m, t : a \in A \wedge m \in \mathit{type}(a) \wedge t \in \mathcal{T} : (a, m, t) \,\}$$

where A is the set of input channels of the process. Initially, no communications have taken place and, hence, E equals the empty set $\emptyset$. For $e \in E$ we use $e.a$ and $e.t$ to denote the input channel and the time stamp respectively. The set of channels currently in E is denoted by $\mathbf{a}E$:

$$\mathbf{a}E = \{\, e : e \in E : e.a \,\}$$

Some choices need no further elaboration: for those with an F which satisfies $F \subseteq \mathbf{a}E$, we can determine an alternative to continue. This situation can be achieved when the environment is willing to communicate via all channels in F. The next chapter describes an example in which this idea is applicable to all choice constructs, at any time. In some cases, however, condition $F \subseteq \mathbf{a}E$ is not needed to select a valid alternative: when there is a triple e in E such that $e.a \in F$ and $e.t \leq \tau$, there is also a legal guard to pass.

In general, the previous conditions do not hold. To determine a proper alternative, processes enter a special mode of operation before every choice, after they have determined their F and *timeout*; such processes are called *locked.* When locked, a process computes on basis of the current triples in E, the channels in F, and the value of *timeout*, the next moment *minTime* when it regains activity:

$$\mathit{minTime} = \mathit{timeout} \min (\,\mathbf{min}\, e : e \in E \wedge e.a \in F : e.t\,)$$

Possibly, processes try to communicate with a locked process. Thus, there can be pending communications on the input channels of a locked process which influence the value of *minTime.* Locked processes are, therefore, willing to communicate via the input channels in their F set and, when an input step has happened, set E and minimum *minTime* are updated. As soon as *minTime* is less or equal to process time τ, or when $F \subseteq \mathbf{a}E$, the locked state can be left, because there is a valid guard available. These conditions are not guaranteed to happen but are used to optimize the strategy. In order to restart its operation, a locked process wants to advance process time τ to *minTime*. However, the advancement may happen only when the value of *minTime* will not become less than the current value owing to other communications. This requires that the times of the processes in the environment are at least *minTime* or can be advanced to *minTime*.

Without any further regulations, a deadlocking state is likely to occur. After a while, processes are either locked or suspended in a communication. To prevent processes from being suspended in input steps, a lock is also added before every input step. We look upon the input steps as if they were choices with just one alternative; for example, input step $a?(w, \tau')$ has $F = \{a\}$ and *timeout* $= \infty$. Furthermore, instead of communicating via channels in F only, locked processes are willing to communicate via all input channels. Consequently, the suspension in output steps is avoided. Note that reading all input channels may happen because of the two-phase communication protocol. In fact, only processes locked before a reverse input step have to take the reverse channel into account. In order to have only one kind of message around, we assume that value '$\surd$' is added to time values sent via reverse channels. Moreover, the range of E is straightforwardly extended to record reverse inputs.

When a locked process reads one of the input channels, the process performing the corresponding output becomes locked too, except for reverse inputs. Eventually, the computation may reach a situation in which all processes are locked. To recover from this situation, the processes with minimum *minTime* are restarted by advancing their times to *minTime*. In order to have access to the processes, they are linked into a *token ring*; this is accomplished by the introduction of additional input and output channels [7, 12]. The ring passes through another process, namely the *TokenHandler*, which initializes the token in the ring. A sketch of the idea is given in Figure 4.1.

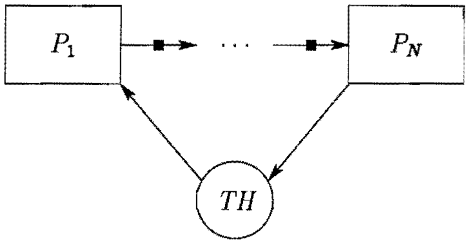


Figure 4.1: An overview of the ring between processes P_i, with $1 \leq i \leq N$, and token handler *TH*.

Process *TokenHandler* is depicted by a circle to distinguish between processes in the specification and the implementation. The channels forming the ring are denoted by plain arrows. In order to avoid deadlock, communication via the ring happens asynchronously, except for the channels connected to the token handler which serves already as a special kind of one-place buffer. The token is used for the following purposes: first, to find out whether all processes are locked and, at the same time, to determine the first moment on which a process can continue; second, when all processes are locked, to restart the processes with minimum *minTime*. As will be apparent later on, recovering from an overall lock costs at least three tours of the token through the ring: at least two are needed to verify whether all processes are locked, and an extra tour is needed to distribute the extracted minimum.

The token is a triple $(phase, allLocked, time)$, with

$$phase \in \{1,2\} \quad , \quad allLocked \in \text{Bool} \quad , \quad time \in \mathcal{T}$$

The *phase* denotes the use of the token. When *phase* equals 1, the token is used to detect whether all processes are locked and to determine the minimum restart time. When receiving the token back, the token handler inspects *allLocked* to find out whether all other processes are locked. When positive, the token is used for phase 2: restart the processes with minimum restart time *time*. Otherwise, the procedure is repeated. The specification of the token handler with input channel c_i and output channel c_o, both of type equal to that of the token, is:

```
$TokenHandler =$
    $c_o!(1, \text{true}, \infty)$
  ; $c_i?(phase, allLocked, time)$
  ; **if** $\neg allLocked \rightarrow \varepsilon$
    [] $allLocked \rightarrow c_o!(2, allLocked, time)$
                      ; $c_i?(phase, allLocked, time)$
    **fi**
  ; $TokenHandler$
```

When the token is used in phase 2, the *allLocked* field is superfluous. The processes react upon the receipt of a token with $phase = 2$ by advancing, if necessary, the process time to *time*.

The operation of locked processes is described by pseudo code S. The code contains boolean variable *passed* which is used to determine the locked state of the whole computation: *passed* records whether the token has passed in this locked state. The ring channels c_i and c_o denote the input channel and the output channel, and the other input channels connected to the process are $a[i]$, $0 \leq i < M$, where M will vary per process.

```
$S =$ $passed := \text{false}$
    ; $minTime := timeout \min(\, \mathbf{min}\, e : e \in E \wedge e.a \in F : e.t\,)$
    ; **do** $\tau < minTime$
        $\rightarrow$ **if** $F \subseteq \mathbf{a}E \rightarrow \tau := minTime$
           [] $F \not\subseteq \mathbf{a}E \rightarrow$ **if** $i : 0 \leq i < M : \bar{a}[i] \rightarrow S_1(i)$
                             [] $\bar{c}_i \rightarrow S_2$
                             **fi**
           **fi**
      **od**
```

Procedure $S_1(i)$ reads the pending input on channel $a[i]$, after which a new triple is added to E and *minTime* is updated.

```
$S_1(i) =$ $a[i]?(m,t)$
        ; $E := E \cup \{(a[i], m, t)\}$
        ; $minTime := timeout \min (\textbf{min}\, e : e \in E \wedge e.a \in F : e.t)$
```

Procedure S_2 describes the reaction upon receiving the token, which depends on the token's phase. In order to speed up the processing of the token, it is sent further through the ring as soon as possible.

```
$S_2 =$ $c_i?(phase, allLocked, time)$
      ; **if** $phase = 1$
          $\rightarrow$ $c_o!(phase, allLocked \wedge passed, time \min minTime)$
             ; $passed :=$ true
      $[]$ $phase = 2$
          $\rightarrow$ $c_o!(phase, allLocked, time)$
             ; $\tau := \tau \max time$
      **fi**
```

Note that when a process is activated by the token, it may wake others before the token has reached these processes. As a consequence, the token may be delayed for some time. The delay causes no errors but we want to avoid it. This is one of the subjects addressed in the next section.

A process locked before a choice leaves its locked state with τ such that there is a valid guard to pass. The remaining selection between the alternatives, which is denoted by program T, is straightforward: take an $e \in E$ with $e.a \in F$ and $e.t \leq \tau$, or select the alternative following the timeout when suitable. With $T_1(e)$ describing the alternative belonging to probed channel $e.a \in F$, and with T_2 the alternative following the timeout, selection T is as follows:

```
$T =$ **if** $e : e \in E \wedge e.a \in F : e.t \leq \tau \;\rightarrow\; T_1(e)$
      $[]$ $timeout \leq \tau \;\rightarrow\; T_2$
      **fi**
```

When $T_1(e)$ completes the communication via $e.a$, e has to be removed from E.

In [30] another approach is described. Instead of adding a ring, the processes send additional time messages to inform their environment about their process time. We did not choose this strategy because it is difficult to implement.

4.3 Some optimizations

With the use of a ring structure we accomplish the implementation of choice-holding computations. As we have mentioned before, an important aspect of implementations is the time needed to compute the information required. Validating the specification is an interactive process and, hence, a fast feedback is essential. Furthermore, a simulation consists of several simulation runs with different initial data. In order to speed up the distributed implementation, we optimize the token transport through the ring. We describe several possibilities of which the actual gain in speed may be considerable.

When a token activates a process, the process may activate others and so on. As a consequence of such a 'chain-reaction,' the token might get stuck in a buffer of the ring structure until the next process is re-locked. Although such a temporal delay in the token transport is not erroneous, the activation of other processes may be delayed. The hold-up of the token is easily avoided: add instead of a ring an extra layer on top of the processes, which takes care of the token transport. A picture of the structure is given in Figure 4.2. To obtain the token, each process P_i, $1 \leq i \leq N$, has

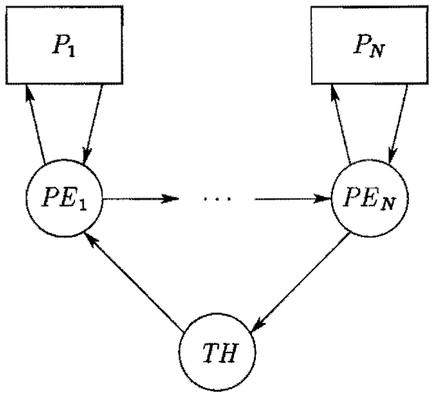


Figure 4.2: The situation in which each process P_i, $1 \leq i \leq N$, has its private entrance PE_i to the ring.

a private entrance PE_i to the ring, which is a kind of one-place buffer. A token with phase 1 passes the P process before it is sent to the next entrance. Upon receiving a token with phase 2, each entrance passes the token immediately to its successor in the ring. The specification of a private entrance is given by process *PrivateEntrance*, with ring channels t_i and t_o, and channels c_i and c_o to establish the connection with the process.

```
PrivateEntrance = $t_i?token$
                  ; if $token.phase = 1 \rightarrow c_o!token$ ; $c_i?token$
                                            ; $t_o!token$
                    [] $token.phase = 2 \rightarrow t_o!token$ , $c_o!token$
                                            ; $c_i?token$
                    fi
                  ; PrivateEntrance
```

Note that entrances make no use of a re-received token with phase 2 from the linked processes. Avoiding this superfluous communication requires a minor update of the processes. The actual gain of this scheme depends on the behaviour of the computation. When there is often the need to activate several processes, the implementation will certainly benefit from it. A drawback of the scheme is the higher communication costs. A tour in the original ring structure between N processes costs $2N$ communications, in which each asynchronous communication is counted twice. In the new scheme, a tour of a token with phase equal to 1 takes $3N+1$ communications; whereas a token with phase 2 requires only N communications to return to the token handler.

In the original ring structure, the token traverses sequentially through all processes. In the preceding discussion on the delayed token we already applied the idea of copying the token to speed up the implementation. We use the same idea to determine an overall lock. The collection of processes is divided into a number of smaller groups. Each group has its local sub-ring and an entrance to the global ring which connects the groups. An overview of this structure is given in Figure 4.3.

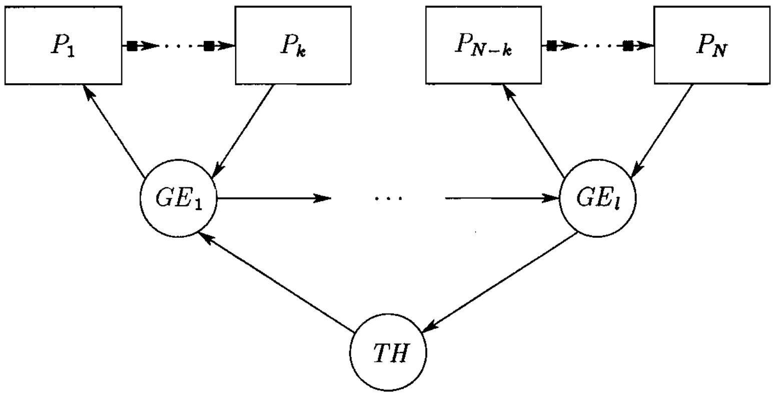


Figure 4.3: A connection diagram with l local rings, where $k * l = N$.

Splitting a group of N processes into $\sqrt{N}$ groups of $\sqrt{N}$ processes yields a local ring of size $2\sqrt{N}$. The entrance processes serve as a kind of local token handler. They initialize the tokens in the local rings and, afterwards, the resulting tokens are added to the token in the global ring. With channels c_i and c_o being part of the local ring, and channels t_i and t_o forming the connection with the global ring, the specification of the local token handler is:

```
$GroupEntrance$ = ( $c_o!(1, \text{true}, \infty)$ ; $c_i?(phase, allLocked, time)$ )
                , $t_i?token$
                ; **if** $token.phase = 1$
                    $\rightarrow$ $t_o!(token.phase, token.allLocked \wedge allLocked,$
                          $token.time$ min $time)$
                  [] $token.phase = 2$
                    $\rightarrow$ $t_o!token$
                      , ($c_o!token$ ; $c_i?token$)
                  **fi**
                ; $GroupEntrance$
```

When the tours in all groups happen simultaneously, it costs $2\sqrt{N}$ communications to compute the local tokens, and another $\sqrt{N}+1$ to collect the local tokens into the global token. In practice, instead of splitting N processes into $\sqrt{N}$ groups, the processes that are mapped on the same processor are linked in a local ring.

There is no need to restrict ourselves to ring structures only, though they are easy to implement. Another possibility is the use of a tree structure on top of the processes. A picture of such a configuration is given in Figure 4.4.

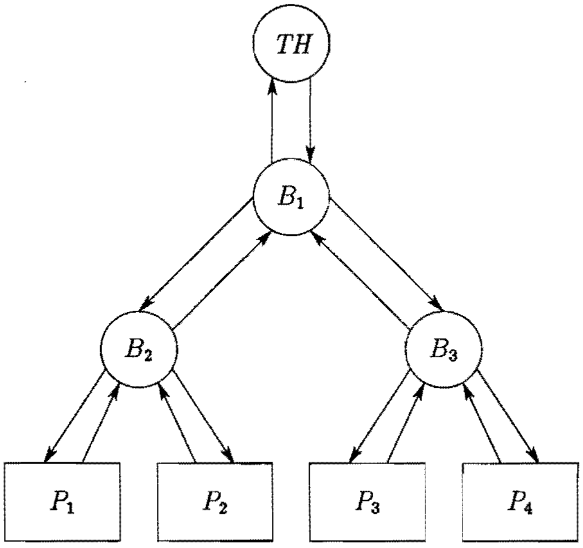


Figure 4.4: A tree structure on top of 4 processes.

Each branch B_i copies a token received from above to its sub-trees and combines the results from its sub-trees. The specification of a branch is given by pseudo code *Branch*, in which the connections to above, left, and right are established by channels t, l, and r respectively. We assume that the leaves do not return tokens with phase 2.

```
Branch = $t_i?token$
         ; $l_o!token$ , $r_o!token$
         , if $token.phase = 1$ → if $\bar{l}_i$ → $U(l_i, r_i, t_o)$
                                   [] $\bar{r}_i$ → $U(r_i, l_i, t_o)$
                                   fi
           [] $token.phase = 2$ → $\varepsilon$
           fi
         ; Branch
```

Procedure U assumes the possibility of using channels as a parameter in procedure calls. Unfolding the first procedure call yields:

```
$U(l_i, r_i, t_o) =$ $l_i?tokenL$
        ; if $tokenL.allvalid$
            $\rightarrow$ $r_i?tokenR$
              ; $tokenR.time := tokenL.time$ min $tokenR.time$
              ; $t_o!tokenR$
        [] $\neg tokenL.allvalid$
            $\rightarrow$ $t_o!tokenL$
              , $r_i?tokenR$
        fi
```

Note that when a sub-tree reports an invalid overall lock, the token of the other sub-tree is not needed to compute the result. Due to the logarithmic nature of trees, for a small number of processes, the speedup obtained by using a tree will be comparable to that of the use of sub-rings. Since the number of processes in the examples we consider is not very large, we will use a sub-ring structure because it is easy to implement.

Although the previous discussion may suggest otherwise, it is not always necessary to link all processes of a system into a ring or tree structure. Sometimes, it is possible to add more than one ring (or tree) structure. The example described in the next section illustrates the use of two rings, with some processes being excluded from both rings.

4.4 Example: buffer-size determination

We consider a simple system that consists of two similar sub-factories which are connected by an N-place buffer, $N \geq 1$. An overview of the system and its environment, which consists of the producer and the consumer of products, is given in Figure 4.5.

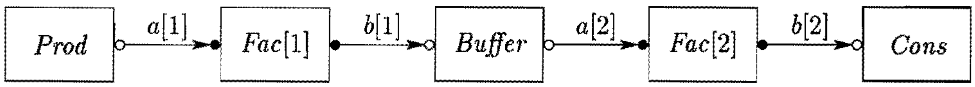


Figure 4.5: An overview of the whole system.

The products that enter the system are processed in both factories, in a fixed order. The buffer is needed to align the operation of the factories: due to some variations in their processing times, the intervals between successive product outputs may vary. We are interested in the throughput per hour as a function of the buffer size.

4.4.1 Specification

The specification of the system requires a closer look at the structure of a factory. A factory consists of two parallel machines that are fed by a single dispatcher. A merge process accepts the processed products from both machines. The contents of factory

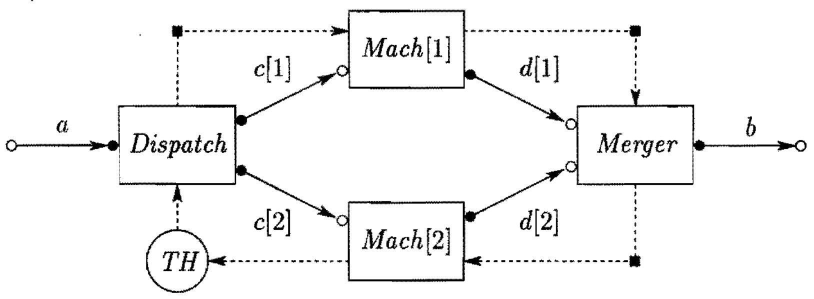


Figure 4.6: The contents of *Factory*, with the dashed lines indicating the ring structure.

Factory are depicted in Figure 4.6. The dashed lines indicate the ring structure that is added in the implementation. The labelled channels are of a proper product type, say *Product*. We continue with a description of the processes.

The machines are specified by program *Machine*. The behaviour is straightforward: a product is read from input channel c and, after a processing step of size $\mathcal{D}_P$, the processed product is sent to output channel d.

```
Machine = c°?x
        ; δ(D_P)
        ; d•!x
        ; Machine
```

A dispatcher gets the products from the producer or the buffer. Instead of sending the product to the first free machine, the dispatcher is a bit clumsy and alternates between the two machines.

```
Dispatcher = a•?x
           ; c•[1]!x
           ; a•?x
           ; c•[2]!x
           ; Dispatcher
```

Unlike the dispatcher, the merge process operates in a smart way. It waits for a machine to report the process completion and the product is accepted and sent to the buffer or the consumer via output channel b.

```
Merger = if d̄[1] → d°[1]?x
         ▯ d̄[2] → d°[2]?x
         fi
       ; b•!x
       ; Merger
```

The specification of the N-place buffer needs no further explanation, it has already been given in the previous chapter. The specifications of the producer and the consumer are not given either. We assume that the environment is maximally cooperative: the producer has an infinite number of products available and is always willing to supply one via its output channel a; the consumer is always greedy to receive processed products via its input channel b.

The resulting system, with an initially empty buffer, is described by the following set of processes:

$$\{Producer, Buffer_N(\epsilon), Consumer\} \cup Factory[1] \cup Factory[2]$$

where $Factory[i], 1 \leq i \leq 2$, equals

$$\{Dispatcher[i], Merger[i],\\ Machine[c := c[1,i], d := d[i,1]], Machine[c := c[2,i], d := d[i,2]]\}$$

4.4.2 Simulation results

We have implemented the system and have used for each factory a ring construct, whose structure is indicated by the dashed lines in Figure 4.6. The ring is needed to implement the choice in the merge process. In fact, the ring is a consequence of the clumsy behaviour of the dispatcher. To make this clear, forget about the ring and suppose the dispatcher is always ready to supply each machine with a new product. Then, the machines are ready to do the next d communication when the previous one has been completed and a new product has been obtained via c. In this situation, the choice in the merge process is easy to implement, because the set of partial completed communications E can be updated such that $F \subseteq E$. As we have described, when $F \subseteq E$, a valid alternative can be determined. This idea is used in the implementation of the buffer. The operation of the factories is mutually independent and, hence, each is given a ring of its own.

In the specification we assumed the existence of type *Products*. The implementation requires an explicit denotation of the structure of this type. Usually, we take the time domain to model the products in a system, because in the resulting computation it is easy to determine the time taken to pass the system. The information we want to extract from the current system does not require the creation time of a product. Therefore, the representation of type products may be a singleton, Signal for example.

Apart from the buffer size N, there is still one parameter left unspecified, namely the processing time generated by distribution function $\mathcal{D}_P$. We describe the processing time of products in the machines by a uniform distribution over the interval between 0 and 100 time units.

For different values of N, the outcomes of the various simulation runs are given in Figure 4.7. The hour mentioned corresponds to a period of 3600 time units. The depicted throughput per hour is the average over the simulated period which started after a startup phase. The maximum of 121.1 is first reached for buffer size 24.

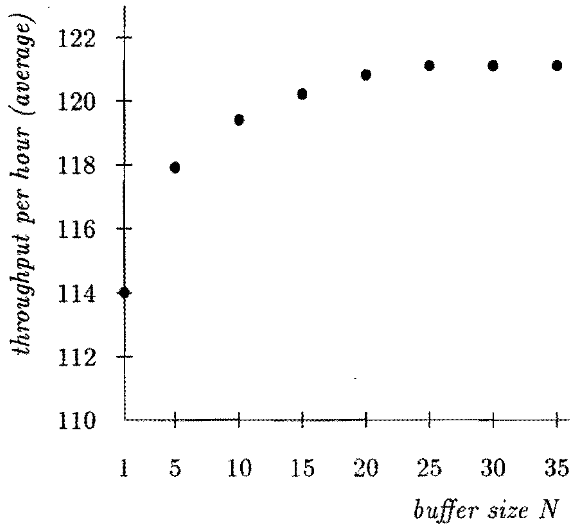


Figure 4.7: The average number of products produced per hour as a function of the buffer size.

On the average, a machine is capable of producing every 50 units a product. As a consequence of the clumsy operation of the dispatcher, the loss in throughput is approximately

$$2 * 3600/50 - 121.1 \approx 23 \text{ products per hour}$$

which corresponds to a decrease in the throughput of about 12%.

We studied several mappings of the system on a network topology. Running all processes concurrently on a single processor consumed 2.8 seconds of execution time. The line structure of the system in Figure 4.5 suggests an implementation on a line topology, in which a factory is mapped on a single processor. The resulting implementation required 2.2 seconds to execute a run. Assigning each of the machines, dispatcher, merge, and token handler to a private processor yields no improved implementation: a single run took 7.5 seconds of execution time. From the resulting figures for various topologies we deduce that, for this example, there is not much to gain from a distributed implementation over many processors, just a few is sufficient. The system is small and does not contain sufficient parallelism to benefit from a distributed implementation.

Chapter 5

A flow-shop factory

A special kind of production facility is described by the class of flow-shop factories. This paradigm serves as a somewhat larger example to illustrate the modelling approach. As will be shown later on, the simulation model is easily obtained from the specification, because there is no need to introduce a ring structure to implement choice constructs in the processes.

Although there is no precise definition of this type of production facility, an unmistakable characteristic of a flow-shop factory is its process-oriented layout. A flow-shop factory consists of a chain of processing units, called shops. Without loss of generality, we assume that the factory is made up of N shops which are consecutively numbered from 1 to N, with $N \geq 1$. Each shop performs an elementary production step: a semi-finished product drawn from stock is joined with another product supplied by its predecessor in the chain. After process completion, the shop sends the processed product to its successor in the sequence. Typical examples of flow-shop factories are found in mass production, for example the production of cars or television sets.

Well-known control strategies for flow-shop factories are based on either *order levels* or the use of cards, so-called *kanbans* [1, 33, 34]. We restrict ourselves to order-level control, which is a kind of distributed control scheme. The order level refers to the buffers in the system, which are needed to absorb temporary mismatches between successive processing units. The objective of order-level control is to minimize the number of products in the whole production line, so as to have the response to a market demand happens *just in time* (the so-called JIT-principle). As soon as the number of products in a buffer falls below its predetermined order level, the buffer issues an order for new products to the relevant supplier. As a consequence, a processing unit manufactures new products only when it has received a request to do so. This type of factory is known as a pull-oriented factory, which is opposite to a push-oriented factory that 'pushes' products onto the market. A major disadvantage of a push factory is the effect a stagnating market has: an accumulation of the unsold products in the buffers.

In Section 5.1 we develop the specification of a shop. Each shop consists of a processing unit, called the work station, and both an input and an output buffer.

In Section 5.2 we link up the shops into a complete factory by adding robots and stores. In order to simulate the system, we close it by adding a description of the market.

In Section 5.3 the free parameters of the system are set and the effect on the performance is studied. To mention one of these performance studies, we vary the order level and look at the corresponding response times of the factory. Moreover, we address a specific implementation issue: mapping processes onto processors. We cluster some processes on a single processor and observe how it affects the execution time of the simulation.

5.1 A single shop

A shop describes a basic unit in the production line and performs an elementary operation. In order to keep the specification of the factory simple, we describe the behaviours of all shops by a single generic program. A shop obtains semi-finished products from its predecessor and delivers the outcome of its processing step to the successor in the sequence. Both predecessor and successor may be other shops or the environment of the factory. We assume that the processing of a product is of a stochastic nature. As a result, successive shops are not attuned to each other and, therefore, need some buffering of the products to align their behaviours. Instead of adding a single, sufficiently large buffer in between each two successive shops, we choose to distribute such a linking buffer over the shops. As a result, each shop has an input buffer, an output buffer, and a processing unit which is called the work station. The work station takes products from the input buffer and joins them with some extra parts obtained from stock; the stock is in the environment of the shop. Afterwards, the processed products are put in the output buffer. An overview of a shop is given in Figure 5.1. The product flow takes place via channels b, c, f, and g. The extra parts are supplied via channel d.

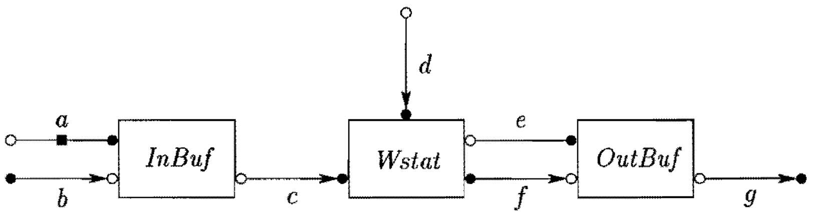


Figure 5.1: A shop in the flow shop factory.

The output buffer controls the operation of the work station. When due to some deliveries via g the number of products in the output buffer drops below a predefined number, which is called the order level, an order for new products is issued via channel e. The work station reacts upon the order by assembling some new products. Analogously, the input buffer orders new products via channel a. Since the supplier of the input buffer may be reluctant for some time to accept an order from a, the input buffer orders new products in an asynchronous way. Each buffer has at most one outstanding order.

A work station is busy only when it has an order of the output buffer. The receipt of an order triggers its operation, and the station collects the required ingredients for a new production cycle. Instead of processing only one product at a time, the operation happens batchwise. The batch size of a work station is given by constant W. The deliveries of products and extra parts have the same batch size. The stochastic behaviour of the processing step is described by a delay of a variable size generated by distribution function $\mathcal{D}_{WS}$. To represent the possible products in the system, we assume type *Products*. The type of the channels c, d, and f is given by the set of all possible lists of length W of type *Products*. The specification of the work station is:

$$\begin{aligned} \mathit{WorkStation} = \; & e^{\circ}? \\ & ;\, c^{\bullet}?\mathit{prods} \;,\; d^{\bullet}?\mathit{parts} \\ & ;\, \delta(\mathcal{D}_{WS}) \\ & ;\, f^{\bullet}!\mathit{prods} \\ & ;\, \mathit{WorkStation} \end{aligned}$$

Note that we do not model the outcome of assembling the extra parts and the semi-finished products. Since our interest is the JIT-principle, it suffices to describe the time consumption only. However, when we want to study the time needed by certain parts to pass through the sequence of shops, an update of the specification is required.

The input buffer maintains list L of semi-finished articles, and operates according to the 'first in first out' principle. The order level of the input buffer is given by constant *Level*. Due to an outstanding order, the environment will deliver the requested products via channel b. Just like the deliveries to the work station, we assume that the incoming products are clustered in batches of size R. As a result, when $R \neq W$, the type of b differs from the type of c. In order to avoid the issue of a new order while the previous one is still unanswered, we introduce boolean variable *ordered*. The interaction with the work station cannot take place unconditionally. A request to supply a batch of W products can be answered only when there are enough products available, which means $|L| \geq W$. Note that this condition requires that the batch size of the work station is at most the order level, $W \leq \mathit{Level}$. The resulting specification of the input buffer is:

```
$\mathit{InBuffer} \in \mathit{Products}^* \times \mathrm{Bool} \to \Pi$, with
$\mathit{InBuffer}(L, \mathit{ordered}) =$
    if $|L| < \mathit{Level} \wedge \neg\mathit{ordered}$
        $\to a^{\bullet}\mathrm{i} \;;\; \mathit{InBuffer}(L, \mathrm{true})$
    [] $\bar{b}$
        $\to b^{\circ}?X \;;\; \mathit{InBuffer}(LX, \mathrm{false})$
    [] $L', M :: \bar{c} \wedge L = ML' \wedge |M| = W$
        $\to c^{\circ}!M \;;\; \mathit{InBuffer}(L', \mathit{ordered})$
    fi
```

The last guard in the previous program yields at most one possibility; the L' and M are introduced to dissect L.

The specification of the output buffer looks very much like the specification of the input buffer. Instead of introducing another order level, we take the same constant *Level* to determine its order condition. In contrast to the input buffer, incoming batches of the output buffer have size W, whereas outgoing batches have size R. Note that this gives rise to requirement $R \leq \mathit{Level}$. Another difference concerns the synchronous way in which orders are sent to the supplier, which is the work station. Since the work station is idle in the absence of an order, it is immediately willing to accept a new order from the output buffer. The specification of the output buffer reads:

```
$\mathit{OutBuffer} \in \mathit{Products}^* \times \mathrm{Bool} \to \Pi$, with
$\mathit{OutBuffer}(L, \mathit{ordered}) =$
    if $|L| < \mathit{Level} \wedge \neg \mathit{ordered}$
        $\to e^{\bullet}! \ ; \ \mathit{OutBuffer}(L, \mathrm{true})$
    $[\!]$ $\bar{f}$
        $\to f^{\circ}?Y \ ; \ \mathit{OutBuffer}(LY, \mathrm{false})$
    $[\!]$ $L', M :: \bar{g} \wedge L = ML' \wedge |M| = R$
        $\to g^{\circ}!M \ ; \ \mathit{OutBuffer}(L', \mathit{ordered})$
    fi
```

We combine the processes of a shop into a subsystem. With respect to the initial state of the buffers we assume that they are empty and have not yet issued an order for new products. As a result, the corresponding subsystem is:

$$\mathit{Shop} = \{\mathit{InBuffer}(\epsilon, \mathrm{false}), \mathit{WorkStation}, \mathit{OutBuffer}(\epsilon, \mathrm{false})\}$$

5.2 The factory and its environment

The flow-shop factory is made by linking N shops together, with each shop given its own local store containing the extra parts. The actual link between successive shops is accomplished by a robot. A robot transports the product batches from the output buffer of the producing shop to the input buffer of the consuming shop. Robots do not appear between shops only, the connections with the environment of the factory are also made with the use of robots. A picture of the factory and its environment, which consists of the producer and the consumer of products, is given in Figure 5.2.

Initially, a robot resides at the output buffer and waits for the arrival of an order of the input buffer to bring a new batch of size R. After receiving the order, the robot picks up the batch (modelled by a communication via g), moves to the input buffer, and hands the batch over to the input buffer (modelled by a communication via b). When the number of products in the output buffer is too small, picking up the batch is suspended for some time. The move of a robot is described by a delay of size *move*. After delivering the batch to the input buffer, the robot returns, with the same cost, to its initial position at the output buffer.

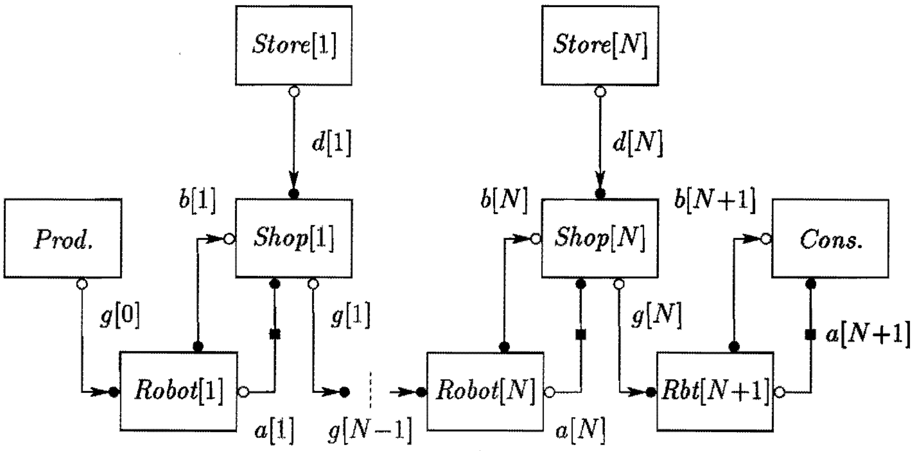


Figure 5.2: An overview of the flow shop factory, $N \geq 1$.

$$\begin{aligned}
\mathit{Robot} = \; & a^{\circ}¿\, ;\, g^{\bullet}?X \\
& ;\, \delta(\mathit{move}) \\
& ;\, b^{\bullet}!X \\
& ;\, \delta(\mathit{move}) \\
& ;\, \mathit{Robot}
\end{aligned}$$

Although it is easily avoided, during the backward move to its original position the robot is unwilling to read an order from its a channel. In order to prevent the input buffer from being unnecessarily suspended, the communications via the a channel happen in an asynchronous way.

The description of a store is kept simple. We assume that it has an infinite number of parts available and is always ready to supply the work station with a new batch. The generation of a batch of size W is described by function $\mathit{newParts} \in \mathit{Products}^{W}$.

$$\begin{aligned}
\mathit{Store} = \; & d^{\circ}!\mathit{newParts} \\
& ;\, \mathit{Store}
\end{aligned}$$

In a sense, the store is superfluous in the simulation, because the effect of the parts on the products is not modelled and, as there is no explicit delay added, the time needed to supply the parts is neglected in the specification. We have mentioned the store in the description of the flow-shop factory because it forms an essential part in the description of the product flow in the factory.

The specification of the flow-shop factory is given by the set consisting of all processes that are part of the system.

$$\begin{aligned}
\mathit{Factory} = \; & (\cup\, i : 1 \leq i \leq N : \mathit{Shop}[i]\,) \\
& \cup \{\, i : 1 \leq i \leq N : \mathit{Store}[i]\,\} \\
& \cup \{\, i : 1 \leq i \leq N{+}1 : \mathit{Robot}[i]\,\}
\end{aligned}$$

A simulation requires the environment to be taken into account. The environment of the factory consists of a product producer and a product consumer. Both producer and consumer are adjusted to the robots in the factory, which means that they also deal with batches of size R. The producer is maximally cooperative: it never fails to supply $Robot[1]$ with a new batch. In the description of the producer, we assume that a new batch of products of size R is generated by function $newProds \in Products^R$. The resulting specification is:

$$\begin{aligned} Producer = \; & g^\circ[0]!newProds \\ & ;\; Producer \end{aligned}$$

Market demands are ordered in the time-domain. The interval between two successive demands issued by the consumer is generated by distribution function $\mathcal{D}_C$. Instead of buffering the demands, we simplify the description by keeping track of the moment upon which the next demand will be issued. As soon as the demand has been fulfilled, the moment of the next demand is generated. In the specification of the consumer, we record the moment of the next demand in variable t.

$$\begin{aligned} & Consumer \in \mathcal{T} \to \Pi, \text{ with} \\ & Consumer(t) = \; \delta((t - \tau) \max 0) \\ & \qquad ;\; a^\bullet[N+1]¡ \\ & \qquad ;\; b^\circ[N+1]?prods \\ & \qquad ;\; Consumer(t + \mathcal{D}_C) \end{aligned}$$

In the instance we consider, the first market demand is issued at moment 0. Collecting the descriptions of the factory and the environment into a closed system yields:

$$Factory \cup \{Producer, Consumer(0)\}$$

5.3 Simulation results

The description developed in the previous section serves as the starting point for the study of a concrete situation. For this purpose we have to transform the specification into its corresponding implementation and substitute specific values for the remaining free system parameters.

Although the descriptions of the input and the output buffer both contain a choice construct, the introduction of a ring structure to implement the choices is not really needed; at any time a valid guard can be determined. In order to make this simplification apparent, we have a closer look at the specification of the input buffer. After a while, the work station will ask for new products and, hence, the time upon which the probe on c evaluates to true becomes available. The robot, however, supplies new products only when it has got an order to do so. As a consequence, the value of *ordered* tells whether the probe on b has to be taken into account. When products have been ordered, the probe on b will eventually become true. For that reason, in the specification of the input buffer we strengthen guard $\bar{b}$ to $ordered \wedge \bar{b}$, which gives:

```
$InBuffer \in Products^* \times \mathrm{Bool} \to \Pi$, with
$InBuffer(L, ordered) =$
    if $|L| < Level \wedge \neg ordered$
        $\to\ a^{\bullet}\mathrm{i}\ ;\ InBuffer(L, \mathrm{true})$
    $[]$ $ordered \wedge \bar{b}$
        $\to\ b^{\circ}?X\ ;\ InBuffer(LX, \mathrm{false})$
    $[]$ $L', M :: \bar{c} \wedge L = ML' \wedge |M| = W$
        $\to\ c^{\circ}!M\ ;\ InBuffer(L', ordered)$
    fi
```

As a result, we can compute a valid alternative to continue the execution. Since an analogous reasoning applies to the output buffer, the implementation of the external choices requires no ring structure.

A remaining question is how to implement type *Products*. The implementation depends on the kind of information that we want to retrieve from the system. Since we are interested in the JIT-principle only, a single value suffices to represent the products. To study the time a product needs to traverse the system, $\mathcal{T}$ is a proper choice. As a result of describing the products by the time upon which they entered the system, it is easy to determine the time needed for the traversal.

The instance we study has the following characteristics. The factory consists of $N = 50$ shops. A work station operates on batches of size $W = 2$, and its processing time is uniformly distributed over interval (0..10). The batch size of the robot is $R = 5$, and a move from the output to the input buffer takes $move = 1$. With respect to the environment we assume that the market demands are described by a Poisson arrival process, with an average of 20 between two successive demands.

In order to determine the effect of the order level on the 'just in time' requirement, we study the average response time of the system as a function of the order level. The response time is the amount of time that elapses between the creation of a demand and obtaining the batch of products. The simulation outcomes are depicted in Figure 5.3. Because of the requirement $R \leq Level$ the search starts at level 5. As expected, a larger order level decreases the average response time. Due to the transport time of the robot that is connected to the consumer, the response time of the factory is at least 1.

Suppose that 4 is an adequate average response time of the system. Then, for the current market, the order level must be at least 15. An interesting study concerns the sensibility of the system to variations in the market demand. Therefore, several other market demands have been filled in and the corresponding response times are shown in Figure 5.4. With respect to the JIT-principle, a stagnating market is harmless: the response times are better than actually needed. On the other hand, a growing market is rather problematic. It appears that only small variations are allowed.

The implementation requires a mapping of the processes onto the available processors. The structure of the system induces a line or ring topology of the network. Processes that are mapped on the same processor are executed in a time-sliced fashion. Communication between processes on different processors is more expensive

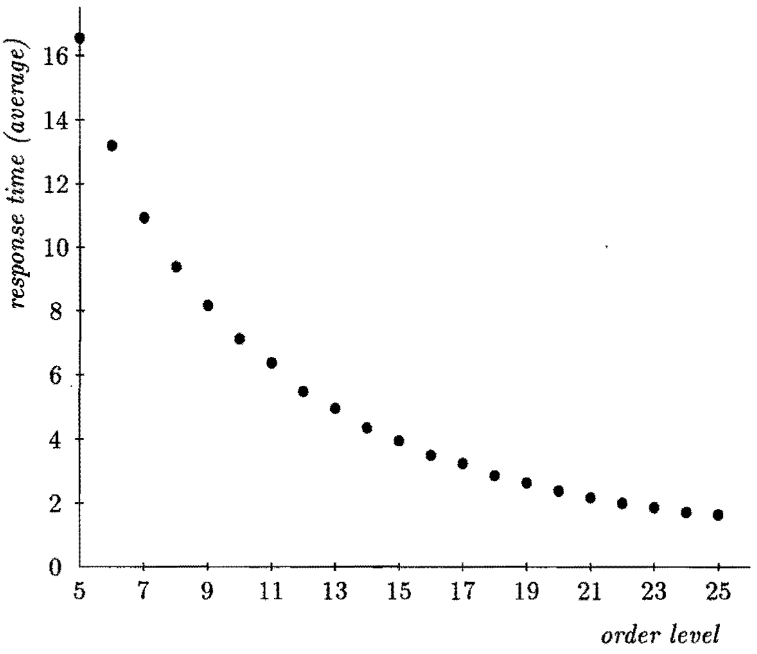


Figure 5.3: The average response time as a function of the order level, for mean time between market demands equal to 20.

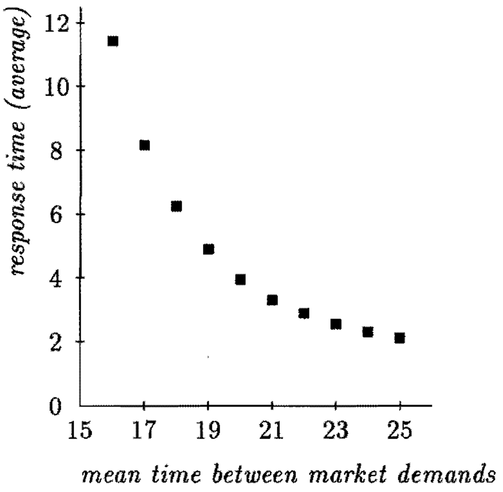


Figure 5.4: The relation between the market and response time, where *Level* = 15.

than between processes on the same processor. However, the number of processes on a single processor should not become too large, because otherwise the sequential time-sliced mode of operation consumes too much time. For the given problem size $N = 50$, we want to determine the number of processors that yields an optimal performance.

We have done some experiments in which we varied the number of processors used. In all implementations, the N shops of the factory are equally distributed over the available processors; for p processors, the first $N \bmod p$ processors are given one shop more. The order level of the buffers is set to $Level = 15$. Note that for $N = 50$ there are 304 processes in the system (N stores, $N + 1$ robots, $3N$ processes in the shops, the producer and the consumer, and $N + 1$ buffers to accomplish the asynchronous communication). The various outcomes are depicted in Figure 5.5. Each run simulated 2000 demands, which corresponds to a period of size $2000 * 20$. The optimum is found for the distribution over 11 processors.

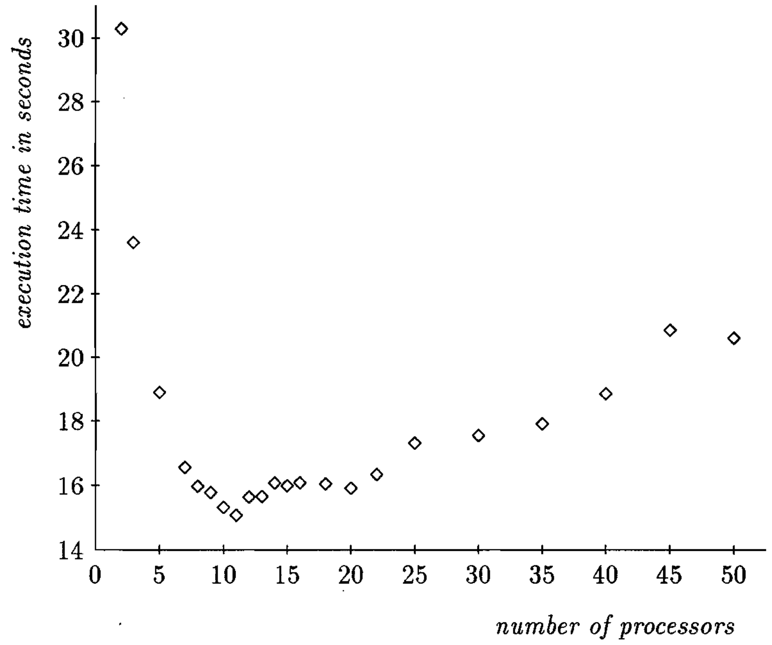


Figure 5.5: The execution time for various numbers of processors.

Usually, the performance of a parallel implementation is expressed in terms of *speedup* and *efficiency* [26]. The speedup compares the execution time of a many-processor implementation to the computing time of the single-processor case. The speedup function does not express the quality of the parallel algorithm: although the speedup is good, the performance may be bad. There is a subtle difference between execution and computing time, namely the execution time is the total amount of time spent on: computing, communicating, and being idle. For each processor in the many-processor implementation, the execution time is the same. In case of a single-processor implementation, the idle time equals zero, but some time will be consumed by internal communications. Since our primary concern is the time the execution takes, we use the execution time instead of the computing time. Besides the dependence on the number of processors used, the speedup is also a function of the problem size—in the running example the problem size is N. With $T(p, n)$ denoting the time p processors need to execute an algorithm solving a problem of size n, the speedup s is a function of p and n, and is defined by:

$$s(p,n) = \frac{T(1,n)}{T(p,n)}$$

The efficiency is a measure for the effective use of the processors taking part and is defined as the speedup per processor:

$$e(p,n) = \frac{s(p,n)}{p}$$

For the current example, we compute a speedup of $50.18/15.06 = 3.3$ for 11 processors; the corresponding efficiency equals 30%. In Figure 5.6 the speedup and efficiency are depicted for $N = 50$.

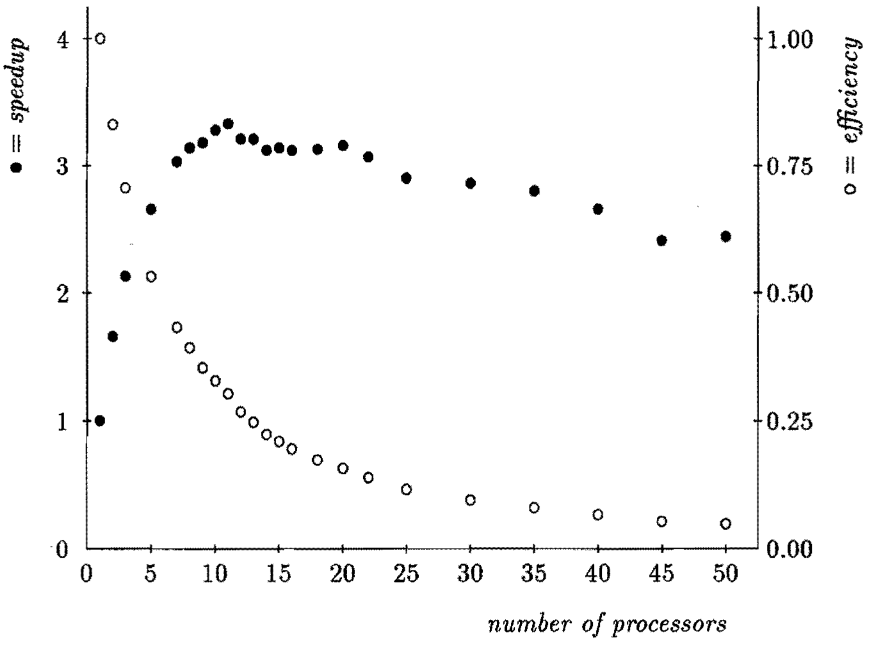


Figure 5.6: The speedup and efficiency as a function of the number of processors, with $N = 50$; they are denoted by symbols '•' and 'o' respectively.

Chapter 6

A lift system

Nowadays, almost all multistory buildings have one or more lifts to transport people or goods from one floor to another. The lifts work simultaneously to reduce the waiting times and to improve the throughput of the system. In this chapter we develop a description of such a parallel-operating lift system [3, 6, 31]. With the use of the corresponding implementation we look at the effect of various control strategies on the performance of the system and study some implementation issues.

We consider a system that consists of M lifts, $M \geq 1$. Each lift has the capacity to transport C people simultaneously. The lifts move between $N + 1$ floors, with $N \geq 1$. The floors are consecutively numbered from 0 to N and the lifts from 1 to M, where floors 0 and N are the bottom and the top floor respectively. We assume that the lift shafts are situated side by side and that they are numbered from left to right. People can arrive at every floor and they join either the 'up-wanting' or the 'down-wanting' queue. It speaks for itself that both the top and the bottom floor have only one waiting queue. In order to form an idea about the structure of the system, the people flow through the system is depicted in Figure 6.1.

On each floor, for both directions up and down, there is a button to signal the control of the lifts that there are people waiting. Inside each lift there is a button for each floor. A person who enters a lift presses the button that signals the destination floor to the control. The control supervises the operation of the lifts and reacts on basis of: the received signals, the status of the lifts, and the imposed control strategy. Although complex control strategies have been devised, we keep the ones we study relatively simple. We have no intention to describe a lift system with a lot of features; we restrict ourselves to the essentials.

In Section 6.1 we develop the specification of the lift system. We start with a description of the up and down buffers. Afterwards, we discuss the behaviours of the lifts and conclude with a (partial) description of the control process.

In Section 6.2 we consider some possible control strategies. They can be substituted in the control process in order to complete its specification.

In Section 6.3 we study the simulation outcomes. We consider in particular the latency and the throughput of the system as functions of the number of lifts. Fur-

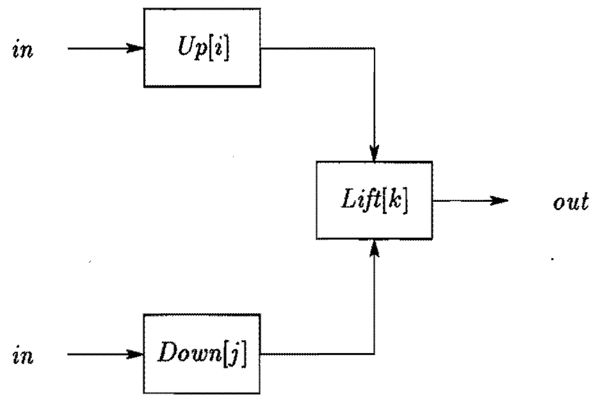


Figure 6.1: The people flow in the lift system, with $0 \leq i < N$, $0 < j \leq N$, and $1 \leq k \leq M$.

thermore, we address some implementation issues.

In Section 6.4 the specification of the lifts is changed so as to include an upper bound on the time taken by an interaction with the control. We compare the performance of the resulting system with the original one.

6.1 System description

Before we specify the processes, we have a closer look at the system as a whole and discuss briefly the interactions. An overview of the lift system together with its environment is given in Figure 6.2. Generators $Gen_U[i]$ and $Gen_D[j]$, which are part of the environment, produce new people: $Gen_U[i]$ generates the up-wanting people on floor i and $Gen_D[j]$ generates the down-wanting people on floor j. The waiting queues are described by FIFO-buffers $Up[i]$ and $Down[j]$; they model the queues with up-wanting and down-wanting people on floors i and j respectively. We consider the behaviour of $Up[i]$. The presence of waiting people is signalled via channel $u[i]$; these signals are called *external requests.* When the control concludes that there are no waiting people left, it sends a signal via channel $p[i]$. Note that the communication via $p[i]$ happens asynchronously. The conclusion of the control can be wrong. In that case, the waiting people react by renotifying their presence by a communication via $u[i]$. People leave the buffer via $x[i,k]$ and enter lift k which is described by $Lift[k]$. The destination floors chosen inside lift k, which are called *internal requests,* are reported to the control via channel $w[k]$. Every now and then lifts require from the control a new direction to move in. For that reason, channel $v[k]$ is introduced. Eventually, the people leave the lift via channel $z[l,k]$ and re-enter the environment which is partly modelled by process $Out[l]$. A similar description applies to the down buffers $Down[j]$. In the following sections we fill in the remaining details about the processes.

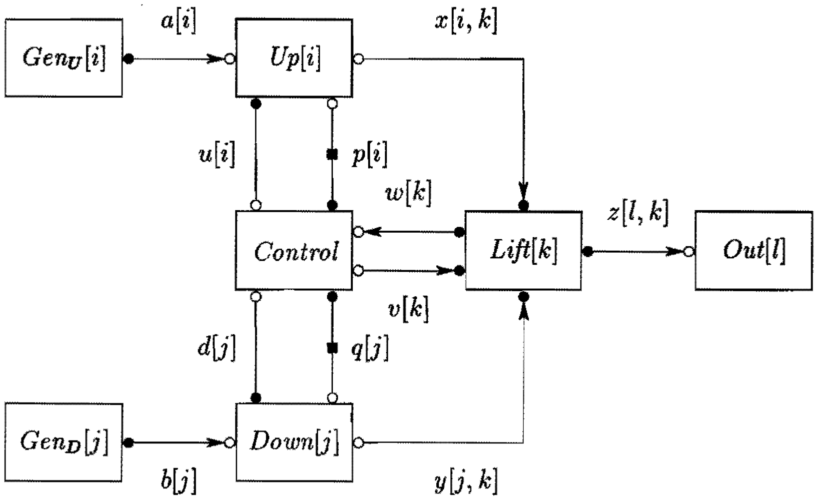


Figure 6.2: An overview of the system, with i, j, and k as before and $0 \leq l \leq N$.

6.1.1 Up and down buffers

Apart from the connected channels, up and down buffers have the same behaviours. Therefore, we restrict ourselves to developing a generic program for the buffers containing the up-wanting people. Buffer process *Up* maintains an ordered list of people which are handled on a first come first serve basis. At most one person enters or leaves the buffer at the same time. Besides the list of people, the buffer process keeps track of whether the control is aware of the presence of people in the buffer. For that purpose we introduce boolean variable *called*, where *called* $\equiv$ true means that the control has been informed. When a person enters the buffer via the a channel, and the control has not been informed about waiting people in the buffer, a signal is sent via channel u. The control cancels this awareness as soon as a lift picks up some people from the buffer. In order to inform the buffer about the cancellation, a signal is sent via the p channel. When receiving a signal from p, the buffer is not necessarily empty and can react by sending a signal via u. The withdrawal of people from the buffer takes place via one of the x channels, which one depends on the number of the lift. For simplicity's sake we introduce the notion of the empty person. When there are no waiting people left and a lift asks for another person, the reply of the buffer consists of the empty person. In a more advanced system specification, there will be a timeout in the description of the lifts to determine the absence of people. In order to represent people, we assume type *People*, in which *empty* denotes the empty person. The specification of process *Up* is as follows.

```
$Up \in People^{*} \times \mathrm{Bool} \to \Pi$, with
$Up(L, called)$ =
  if $\bar{a}$ → $a^{\circ}?l$
            ; if $\neg called$ → $u^{\bullet}!$ ; $Up(Ll, \mathrm{true})$
            [] $called$ → $Up(Ll, \mathrm{true})$
            fi
  [] $k :: \bar{x}[k]$ → if $L = \epsilon$ → $x^{\circ}[k]!empty$ ; $Up(L, called)$
                   [] $L \neq \epsilon$ → $x^{\circ}[k]!(hd.L)$ ; $Up(tl.L, called)$
                   fi
  [] $k :: \bar{p}[k]$ → $p^{\circ}[k]¿$
                   ; if $L = \epsilon$ → $Up(L, \mathrm{false})$
                   [] $L \neq \epsilon$ → $u^{\bullet}!$ ; $Up(L, \mathrm{true})$
                   fi
  fi
```

6.1.2 Lifts

A lift is moving up or down between floors, or it has stopped at a certain floor. With each lift we associate a floor and a direction it is moving in. The floor of a stopped lift equals the floor where it resides. For a moving lift, its floor equals the next floor that will be reached. The set of all possible directions is given by *DIR*, with

$$DIR = \{\square, \triangle, \triangledown, \boxed{\triangle}, \boxed{\triangledown}\}$$

Direction '$\square$' indicates that the lift has stopped at a certain floor and waits for a new move. Directions '$\triangle$' and '$\triangledown$' tell that the lift is moving up and down respectively. The two remaining directions '$\boxed{\triangle}$' and '$\boxed{\triangledown}$' are in a sense a combination of two: the lift has stopped and is still present at that floor, but when its has taken in the up or down-wanting people its next move will be up in case of a '$\boxed{\triangle}$' and down in case of a '$\boxed{\triangledown}$'. The reason for introducing directions '$\boxed{\triangle}$' and '$\boxed{\triangledown}$' is found in the control: when the lift has taken in new passengers, the control reports to the buffer the assumption that no more people are present in the queue. Not all possible changes in the direction of a lift are allowed; the legal changes are denoted in Figure 6.3, where the initial direction '$\square$' is explicitly indicated by the sourceless arrow. For example, an upwards moving lift has to stop before it may start moving in the opposite direction.

When moving up or down, we assume that there is a point between each two consecutive floors at which lifts require a new direction indicating whether they have to move on or to stop. For new directions '$\triangle$' and '$\triangledown$' the concerning lift keeps moving on; for new direction '$\square$' the lift stops at the floor it is heading for. This specific point is reached some fixed time after starting from or passing the previous floor. The timings of the various possibilities are depicted in Figure 6.4. When a lift starts from floor x and moves up to floor $x+1$, point p_x is reached after t_3 time units. A stop at floor $x+1$ takes t_2 time units extra to reach that floor, whereas passing floor $x+1$ consumes t_1 additional time units before point p_{x+1} is reached. Due to some starting-up and slowing-down factors we assume that $t_1 \leq t_2 + t_3$. When the difference between t_1 and $t_2 + t_3$ is considerable, the performance of the system will

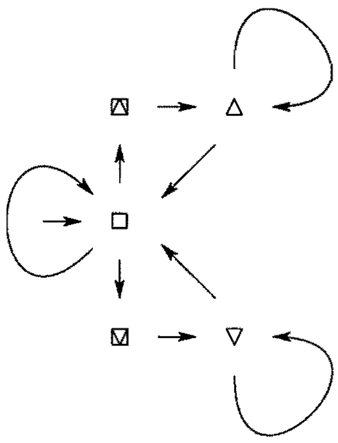


Figure 6.3: Legal changes in the direction of a lift.

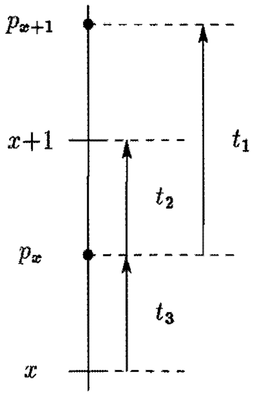


Figure 6.4: The travel times of an up-moving lift approaching floor $x + 1$, with $t_1 \leq t_2 + t_3$.

certainly benefit from avoiding unnecessary stops.

Besides the floor and the direction of a lift, the people inside the lifts have to be taken into account. Therefore, we introduce for each lift an array R in $[0..N] \to \mathcal{P}(People)$, where $R[i]$ gives the set of people with destination floor i.

The specification of the generic lift consists of a case analysis of the current direction and the possible new directions. At the beginning of its specification, the lift requires a new direction and, hence, it is at a point p_x or it has stopped at a certain floor.

```
$Lift \in [0..N] \times DIR \times ([0..N] \to \mathcal{P}(People)) \;\to\; \Pi$, with
$Lift(floor, dir, R) =$
    $v^{\bullet}?newDir$
  ; $\mathbf{if}\ dir \in \{\triangle, \nabla\} \;\to\; S_1$
    $[]\ \ dir = \square \qquad\quad \to\; S_2$
    $[]\ \ dir \in \{⍓, ⍌\} \;\to\; S_3$
    $\mathbf{fi}$
```

The description of S_1 starts from a situation in which the lift is moving up or down. In Figure 6.3 we see that the only possible new directions are: '$\triangle$' in case of an up-moving lift, '∇' in case of a down-moving lift, and '$\square$'. In case of an onwards move the reaction is: the next floor is passed, the floor of the lift is updated, and after t_1 time units a new direction is required. In case of a stop, the next floor is reached after t_2 time units, and we add an extra delay of size *doorOpen* to describe the opening of the door of the lift. As soon as the door is open, the people with this specific destination floor are sent out to the environment.

```
$S_1 = \mathbf{if}\ newDir = \triangle \;\to\; \delta(t_1)\ ;\ Lift(floor + 1, \triangle, R)$
       $[]\ \ newDir = \nabla \;\to\; \delta(t_1)\ ;\ Lift(floor - 1, \nabla, R)$
       $[]\ \ newDir = \square \;\to\; \delta(t_2)\ ;\ \delta(doorOpen)$
                                  $;\ Disembark(R[floor], floor)$
                                  $;\ Lift(floor, \square, R)$
       $\mathbf{fi}$
```

A person takes some time to leave the lift. This time consumption is described by a delay of size *leave*. The set of people that want to leave is denoted by S in the following program.

```
$Disembark \in \mathcal{P}(People) \times [0..N] \;\to\; \Pi$
$Disembark(S, floor) =$
    $\mathbf{if}\ S = \emptyset \;\to\; \varepsilon$
    $[]\ \ s :: s \in S \;\to\; \delta(leave)$
                         $;\ z^{\bullet}[floor]!s$
                         $;\ Disembark(S \backslash \{s\}, floor)$
    $\mathbf{fi}$
```

When the lift has stopped, it asks for a new direction. The possible continuations are described in S_2. Receiving new direction '$\square$' causes no change in the current state. Directions '$\boxslash$' and '$\boxbslash$' indicate a forthcoming move up and down respectively. The lift reacts on the latter two directions by picking up people from the buffer that corresponds to the direction in which the move will be. As a result, a stopped lift is always willing to take new passengers before it moves on.

```
$S_2 =$ if $newDir = \square \;\rightarrow\; \varepsilon$
     [] $newDir \neq \square \;\rightarrow\; Embark(floor, newDir, R)$
     fi
     ; $Lift(floor, newDir, R)$
```

The capacity of the lift is limited to a maximum of C people. A lift picks up as many people as possible; the actual number depends on: the capacity, the number of people already inside, and the number of people in the buffer. A person who enters the lift needs some time to get in. This time consumption is modelled by a delay of size *enter*. Inside the lift, each person selects its target floor by pressing the button that corresponds to the destination floor. In order to select a person's destination floor, we choose a uniform-distribution function $\mathcal{D}_T$ applied to a certain interval. For the up-going direction the interval is $[floor + 1..N]$, and for the down-going direction the interval is $[0..floor - 1]$, where *floor* denotes the current location of the lift. When there are no other people inside the lift who have already selected the same destination floor, the target floor chosen is reported to the control by a communication via channel w. Apart from this use of the w channel, the lift uses it for another purpose as well: when the number of passengers equals the maximum capacity, a special value *full* is communicated via w. As a shorthand for the number of passengers in the lift we use $|R|$, $|R| = (\,\Sigma n : 0 \leq n \leq N : |R[n]|\,)$, where $|R[n]|$ denotes the number of people with destination floor n. Picking up new passengers is now described by:

```
$Embark \in [0..N] \times DIR \times ([0..N] \rightarrow \mathcal{P}(People)) \;\rightarrow\; \Pi$, with
$Embark(floor, newDir, R) =$
   if $|R| = C \;\rightarrow\; w^{\bullet}!full$
   [] $|R| < C$
      $\rightarrow$ if $newDir = \boxslash \;\rightarrow\; x^{\bullet}[floor]?person$
         [] $newDir = \boxbslash \;\rightarrow\; y^{\bullet}[floor]?person$
         fi
        ; if $person = empty \;\rightarrow \varepsilon$
         [] $person \neq empty$
            $\rightarrow\; \delta(enter)$
             ; if $newDir = \boxslash \;\rightarrow\; m := \mathcal{D}_T(floor + 1, N)$
              [] $newDir = \boxbslash \;\rightarrow\; m := \mathcal{D}_T(0, floor - 1)$
              fi
             ; if $R[m] = \emptyset \;\rightarrow\; w^{\bullet}!m$
              [] $R[m] \neq \emptyset \;\rightarrow\; \varepsilon$
              fi
             ; $Embark(floor, newDir, R[m := R[m] \cup \{person\}])$
         fi
   fi
```

We are left with the description of continuation S_3. The lift has possibly taken up new passengers and adopts the subsequent direction up or down. After closing its door, the lift moves for t_3 time units in its new direction and reaches the point at which it requires a new direction again.

$$\begin{array}{rl} S_3 = & \delta(\mathit{doorClose})\ ;\ \delta(t_3) \\ & ;\ \mathbf{if}\ \mathit{newDir} = \triangle\ \rightarrow\ \mathit{Lift}(\mathit{floor}+1, \triangle, R) \\ & \ \ []\ \ \mathit{newDir} = \nabla\ \rightarrow\ \mathit{Lift}(\mathit{floor}-1, \nabla, R) \\ & \ \ \mathbf{fi} \end{array}$$

6.1.3 Control and system

The control process supervises the behaviours of the lifts by supplying the new directions. In order to determine the new directions to move in, the control records in a number of variables the necessary information about states of the buffers and lifts.

A communication via channel $u[i]$ reports the presence on floor i of at least one person who wants to be transported up. To keep track of these up-requests, we introduce set U of type $\mathcal{P}([0..N-1])$, which contains the floors with a request that has not yet been answered by a visit of a lift. Analogously, we introduce set D of type $\mathcal{P}([1..N])$ to record the unanswered down-requests.

The states of the lifts are captured in variable I of type $[1..M] \rightarrow \mathcal{P}([0..N] \cup \{\mathit{full}\})$ and variable L of type $[1..M] \rightarrow ([0..N] \times \mathit{DIR})$. For lift k, $I[k]$ contains the destination floors that have been selected by the people inside the lift. When the number of people in lift k equals C, the special value *full* is also in $I[k]$. The current floor and the current direction of lift k are described by pair $L[k]$.

In order to compute new directions for the lifts, we use control strategy f which depends on the information in U, D, I, and L.

$$\begin{array}{rl} f \in & \mathcal{P}([0..N-1]) \times \mathcal{P}([1..N]) \\ & \times\ ([1..M] \rightarrow [0..N] \cup \{\mathit{full}\}) \\ & \times\ ([1..M] \rightarrow ([0..N] \times \mathit{DIR})) \\ \rightarrow & ([1..M] \rightarrow \mathit{DIR}) \end{array}$$

Function application $f(U, D, I, L)$ yields an array of length M of which the kth value denotes the new direction for lift k. We assume that subsequent directions computed by strategy f satisfy the constraints shown in Figure 6.3. The details of the control strategy are given in the next section.

Control process $\mathit{Control}(U, D, I, L)$ consists of a select statement whose guards are formed by a probe on the input channels. The reactions that are made as a consequence of the internal and external requests via channels $u[i]$, $d[j]$, and $w[k]$, involve a simple update of the relevant set. Some more effort is required for handling the communications via $v[k]$. We have to avoid sending direction '$\square$' to a lift that has stopped: it leaves the state of the lift unchanged and results in the possibility of 'infinite chatter' between lift and control. For that reason, requests that stem from a stopped lift are answered only when the new direction causes a change in the state of the lift. With a domain similar to the one of control strategy f, the specification of the control process is:

```
$Control(U, D, I, L) =$
    $newDirs := f(U, D, I, L)$
  ; **if** $i :: \bar{u}[i]$
      $\rightarrow\ u^{\circ}[i]?\ ;\ Control(U \cup \{i\}, D, I, L)$
    $[]\ j :: \bar{d}[j]$
      $\rightarrow\ d^{\circ}[j]?\ ;\ Control(U, D \cup \{j\}, I, L)$
    $[]\ k :: \bar{w}[k]$
      $\rightarrow\ w^{\circ}[k]?x\ ;\ Control(U, D, I[k := I[k] \cup \{x\}], L)$
    $[]\ k :: \bar{v}[k] \wedge \neg(L[k].dir = \square \wedge newDirs[k] = \square)$
      $\rightarrow\ v^{\circ}[k]!newDirs[k]\ ;\ S_4$
    **fi**
```

After supplying a lift with its new direction, the state information of the lift has to be updated, which is done in S_4. The update depends on both the current and the new direction of the lift. Most of the possibilities are straightforward, though some need some further explanation. A moving lift which is instructed to stop at the next floor, will deliver the passengers who have that floor as their destination. The control cancels the internal request posed for that floor, because it is answered. Furthermore, the control assumes that the maximum capacity is no longer taken. If the assumption is false, it is corrected when the lift is ordered to take in new passengers: the lift will communicate value *full*. For the situation described in Section 6.4 this may occur. Another explanation concerns the updates of sets U and D. Removing a value from one of these sets is preceded by a signal to the relevant buffer, which reports that the awareness of the external request is cancelled. Since the communication via the p and q channels happens asynchronously, the control is never suspended in these output communications.

```
$S_4 =$ **if** $L[k].dir = \triangle \wedge newDirs[k] = \triangle$
          $\rightarrow\ Control(U, D, I, L[k := (L[k].floor + 1, \triangle)])$
        $[]\ L[k].dir = \nabla \wedge newDirs[k] = \nabla$
          $\rightarrow\ Control(U, D, I, L[k := (L[k].floor - 1, \nabla)])$
        $[]\ L[k].dir \in \{\triangle, \nabla\} \wedge newDirs[k] = \square$
          $\rightarrow\ Control(U, D, I[k := I[k] \backslash \{L[k].floor, full\}], L[k := (L[k].floor, \square)])$
        $[]\ L[k].dir = \square$
          $\rightarrow\ Control(U, D, I, L[k := (L[k].floor, newDirs[k])])$
        $[]\ L[k].dir = \boxed{\triangle}$
          $\rightarrow\ p^{\bullet}[floor]$¡ ; $Control(U \backslash \{L[k].floor\}, D, I, L[k := (L[k].floor + 1, \triangle)])$
        $[]\ L[k].dir = \boxed{\nabla}$
          $\rightarrow\ q^{\bullet}[floor]$¡ ; $Control(U, D \backslash \{L[k].floor\}, I, L[k := (L[k].floor - 1, \nabla)])$
        **fi**
```

Apart from the description of the control strategy, all processes in the lift system have now been specified. Initially, we assume that all buffers are empty and have not sent external requests to the control. With respect to the lifts we assume that they are all present at floor 0, they all have direction '$\square$', and there are no people inside. Furthermore, no internal requests have been signalled to the control. Hence, sets U, D, and all sets I are empty. The resulting lift system LS is given by the following set.

$$\begin{aligned} LS = \; & \{\, i : 0 \leq i < N : Up[i](\epsilon, \text{false}) \,\} \\ \cup \; & \{\, j : 0 < j \leq N : Down[j](\epsilon, \text{false}) \,\} \\ \cup \; & \{\, k : 1 \leq k \leq M : Lift[k](0, \square, ([0..N] \rightarrow \emptyset)) \,\} \\ \cup \; & \{\, Control(\emptyset, \emptyset, ([1..M] \rightarrow \emptyset), ([1..M] \rightarrow (0, \square))) \} \end{aligned}$$

6.2 Control strategies

The specification of the control process assumes control strategy f. Until now, the only requirement that has been imposed on the strategy is that it computes legal new directions, but there are more requirements. First of all, floors with external requests should be visited by a lift which transports the people in the required direction. Lifts that have stopped, with surplus capacity, are willing to pick up people that want to travel in the direction intended. Once the people have put the internal requests, the destination floors have to be taken into account. In order to improve the performance of the system, we require that the lifts do not stop unnecessarily. This means that lifts stop only at floors that are internally requested or that contain waiting people. Furthermore, when the maximum capacity of a lift is being used, the next stop is at the nearest destination floor.

In order to distribute the work over the lifts, we associate with lift k two sets of floors, namely $F_U[k]$ of type $\mathcal{P}([0..N))$ and $F_D[k]$ of type $\mathcal{P}((0..N])$. These sets denote the buffers for which the lift is responsible: $F_U[k]$ gives the floors of its up buffers, and $F_D[k]$ gives the floors of its down buffers. As a consequence, external requests affect the behaviours of only those lifts k that are responsible. For every up and down buffer we require that at least one of the lifts is responsible for the requests:

$$\begin{aligned} & (\cup k : 1 \leq k \leq M : F_U[k]\,) = [0..N) \\ & (\cup k : 1 \leq k \leq M : F_D[k]\,) = (0..N] \end{aligned}$$

Later on we will change the control strategy by varying the assignment of floors to these sets.

A lift keeps moving in the same direction as long as there are destination floors to be reached in that direction, or when it will reach a floor with an external request for which it is responsible. The description of the control strategy is given in terms of variables F_U and F_D, and is presented as a case analysis of the current direction of lift k.

When the current direction of the lift equals '$\boxed{\triangle}$', there is only one possible new direction, namely '$\triangle$'.

$$f(U, D, I, L)[k] = \textbf{if } L[k].dir = \boxed{\triangle} \;\rightarrow\; \triangle$$

An analogous definition applies when the current direction is '⍌'. For up-moving lift k we introduce $g(k)$ as a shorthand for 'there is an external request on one of the floors above $L[k].\mathit{floor}$ and lift k is responsible.' Formally,

$$g(k) \equiv (\,\exists i : i > L[k].\mathit{floor} : i \in F_U[k] \cap U \ \vee \ i \in F_D[k] \cap D\,)$$

Up-moving lifts stop when the floor approached is internally requested or when there is a relevant external request. A non-full lift moves on when the next floor has no external request in its direction or when it is not responsible for the request. Full lifts take, however, only internal requests into account and move on if the next floor is not among them, thereby disregarding external requests. There must be a reason for the onward move, which can be: $g(k)$ holds or a higher floor has been requested internally. In all other situations, lifts are instructed to stop.

```
$f(U,D,I,L)[k] =$
  if $L[k].dir = \triangle$
    $\rightarrow$ if $L[k].\mathit{floor} \in I[k] \ \rightarrow\ \Box$
       [] $L[k].\mathit{floor} \notin I[k]$
         $\rightarrow$ if $\mathit{full} \in I[k] \ \rightarrow\ \triangle$
            [] $\mathit{full} \notin I[k]$
              $\rightarrow$ if $L[k].\mathit{floor} \in F_U[k] \cap U \ \rightarrow\ \Box$
                 [] $L[k].\mathit{floor} \notin F_U[k] \cap U$
                   $\rightarrow$ if $|I[k]| > 0 \vee g(k) \ \rightarrow\ \triangle$
                      [] $|I[k]| = 0 \wedge \neg g(k) \ \rightarrow\ \Box$
       fi    fi    fi    fi
```

Note that other lifts can affect $g(k)$ and, hence, situation $|I[k]| = 0 \wedge \neg g(k)$ is possible. Determining the new direction of a down-moving lift is similar to the definition above. When a lift has stopped, it is restarted if there are still internal requests or relevant external requests. With $min(I[k])$ short for $(\,\mathbf{min}\, i : i \in I[k] : i\,)$, we define:

```
$f(U,D,I,L)[k] =$
  if $L[k].dir = \Box$
    $\rightarrow$ if $I[k] = \emptyset \wedge F_U[k] \cap U = \emptyset \wedge F_D[k] \cap D = \emptyset \ \rightarrow\ \Box$
       [] $I[k] = \emptyset \wedge F_U[k] \cap U \neq \emptyset \ \rightarrow\ h(F_U[k] \cap U, L[k].\mathit{floor})$
       [] $I[k] = \emptyset \wedge F_D[k] \cap D \neq \emptyset \ \rightarrow\ h(F_D[k] \cap D, L[k].\mathit{floor})$
       [] $I[k] \neq \emptyset$
         $\rightarrow$ if $L[k].\mathit{floor} < min(I[k]) \ \rightarrow$ ⍓
            [] $L[k].\mathit{floor} > min(I[k]) \ \rightarrow$ ⍌
            fi
       fi
```

where, for set of floors X, $h(X,\mathit{floor})$ is defined by:

```
$h(X,\mathit{floor}) =$ if $(\,\exists x : x \in X : x \geq \mathit{floor}\,) \ \rightarrow$ ⍓
                [] $(\,\exists x : x \in X : x \leq \mathit{floor}\,) \ \rightarrow$ ⍌
                fi
```

Next, we consider some possible assignments of floors to variables F_U and F_D. A further distinction is made between static and dynamic assignments.

6.2.1 Static assignment

In case of a static assignment, sets $F_U[k]$ and $F_D[k]$ are fixed subsets of $[0..N]$. Hence, their contents is independent of the state of the lifts. We try to avoid the situation in which several lifts head for the same buffer. For that reason, we choose the sets in such a way that each buffer appears in exactly one set:

$$(\forall i,j : 1 \le i < j \le M \wedge X \in \{U,D\} : F_X[i] \cap F_X[j] = \emptyset\,)$$

The static assignments are made under the assumption that the number of lifts is at most the number of floors, $M \le N+1$. We define two static assignments: one in which we spread the floors, and another one in which we take consecutive floors. As a result we obtain two control strategies, which are called f_1 and f_2.

Strategy f_1 :
$F_U[k] = \{\, i : 0 \le i < N \wedge i \bmod M = k-1 : i \,\}$
$F_D[k] = \{\, i : 0 < i \le N \wedge i \bmod M = k-1 : i \,\}$

With $m = (N+1) \operatorname{div} M$ and $n = (N+1) \bmod M$, we define the following assignment in which the first n lifts are given $m+1$ consecutive floors and the others m.

```
Strategy $f_2$ :
$F_U[k]$ = if $k \le n$ $\rightarrow$ $[(k-1)*(m+1)..k*(m+1))$
           [] $k > n$ $\rightarrow$ $[n*(m+1)+(k-1-n)*m\,..$
                          $(n*(m+1)+(k-n)*m) \min N)$
           fi
$F_D[k]$ = if $k \le n$ $\rightarrow$ $[1 \max ((k-1)*(m+1))..k*(m+1))$
           [] $k > n$ $\rightarrow$ $[1 \max (n*(m+1)+(k-1-n)*m)..$
                          $n*(m+1)+(k-n)*m)$
           fi
```

Note that in both strategies f_1 and f_2 the up and down buffer of the same floor are assigned to the same lift.

6.2.2 Dynamic assignment

Unlike a static assignment, a dynamic assignment depends on the state of the lifts and, therefore, varies. Consequently, the descriptions of F_U and F_D are given in terms of variable L. We choose an assignment in which the sets contain consecutive, non-overlapping sequences of floors.

We look upon the trajectory of a lift as if it were a circle, where the only possible directions are $\{\Box, \triangle,$ ▨$\}$. Each buffer gives rise to a floor; since both the top and the bottom floor have only one buffer, there are $2(N+1)-2 = 2N$ floors in total. The 'up floors' are numbered from 0 to $N-1$ and the 'down floors' are numbered from N to $2N-1$. Note that the difference between directions '$\triangle$' and '$\triangledown$' and between directions '⍓' and '⍌' is given by the floor of the lift; for example, state $(\text{⍌}, i)$ is translated to $(\text{⍓}, 2N - i)$. A lift with direction '□' and floor i, $i \neq 0$ and $i \neq N$, resides at 2 floors, namely floors i and $2N - i$.

For floors $i \in [0..2N)$ and directions $r \in \{\Box, \triangle, \text{⍓}\}$, we introduce $G(i, r)$ which gives the set of lifts with state (i, r):

$$\begin{aligned} G(i,r) = \{\, k \,:\, & 1 \le k \le M \wedge L[k].\mathit{dir} = r \wedge \\ & ((r \in \{\triangle, \text{⍓}\} \wedge L[k].\mathit{floor} = i) \vee \\ & \;(r = \Box \wedge (L[k].\mathit{floor} = i \vee L[k].\mathit{floor} = 2N - i)) \,:\, k \,\} \end{aligned}$$

The assignment of floor i to a lift is recursively defined by function $H(i)$. When several lifts with distinct states are present at the same floor, the order of preference is given by sequence

$$\text{⍓} \quad \Box \quad \triangle$$

When there are several lifts with the same state, the one with lower number is chosen.

$$\begin{aligned} H(i) = \mathbf{if}\ & G(i, \text{⍓}) \neq \emptyset \ \rightarrow\ \min(G(i, \text{⍓})) \\ [\,]\ & G(i, \text{⍓}) = \emptyset \wedge G(i, \Box) \neq \emptyset \ \rightarrow\ \min(G(i, \Box)) \\ [\,]\ & G(i, \text{⍓}) = \emptyset \wedge G(i, \Box) = \emptyset \wedge G(i, \triangle) \neq \emptyset \ \rightarrow\ \min(G(i, \triangle)) \\ [\,]\ & G(i, \text{⍓}) = \emptyset \wedge G(i, \Box) = \emptyset \wedge G(i, \triangle) = \emptyset \ \rightarrow\ H((i-1) \bmod 2N) \\ \mathbf{fi}\ & \end{aligned}$$

Control strategy f_3 operates on basis of this dynamic assignment. In order to complete the definition of f_3, we state for lift k the contents of $F_U[k]$ and $F_D[k]$.

Strategy f_3 :
$F_U[k] = \{\, i : 0 \le i < N \wedge H(i) = k : i \,\}$
$F_D[k] = \{\, i : N \le i < 2N \wedge H(i) = k : 2N - i \,\}$

For $N = 10$ and $M = 5$, an example of the dynamic assignment is shown in Figure 6.5. The directions of the lifts are straightforwardly denoted; for example, lift 1 has stopped at floor 2, and lift 2 is moving down towards floor 5. The arrows in the picture denote the contents of the F_U and F_D sets. The up-pointing arrows refer to F_U and the down-pointing arrows refer to F_D. A lift with no arrows in its column has an empty F_U and an empty F_D set; lift 4 is an example of such a lift. For lift 1 we find: $F_U[1] = [0..5]$ and $F_D[1] = [1..2]$; and for lift 3 we have: $F_U[3] = \emptyset$ and $F_D[3] = [6..8]$.

6.3 Simulation results

We close the system by adding a description of the environment. Processes $\mathit{Out}[l]$ are maximally cooperative and, hence, do not delay the lift system. The arrival of new people, which is accomplished by processes $\mathit{Gen}_U[i]$ and $\mathit{Gen}_D[j]$, is described by a Poisson arrival process with an average mean time of 100 between

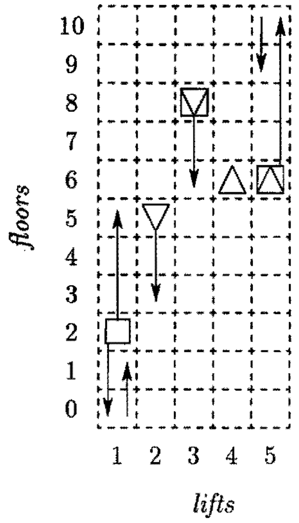


Figure 6.5: An example of the assignment of floors in dynamic mode.

two successive arrivals. The specifications of these processes need no further comment. The various timings in the system are taken as follows. The travel times of the lifts are equal: $t_1 = t_2 = t_3 = 4$. Both opening and closing of the door takes $doorOpen = doorClose = 3$, and a person who enters or leaves consumes $enter = leave = 2$ time units. We take a building with $N = 10$ floors and vary the number of lifts in the simulation runs. Each lift has a capacity of $C = 15$ people.

As a first study we determine the effect of the various control strategies on the latency. The latency of a system is defined as the time needed to pass through the system. For a person in the lift system we take the latency to be the time which elapses between the moment of entering the buffer and the moment of leaving the lift. The latency figures of the different control strategies are given in Figure 6.6. As could be expected, increasing the number of lifts reduces the latency. From the simulation outcomes we deduce that, with respect to the latency, the strategy with the dynamic assignment results in all cases in a better control scheme than the strategies with static assignments. The differences are, however, small. In the other performance studies of the system we will use control strategy f_3.

Next, we consider the throughput of the system, which is the number of people handled during a certain time interval. For a varying number of lifts, we also vary the capacity and observe how the throughput is affected. The outcomes are depicted in Figure 6.7, where the hour mentioned represents 3600 time units. Not very surprisingly, increasing the number of lifts and the capacity results in a larger throughput. The cut-off at throughput 720 is easily explained: for N equal to 10 there are $2N = 20$ person generators which generate 36 people per hour each, which results in a total of 720. Configurations that reach this maximum throughput operate in a kind of saturation: there is an excess of productive capacity.

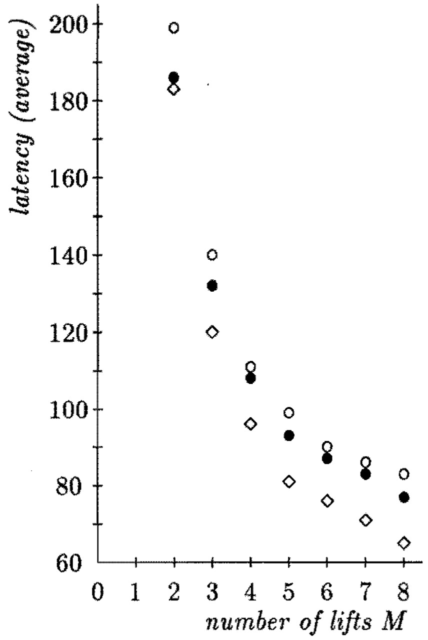


Figure 6.6: The latency as a function of the number of lifts M, where the outcomes of strategies f_1, f_2, and f_3 are denoted by symbols '∘', '•', and '◇' respectively.

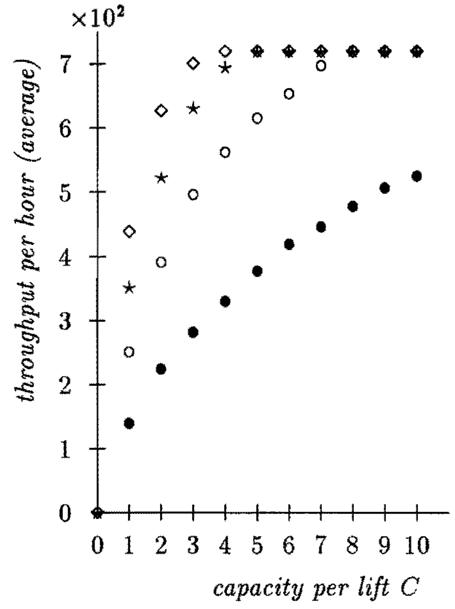


Figure 6.7: The throughput as a function of the capacity, where the situations with 1, 2, 3, and 4 lifts are respectively given by '•', '∘', '⋆', and '◇'.

In order to implement the choice constructs in the specification of the system, a ring construct with local rings has been added. We did some experiments with the implementation: we varied the problem size by changing the number of lifts, and we also looked at the effect of using 6 and 7 processors in the implementation. The simulation results are shown in Figure 6.8.

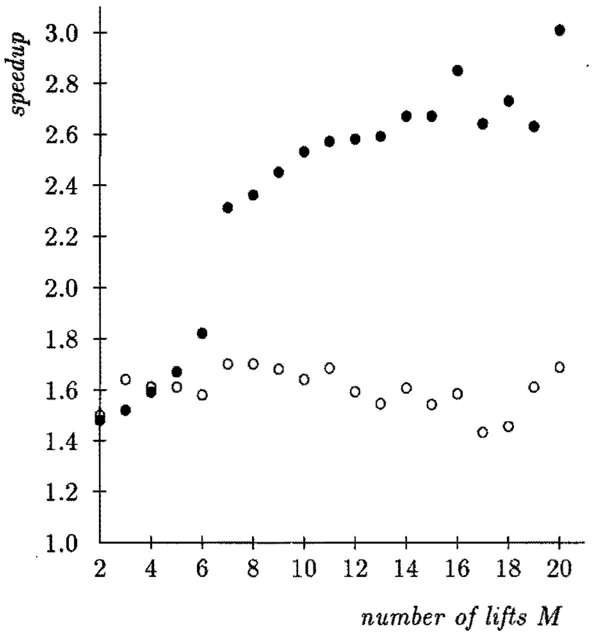


Figure 6.8: The speedup figures as a function of the number of lifts, where the 6 processor implementation is denoted by 'o' and the use of 7 processors by '•'.

Although a large number of lifts has no practical relevance, the numbers have been added to denote the performance of the implementations. From the results follows that, for a small number of lifts, the implementation using 6 processors is slightly faster than the one using 7 processors. Increasing the problem size M shows a clear improvement in the performance of the implementation on 7 processors, whereas the performance of the implementation on 6 processors remains roughly the same. An implementation that uses more than 7 processors yielded no improvement of the speedup. When using n processors, the maximum speedup that can be obtained is also n: the work is spread evenly over the n processors and can happen at most n times faster. Usually, the speedup is smaller because of the communication between the processors. Another cause that reduces the speedup is an imbalanced workload; this results in idle times of the processors.

The execution time of a simulation run is determined by the speed in which the events are handled. As a measure of this speed we look at the average number of time-jumps per second, which is counted in the token handler of the ring construct.

Note that a time-jump occurs when the phase of the token changes from 1 to 2. The results are shown in Figure 6.9. Apart from the number of time-jumps performed by the implementations on 6 and 7 processors, we also looked at the numbers that are achieved without the use of local rings; for both implementations without local rings the results are about the same. From the results it is obvious that the introduction of local rings results in a faster implementation. For the implementation on 7 processors, the local rings improve the speed by a factor of about 2.

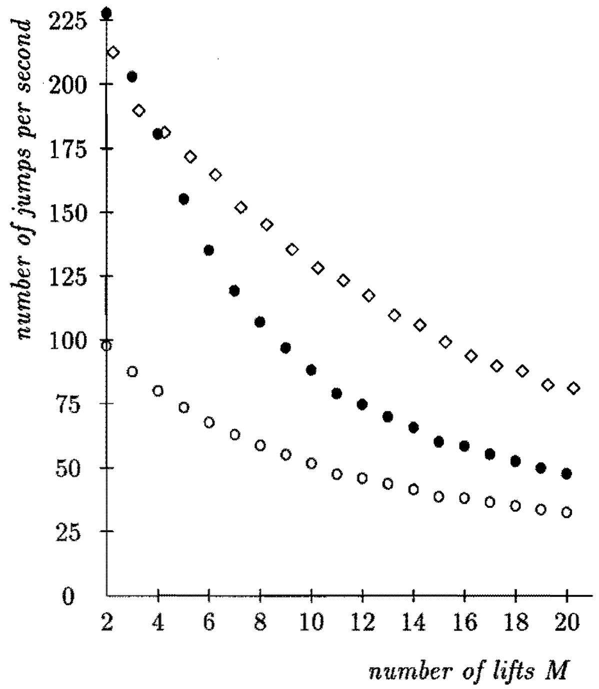


Figure 6.9: The number of time-jumps as a function of the number of lifts, with '•' and '⋄' denoting the 6 and 7 processor implementation respectively; for both cases 'o' gives the numbers in the absence of local rings.

Although the introduction of local rings speeds up the implementation, the number of time-jumps per second is rather low. In order to get an idea about the execution time, the implementation on 7 processors takes 140 seconds to simulate 8 lifts during 3600 time units. Probably, an undistributed discrete-event implementation will perform better. The poor performance is caused by the lack of parallelism in the algorithm. Due to the dense time domain, on every moment there will happen at most one event. As a consequence, a token with phase 2 restarts only one process; the others simply wait for their turn to come. Selecting the next event via the ring construct takes a lot of communications, whereas in the sequential undistributed discrete-event approach the next event is obtained by taking the first element from the ordered event list maintained. The amount of time spent in maintaining the event list determines whether or not the distributed approach is faster.

6.4 Time-critical behaviour

We end this chapter with a small change in the specification of the lift system and study the effect on the throughput. The change concerns the time-critical action $v^{\bullet}?newDir$, where time critical means that the action must happen within a limited amount of time after it has been enabled. When the lift has stopped, the input action may take an arbitrary amount of time. However, an upwards or downwards moving lift needs the new direction within a certain time interval; otherwise there may be a shortage of time left to stop. This problem is solved by the introduction of a so-called *watchdog timer* [18]. The idea is as follows: when the new direction is not in time, a stop is chosen as the new direction of the lift.

Instead of one $v[k]$ channel between lift k and the control we now use three of them, as depicted in Figure 6.10. Lift k asks the control for a new direction by sending

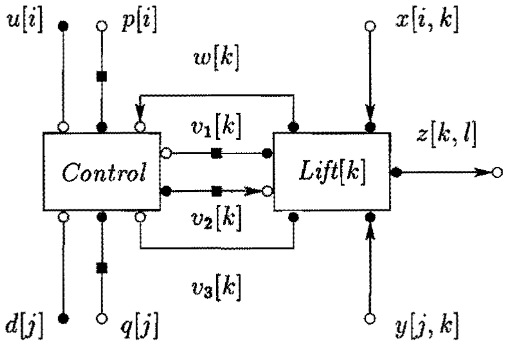


Figure 6.10: The new connection scheme of *Control* and *Lift*$[k]$.

a signal via $v_1[k]$. The subsequent answer is supplied via channel $v_2[k]$. When the answer arrived too late and the lift has stopped, the lift still has the obligation to read the answer of the control. A possible consequence is the introduction of an inconsistency between the lift information maintained in the control and the actual state of the lifts. For example, the control instructs to move on but the lift has chosen a stop. In order to reconcile the lift information in the control with the state of the lifts, we add another v channel, namely v_3 of type Signal. When the direction supplied by the control differs from the new direction of the lift, a signal is sent via v_3 to inform the control about the discrepancy.

In the specification of the lift we replace action $v^{\bullet}?newDir$ by a more complex selection construct which depends on the current direction of the lift. When a lift is moving up or down, the answer of the control may take at most t time after the request has been sent. Note that the requests are asynchronous communications and, hence, they happen immediately. We introduce variable *timedout* to record whether the reply of the control arrived in time.

```
if $dir \notin \{\Delta, \nabla\}$ $\rightarrow$ $v_1^{\bullet}¡$ ; $v_2^{\circ}¿newDir$
[] $dir \in \{\Delta, \nabla\}$
   $\rightarrow$ $v_1^{\bullet}¡$
     ; if $\bar{v}_2 \wedge \tau < \sigma.v_1 + t$ $\rightarrow$ $v_2^{\circ}¿newDir$ ; $timedout :=$ false
      [] $\tau \geq \sigma.v_1 + t$        $\rightarrow$ $timedout, newDir :=$ true, $\Box$
      fi
fi
```

Further changes in the specification of the lift are confined to S_1, the situation in which the lift is moving and has got a new direction. When the new direction has been obtained in time and indicates an onward move, the only change concerns delay $\delta(t_1)$: it is replaced by $\delta(t_1 - (\sigma.v_2 - \sigma.v_1))$ to take the time consumption of the reply into account. A stop is something more elaborate. The time left over to stop depends on the value of *timedout*: when true, the maximum amount t has been consumed and otherwise just a portion. Irrespective of the reason of the stop, a lift opens its door and gives the passengers the opportunity to leave. Afterwards, when *timedout* equals true, the lift reads the direction supplied by the control and, if necessary, reconciles with the control via v_3.

```
if $newDir = \Delta$ $\rightarrow$ $\delta(t_1-(\sigma.v_2-\sigma.v_1))$ ; $Lift(floor+1, \Delta, R)$
[] $newDir = \nabla$ $\rightarrow$ $\delta(t_1-(\sigma.v_2-\sigma.v_1))$ ; $Lift(floor-1, \nabla, R)$
[] $newDir = \Box$ $\rightarrow$ if  $timedout$  $\rightarrow$ $\delta(t_2-t)$
                        [] $\neg timedout$ $\rightarrow$ $\delta(t_2-(\sigma.v_2 - \sigma.v_1))$
                        fi
                       ; $\delta(doorOpen)$
                       ; $Disembark(R[floor], floor)$
                       ; if $\neg timedout$ $\rightarrow$ $\varepsilon$
                        []  $timedout$  $\rightarrow$ $v_2^{\circ}¿newDir'$
                                      ; if $newDir = newDir'$ $\rightarrow$ $\varepsilon$
                                       [] $newDir \neq newDir'$ $\rightarrow$ $v_3^{\bullet}!$
                                       fi
                        fi
                       ; $Lift(floor, \Box, R)$
fi
```

The remaining changes concern the specification of the control process. The control requires some time between the receipt of a request via a v_1 channel and the subsequent reply via the corresponding v_2 channel. We model this time consumption by a delay of variable size; distribution function $\mathcal{D}_W$ generates the size of the delay.

```
if $\bar{v}_1[k] \wedge \neg(L[k].dir = \Box \wedge newDir[k] = \Box)$
   $\rightarrow$ $v_1^{\circ}[k]¿$ ; $\delta(\mathcal{D}_W)$ ; $v_2^{\bullet}[k]¡newDir[k]$ ; $S_4$
```

Note that the control is never suspended in an active communication: they all happen asynchronously.

Handling reconcilements from channels v_3 causes no real problems. A reconcilement happens when the new direction of the lift is not in accordance with the reply

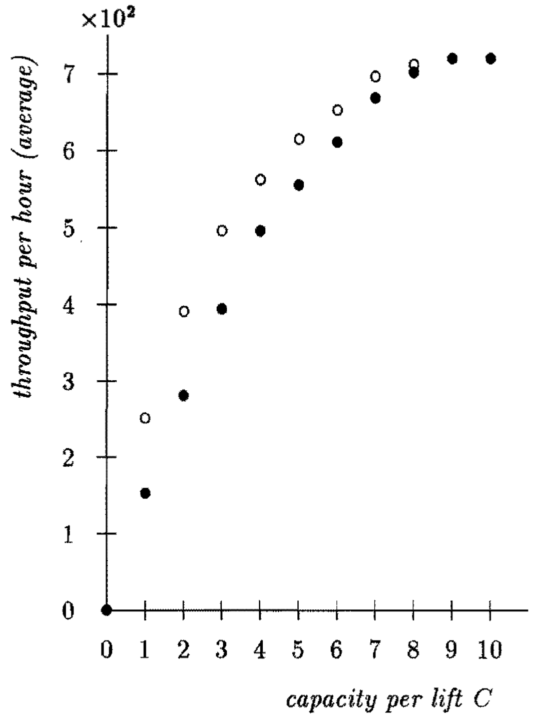


Figure 6.11: The influence of the watchdog timer, where the case without and with the watchdog timer are marked by 'o' and '•'. The number of lifts equals $M = 2$.

of the control. This means that the control instructs to move on and the lift has chosen a stop as its new direction. Apparently, there are no internal requests for this floor and, hence, the destination floors recorded in I are still valid. The only change involves the floor of the lift.

```
if $\bar{v}_3[k]$ $\rightarrow$ $v_3^\circ[k]$?
                ; if $L[k].dir = \triangle$ $\rightarrow$ $Control(U, D, I, L[k := (L[k].floor-1, \square)])$
                [] $L[k].dir = \nabla$ $\rightarrow$ $Control(U, D, I, L[k := (L[k].floor+1, \square)])$
                fi
```

As promised before, we conclude with the effect of the watchdog timer on the throughput of the system. The results concern the situation in which there are 2 lifts, and the other parameters are taken as before. Furthermore, the lifts require a new direction within $t = 0.1$ time units, and the time needed by the control is drawn from a uniform distribution over time interval (0..0.5). The simulation results are given in Figure 6.11. From the figures it is clear that for small capacities there is a considerable reduction in the throughput of the system. For larger capacities the penalty is minimal. The reason is that the lift had to stop anyway, because it had some capacity left and there were waiting people on that floor.

Chapter 7

Jump-cutted[1] real-time simulation

As described in Chapter 4, external choice can be implemented at the expense of an additional structure, but it requires some effort to do so. A totally different, near-effortless implementation method is described in this chapter; we introduce a sort of approximation technique by using a *real-time programming language.*

In the enabling model we use the idea of a conceptual global clock. The clock has been implemented in Chapter 4 with the use of local variables and extra communications to align the local times of processes when interacting. Another approach is to abandon the time updates and to presume the presence of a real global clock which shows the time that has elapsed after the simultaneous start of the execution of all processes. Hence, the notion of time is in fact the elapsed execution time. The execution of the processes proceeds conform the specification. A delay suspends the process for the specified amount of time. At first sight, this idea may seem rather strange: the execution of each instruction in the execution code consumes some processor time, which adds an erroneous contribution to the time. As a result of that, a discrepancy arises between the time in the specification and the time in the implementation. Apart from the errors, another problem faced with is the length of a simulation run: simulating a period of length X consumes an execution time of about X. For small X this may be acceptable, but when a reasonable simulation period has to be covered this results in an undesirably long execution time.

Notwithstanding the foregoing drawbacks, some real-time programming languages like Occam [20] and Transputer Pascal [25] come very close to our notion of programs and, therefore, we will try to exploit their use. Note that these languages contain an external-choice construct and the possibility to describe delays. In order to form an idea about the validity of the simulation outcomes, we have to study the influence of the introduced errors. A way to make the erroneous time consumptions relatively small is obtained by enlarging the time unit of the computation; this varying of the computation time is called *scaling.* However, a consequence of a larger time unit in the computation is an even longer execution time. In order to reduce the execution time, we introduce so-called *jump-cuts*: when no activity happens for a while, the

[1]Jump-cut: *Film* an abrupt change from one shot, scene, or sequence to another, caused by the absence of intermediate or transitional action, effects, etc. [Webster's New World Dictionary]

clock is advanced to the next moment of interest. This yields a kind of mixed approach: on the one hand we have the real-timebased execution, and on the other hand we have a sort of event approach.

In Section 7.1 we introduce the implementation method and indicate, without going into details, how it is realized on a single processor.

In Section 7.2 we discuss the validity of the outcomes that are obtained by this implementation technique. As a means to improve the validity of the results we introduce scaling of the computation time.

7.1 Real-time approach

The power of an event-based simulation lies in the fact that the execution sort of 'jumps' from event to event; the uninteresting periods between successive events are skipped. The simulation time is recorded in a variable and time updates consist of simple assignments to that 'time variable.' As a result the simulation will be fast, provided that determining the next event consumes only a minor amount of execution time. The latter is essential for this simulation technique.

Programming languages like Occam and Transputer Pascal include the notion of a clock. This stems from the presence of a clock in the hardware of the target processor, which is a Transputer [21, 22]. The clock has been added to support the applicability of the processor in real-time applications. If we are able to use this built-in clock for the time in a process, the gap between specification and implementation becomes small. The constructs in these programming languages come very close to our program constructs which yields the possibility of an almost literal translation. This also holds for external choices and delays. We do not point out the corresponding constructs of our programs in these languages, but note that similar constructs exist.

Instead of using a variable to encode the time in a simulation, we (ab)use the processor clock for this purpose. Similar to function τ in our specifications, we obtain a function which returns the processor time. The clock in a processor is started at the moment of initiating the execution code and shows the elapsed execution time. Note the difference between our conceptual global clock in the underlying model and the real-time processor clock. The conceptual clock is a set of time values on which actions can be mapped, whereas a processor clock is a physical device showing the elapsed execution time. Each action in the specification is encoded by a sequence of instructions, and the execution of each instruction requires a certain amount of processor time. As a consequence, the idea of timeless actions is unrealizable. Every time when the execution requires more time than the specified amount of time, an error is introduced. This error will be significant if computationally intensive tasks are involved. In fact, we want the processor to be infinitely fast, thereby causing the erroneous time consumptions to be negligible small.

Example 7.1
Consider the closed system which consists of processes S, T, and U. Process S contains an external-choice construct in which the guards are probed input channels a and b.

$$\begin{array}{ll} S = & \textbf{if}\ \bar{a} \rightarrow S_1 \\ & [\!]\ \bar{b} \rightarrow S_2 \\ & \textbf{fi} \end{array}$$

The behaviours of processes T and U are similar: process T performs computation X and afterwards communicates via channel a; process U does computation Y and next communicates via channel b. We assume that computations X and Y are free from delays and communications.

$$\begin{array}{l} T = X\ ;\ a^{\bullet}! \\ U = Y\ ;\ b^{\bullet}! \end{array}$$

Both computations X and Y consume processor time, say t_X and t_Y. The choice in S is determined by the length of t_X and t_Y. When we want to avoid this, we explicitly model the time consumptions of X and Y. Suppose we want X to consume x time and Y to consume y time. In order to achieve this, X and Y are replaced by X , $\delta(x)$ and Y , $\delta(y)$. For $t_X \leq x$ and $t_Y \leq y$, a valid choice in S is made.
□

Apart from the erroneous time consumptions, we are faced with another problem associated with this idea of real-timebased simulation. As mentioned before, simulating a period of length X requires an execution time of about X. Since the implementation will be used for many simulation runs, short execution times are needed. Unfortunately, the length of simulation periods tend to be long. In order to accelerate the simulation, we want to skip uninteresting periods: when certain conditions hold, the processor clock is advanced to the next moment of interest. To express the precondition of the advancement, some terminology is introduced. A process is called *idle* if it is suspended in a communication or if it is suspended due to a delay $\delta(M)$; otherwise we say that a process is *active*. As long as there are active processes, fruitful work is being done. In situations in which all processes are idle, at least one process is suspended in the execution of a delay; otherwise the computation would be in a deadlocking state, assuming that no processes terminate.

In fact, the periods in which all processes are idle are uninteresting: nothing real happens until the moment when one of them regains activity. We skip these periods from the execution by performing so-called *jump-cuts*: when all processes are idle, the clock is advanced to the next moment upon which a process suspended in a delay regains activity. Note that for simulations in which all processes are idle most of the time, the speedup obtained by this advancement scheme will be significant.

Verifying the advancement condition of the clock requires the state of all processes and, in case of all processes being idle, the next moment when activity is regained. In the absence of a sort of shared bookkeeping, acquiring all the necessary information from a collection of processes is difficult. Another problem may stem from a possible hardware deficiency: each processor has its own local clock and there is no global one available. For the time being, suppose that we have easy access to the required information and that there is a global clock present. We assume boolean function *AllIdle* to denote the overall idle situation. In case of *AllIdle* $\equiv$ true, function *NextActiveMoment* gives the next moment when a process becomes active. In order to advance the processor clock, which we denote by τ, we introduce process *Arbiter*. Under the assumption that there is always sufficient time left to determine the outcome of the functions and to advance the clock, the specification of process *Arbiter* is:

```
Arbiter = if ¬AllIdle
              → Arbiter
          [] AllIdle
              → if NextActiveMoment = ∞  → ε
                [] NextActiveMoment < ∞  → τ := NextActiveMoment
                                           ; Arbiter
                fi
          fi
```

In theory, this looks all very nice but we still have to come up with a practical realization. There is, however, one situation in which the required information is easily obtained, namely in the case of a single-processor implementation executing the processes is a time-sliced manner. A single Transputer provides a global clock and easy access to the state of the processes. Without going into too much detail, we mention that a Transputer is equipped with a process list, a timer list, and an adjustable clock. The process list contains the processes whose executions may be scheduled after a deschedule of the current one being executed; we say that these processes are *queued.* The timer list contains the idle processes that are suspended because of a delay. Once the delay has elapsed, the process is added to the process list and its state becomes 'queued.' A simple inspection of the relevant list suffices to determine the outcomes of functions *AllIdle* and *NextActiveTime*. The maintenance of the process and timer list is done in hardware and happens, therefore, very fast. An overview of the state transitions of a process is given in Figure 7.1, where the idle state is split into *pending* and *delayed*; they denote the suspension caused by a communication and a delay respectively. The initial state of processes is set to 'queued.'

Although a single-processor implementation enables easy access to the required information, the newly introduced state 'queued' causes another error because of the time spent in the process list. Note that the effect on the processor time is negligible small if the processor is infinitely fast and, consequently, the time spent in the list may be disregarded. Another remark concerns the idle time of the processes: when all processes are idle most of the time, the execution happens in an almost parallel way.

Accelerating the simulation requires the addition of process *Arbiter* only. Nevertheless, it would be nice if the hardware could be set in such a way that it takes care of the time advancements, as soon as the advancement condition holds.

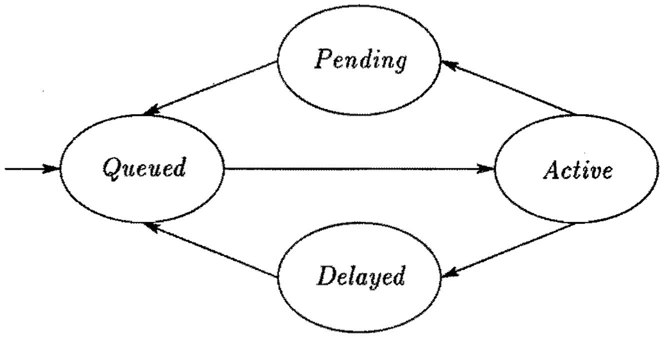


Figure 7.1: State diagram of the execution of a process.

7.2 Validating the results

Due to the errors described, care should be taken when interpreting the simulation outcomes acquired. In fact, the results are useless if we have no idea about the induced effect; hence, we have to determine the validity of the outcomes, which needs to be simple.

The size of the errors depends on the processor speed: for an infinitely fast processor the errors are infinitely small. An infinitely fast processor is, however, purely hypothetical; a normal processor runs at a fixed finite speed. For this reason, we need another way to obtain an imaginary faster execution device. This is achieved by means of *scaling* the time unit in the computation: scaling-up yields a larger time unit, and scaling-down results in a smaller time unit. When the simulation outcomes do not change significantly after a reasonable increase of the time unit then we may neglect the errors. Otherwise a further increase of the time unit is needed to validate the last-obtained results. Note that scaling-up the computation results in an increase of the percentage of time in which all processes are idle. Furthermore, scaling the computation time does not affect the execution time of the simulation.

In order to illustrate the effect of scaling, we consider system *Factory* which is given by:

$$Factory = \{Producer, Worker, Consumer\}$$

The processes of the system are specified as follows.

$$\begin{aligned}
Producer &= a^{\circ}!prod \ ; \ Producer \\
Worker &= a^{\bullet}?prod \ ; \ \delta(t) \ ; \ b^{\bullet}!prod \ ; \ Worker \\
Consumer &= b^{\circ}?prod \ ; \ Consumer
\end{aligned}$$

Suppose we are interested in the time that elapses between successive a and b communications. Due to the delay of size t and an erroneous overhead of size say β, we find that the elapsed time equals $t + \beta$. As a result of scaling the time in the computation with factor α, the elapsed time becomes $\alpha * t + \beta$. Since the processor overhead remains the same, scaling back to the original time size yields $t + (\beta/\alpha)$. For sufficiently large α, the effect of the errors becomes negligible small and converges to 0. Note that the scale factor is limited when the clock can reach a maximum value: for maximum time value *maxTime* and simulation period *simTime*, the factor α is at most *maxTime*/*simTime*.

According to the jump-cutted principle, we have made an implementation of system *Factory*, with delay size $t = 1$. The average of the measured t values is shown in Figure 7.2, as a function of scale factor α. Apparently, for this system the overhead β is about 1 as well.

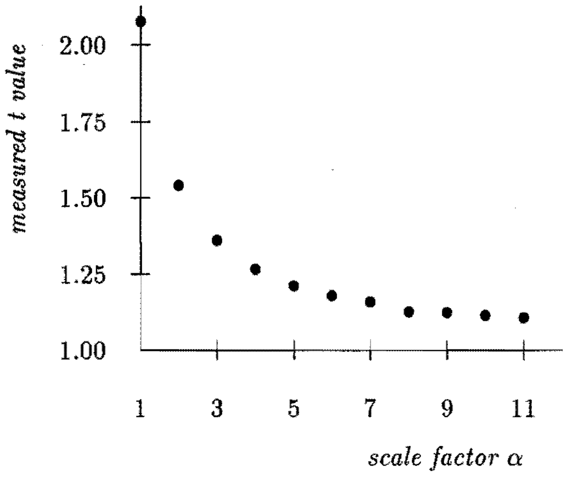


Figure 7.2: The measured t values, for different scale factors α.

In order to form an idea about the speed of the simulation, we have counted the number of jump-cuts that are performed during a certain interval. For the current example, the counted (average) number is 1.78×10^4 per second of execution time.

Chapter 8

A job-shop factory

In the introductory chapter we mentioned the rough classification of industrial systems into flow-shop and job-shop factories. Both types are illustrated by an example: Chapter 5 contains the description of a flow-shop factory, and in this chapter we address the problem of modelling a job-shop factory. After the development of a rather general specification of the job shop, we study the performance of a specific instance. The implementation is obtained by applying the jump-cutted real-time approach described in the previous chapter. At the end of this chapter, we study the realized number of jump-cuts per second of execution time.

Just like a flow-shop factory, a job-shop factory consists of a number of shops. Each shop can perform a certain operation or processing step on a product. Several shops may be of the same type, which means that they perform the same operation. Unlike a flow shop, a job shop has no imposed process layout. The execution path of a product depends on the specific treatment required and the presence of other products in the shops. Instead, the layout is based on the function of the shops: shops that perform the same processing step are grouped together. In order to exploit the full production capacity of such factories, the control of the variable product flow is usually based on a complex planning strategy. For reasons of simplicity, the strategies we look at are kept relatively simple.

Not all products require the same processing. Therefore, each product is accompanied by a so-called *job* which gives the necessary information about the processing requirements of the product. The job is passed to the control and serves as an indispensable ingredient to determine the route through the system. For more complex assemblages, for example those in which the end products are composed of a number of semi-finished input products, the jobs are issued separately and also state the input products needed. A typical example of a job-shop factory is an IC-manufacturing system, where the wafers are processed in various shops. Detailed descriptions of such systems are given in [35, 38].

Section 8.1 contains some preliminary observations of the job-shop factory. To mention a few, the product flow through the system is illustrated, and there is a concise explanation of the function of the processes in the system.

In Section 8.2 we present a complete overview of the system and the environment, which includes the information flow. The specifications of all processes but the control of the factory are discussed.

In Section 8.3 the control process is described and the initial configuration of the system and the environment is given.

In Section 8.4 we consider a specific instance of the closed system and study the relevant performance characteristics which in this case are throughput and latency.

In Section 8.5 we add a maximally cooperative environment and perform some feasibility studies: other control strategies are tried out, and the effect of a larger routing capacity is examined.

8.1 Preliminaries

In the job-shop factory we assume the presence of N shops, $N \geq 1$, which are numbered from 1 to N. Since we do not allow parallel processing of several products in the same shop, a shop processes at most one product at a time. We also abandon simultaneous processing of the same product by different shops. The types of the shops are represented by integers; the set of all possible types is given by $[1..T]$, with T denoting the number of different shop types. The type of a shop remains the same: it is impossible to put a shop in a different mode of operation. We assume that function $\kappa \in [1..N] \rightarrow [1..T]$ gives the type of the shops. Furthermore, we assume that each type is represented in the factory. As a consequence, the number of shop types is at most the number of shops, $T \leq N$, and function κ is surjective:

$$(\forall t : 1 \leq t \leq T : (\exists j : 1 \leq j \leq N : \kappa(j) = t\,)\,)$$

The required treatment of a product is given by a sequence of shop types. The types of the successive shops visited by a product must satisfy the order in its treatment. Instead of a single sequence, a more liberal form is given by a set of sequences that all yield the desired end product; though, for simplicity's sake, we will not do so. As a result, treatments are static. All relevant treatments are captured in a recipe book $R \in [1..M] \rightarrow [1..T]^*$, with M denoting the number of entries, $M \geq 1$. Each product that enters the system carries a label which shows the entry in the recipe book, thereby indicating the desired treatment indirectly.

In order to form an idea about the processing in the job-shop factory, an overview of the product flow is given in Figure 8.1. Products enter the system via input pile *PileIn* which stacks the products and jobs that are supplied by the environment. The supply robot *Robot*$_1$ transports the products from the input pile to a one-place buffer *Buffer*$[i]$, with $1 \leq i \leq B$ and B the number of these buffers. Once a product has been put in a buffer, it is a candidate to be processed. The various operations are executed in the available shops *Shop*$[j]$, $1 \leq j \leq N$. When all required processing steps have been completed, products leave the system via output pile *PileOut*. The

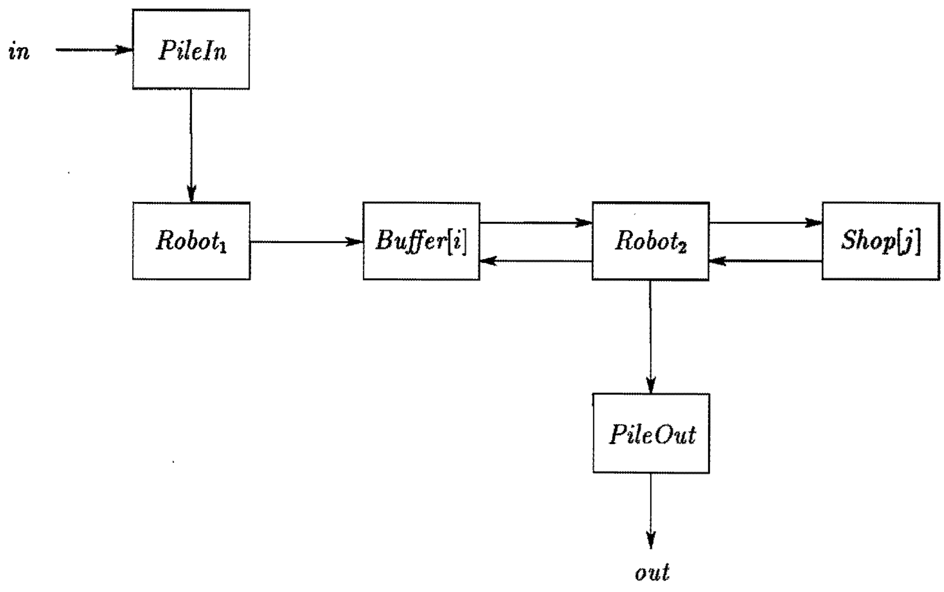


Figure 8.1: The product flow in the job shop factory, with $1 \leq i \leq B$ and $1 \leq j \leq N$.

product transport between buffers, shops, and output pile is accomplished by another robot, the distribute robot *Robot*$_2$. Both *Robot*$_1$ and *Robot*$_2$ are capable of transporting at most one product at a time.

According to its treatment, a product is moved between various shops. The processing of the shops is mutually independent and, in order to enlarge the throughput of the system, several products can be present in the shops. A shop consists of a single work station and has no additional buffering; hence, each shop contains at most one product. After process completion, the processed product is removed from the shop and, if possible, transported to a next free shop of suitable type. When there are no free successors, we could choose to leave the product in the shop until a successor becomes available. However, without any further restrictions on the possible treatments, a so-called deadlocking state is likely to occur. For example, consider a factory that contains 2 shops, and the set of possible recipes is $\{12, 21\}$. When shop 1 processes a product with treatment 12 and shop 2 processes a product with treatment 21, a deadlock will arise. In order to avoid deadlocks, we restrict the number of products in the buffers, *Robot*$_2$, and shops to the number of buffers B. Then, it is possible to assign to each product an input buffer of its own. After the completion of a processing step, and when all shops capable of performing the next step are occupied by other products, the product can be put back in its buffer, thereby releasing the shop where it resides. Obviously, in order to achieve the most favourable performance, a parameter to be determined is the number of buffers B to engage.

8.2 The processes in the system

In order to regulate the product flow, there is also information flow in the system. The channels introduced to accomplish the information flow are connected with the control process. A complete picture of the system and the environment is given in Figure 8.2, in which the removal of products from the output pile has been omitted.

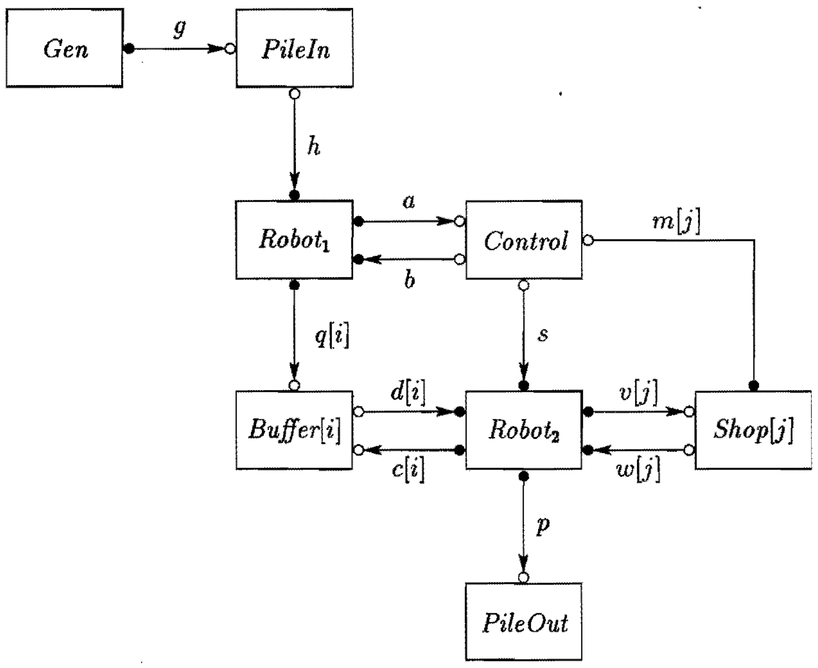


Figure 8.2: An overview of the job shop factory and environment.

$Robot_1$ needs to know the destination buffer of new products that enter the system. For that reason, it is equipped with channel b of type $[1..B]$. We assume that *$Robot_1$* is capable of reading the entry on the product label; the outcome is reported to the control via channel a of type $[1..M]$. When the control receives a communication request via channel s, it informs *$Robot_2$* about the source and destination of a product move. In order to report to the control the completion of the processing of a product, shops use the m channel of type Signal.

As we have done before, we assume suitable product type *Products.* All product flow, except for channels g and h, happens via channels of type *Products.* For communications via channels g and h, the product label is also part of the communication. As soon as the product has been put in its buffer, the label is no longer needed and therefore omitted. The time taken by all interactions in which a product is involved is explicitly modelled by adding a delay of size T_I. For time-consuming interactions with a fixed delay, we will put the delay after the communication. As a result of that, the communication denotes the beginning of the interaction. We continue with the specification of all processes but the control; the specification of the control process is given in the next section.

The input pile has an infinite capacity and maintains list L of products and their entries. New products enter the input pile via channel g and leave via channel h. Since the label is part of these communications, the type of the channels is *Products* $\times$ $[1..M]$. The behaviour of the pile consists of a choice between these channels, where answering a request from h requires L to be non-empty.

```
PileIn ∈ (Products×[1..M])* → Π, with
PileIn(L) = if $\bar{g}$ → $g^\circ?(prod, entry)$ ; $\delta(T_I)$
                          ; $PileIn((prod, entry)L)$
            [] $\bar{h} \wedge L \neq \epsilon$ → $h^\circ!hd(L)$ ; $\delta(T_I)$
                                 ; $PileIn(tl(L))$
            fi
```

When the last processing step of a product has been completed, it is transported to the output pile. The output pile has input channel p from which it accepts the completely processed products. Since we do not model the removal of products from the output pile, it repeatedly reads its p channel.

```
PileOut = $p^\circ?prod$ ; $\delta(T_I)$
        ; PileOut
```

Both robots interact with the buffers: *Robot*$_1$ supplies new products, and *Robot*$_2$ moves the products between buffers and shops. The operation of the buffers is described by their generic representative which alternates between inputs and outputs consuming T_I time. Initially, we assume that the buffer is empty and, hence, the buffer is willing to read one of its input channels.

```
Buffer = if $\bar{q}$ → $q^\circ?prod$
         [] $\bar{c}$ → $c^\circ?prod$
         fi
       ; $\delta(T_I)$
       ; $d^\circ!prod$ ; $\delta(T_I)$
       ; Buffer
```

A shop starts its processing cycle with the receipt of a product via v. The processing of a product is described by a delay of size T_P. When processing has been completed, a signal is sent via channel m. Eventually, *Robot*$_2$ reports its presence and the product is handed over via w which completes the processing cycle. Note that we restrict ourselves to treatments in which successive steps require different shop types.

$$\begin{aligned} \mathit{Shop} = \;& v^\circ?\mathit{prod} \;;\; \delta(T_I) \\ & ;\; \delta(T_P) \\ & ;\; m^\bullet! \\ & ;\; w^\circ!\mathit{prod} \;;\; \delta(T_I) \\ & ;\; \mathit{Shop} \end{aligned}$$

The initial location of *Robot*$_1$ is at the input pile, and the robot is suspended in an interaction to obtain a labelled product from the pile. When the interaction has been completed, the robot needs to know the destination buffer before the move to the buffer location can start. Therefore, the control supplies via channel b the destination of the newly-obtained product. The subsequent move of the robot is described by a delay of size T_M. When the product has been put in its buffer, the robot reports to the control the entry read on the attached label; this is accomplished by a communication via channel a. Simultaneously with the a communication, the robot moves back to its initial position at the input pile. Note that the a communication happens when the interaction with the buffer has been completed and, hence, the communication indicates that the product can be moved to a shop.

$$\begin{aligned} \mathit{Robot}_1 = \;& h^\bullet?(\mathit{prod}, \mathit{entry}) \;;\; \delta(T_I) \\ & ;\; b^\bullet?i \\ & ;\; \delta(T_M) \\ & ;\; q^\bullet[i]!\mathit{prod} \;;\; \delta(T_I) \\ & ;\; a^\bullet!\mathit{entry} \;,\; \delta(T_M) \\ & ;\; \mathit{Robot}_1 \end{aligned}$$

The task of the distribute robot is to take care of the transport of products between buffers, shops and output pile. For each transport, the robot is told where to pick up the product and where to drop it. In order to denote both source and destination of a move, we assign to each process a distinct address: buffer i is given address i, shop j is given address $B + j$, and the output pile has address $B + N + 1$. On basis of a 'source-destination' pair

$$(\mathit{src}, \mathit{dest}) \in [1..B + N] \times [1..B + N + 1]$$

the robot can determine the moves to be made. When the transport job has been received, the robot moves from its current position to the source location and picks up the product. Afterwards, the robot moves towards the destination and drops the product in the right position. The time of a move between two addresses depends on the location of source and destination. To determine the amount of time consumed, we assume the presence of function *transpTime*,

$$\mathit{transpTime} \in [1..B + N + 1] \times [1..B + N + 1] \;\to\; \mathcal{T}$$

The location of the distribute robot is recorded in variable *loc* of type $[1..B + N + 1]$. After a product transport, the current location of the robot equals the address of the process where it resides. Initially, the robot is suspended in the receipt of a new transport job from the control.

```
$Robot_2 : \{1..B + M + 1\} \rightarrow \Pi$, with
$Robot_2(loc) = s^{\bullet}?(src, dest)$
                ; $\delta(transpTime(loc, src))$
                ; if $1 \leq src \leq B \rightarrow d^{\bullet}[src]?prod$
                  $[]$ $B < src \leq B + N \rightarrow w^{\bullet}[src]?prod$
                  fi
                ; $\delta(T_I + transpTime(src, dest))$
                ; if $1 \leq dest \leq B \rightarrow c^{\bullet}[dest]!prod$
                  $[]$ $B < dest \leq B + N \rightarrow v^{\bullet}[dest]!prod$
                  $[]$ $dest = B + N + 1 \rightarrow p^{\bullet}!prod$
                  fi
                ; $\delta(T_I)$
                ; $Robot_2(dest)$
```

8.3 Control process and resulting system

Thus far, we have encountered already a few basic control requirements: a shop reports its process completion; the distribute robot asks for a new move; and the control should regulate the entrance of new products. With respect to a new move, the destination process must be willing to accept a product and, when the destination process is a shop, it must be suitable to execute the next step in the treatment of the product. The reaction of the control process depends on the current situation in the factory. Before we discuss the possible reactions of the control, we describe the variables in which we record the relevant information. The control unit is described by process $Control(E, F, H, L, P, W)$, which consists of a choice on the possible inputs.

8.3.1 The variables of the control process

A product, which has been assigned to a buffer, claims the buffer as long as there is at least one processing step to be done. As a result, when all shops capable of performing the next processing step are occupied by other products, the product in a shop can be brought back to its buffer, thereby releasing the shop. We name the products that are in the buffers, in the shops, and in $Robot_2$ by the address of their buffer place. The claimed buffers are described by set P of type $\mathcal{P}([1..B])$.

The entries received via channel a are recorded in array E of type $[1..B] \rightarrow [1..M]$, with $E[i]$ denoting the entry of the product that has claimed buffer i most recently. For product $i \in P$, application $R(E[i])$ yields the required treatment. Variable E describes a partial function: its application makes sense only for products in P.

The next processing step of a product depends on the ones that have been executed already. For that reason, we introduce array H of type $[1..B] \rightarrow [1..T]^*$ to keep track of the completed steps. Function H is also a partial function which is applicable to products in P only. For product $i \in P$, the sequence of executed steps $H[i]$ is a prefix of the whole treatment $R(E[i])$:

$$(\forall i : i \in P : (\exists u : u \in [1..T]^* : H[i]u = R(E[i])))$$

Hence, with the use of $H[i]$ and $R(E[i])$ we are able to determine the next processing step of product $i \in P$, namely $hd(u)$ if u is non-empty and ϵ otherwise.

A shop is either busy or free. We say that a shop is free when it is suspended in the receipt of a product; otherwise a shop is said to be busy. Hence, an interaction with $Robot_2$ changes the state of a shop. We record the free shops in set F of type $\mathcal{P}([1..N])$. The products in busy shops are recorded in array W of type $[B+1..B+N] \to [1..B]$, where application $W[B+j]$ requires $j \notin F$. Note that, because of displacement B in the domain, W applies to addresses of busy shops. Obviously, the range of W is a subset of P. The next shop type required in the treatment of the product in shop j, with address $B+j$, is determined from $H[W[B+j]]$ and $R(E[W[B+j]])$.

In order to be moved to somewhere else, the products in the buffers and shops need the assistance of $Robot_2$. To obtain a fair scheduling scheme in the assignments of this robot, we introduce list L of type $[1..B+N]^*$, which contains the addresses of the processes with a transportable product. We will see to it that new requests are added to the end of the list, on account of which the longest waiter is considered first when a new transport job has to be determined. As an invariant property of list L we maintain the uniqueness of its elements:

$$(\forall a,u,v : L = uav \wedge 1 \leq a \leq B+N : \neg(\exists x,y : x \neq u : L = xay))$$

The domain of the control process is now straightforward: it consists of the Cartesian product of the types of the variables. The description of the reactions on the possible inputs is given next.

8.3.2 The possible reactions

As we have mentioned before, shop j reports process completion via channel $m[j]$. These messages have to be answered by $Robot_2$ and are therefore added to list L. In order to update the processing information of product $W[j]$ in shop j, the sequence of executed treatments $H[W[j]]$ is extended with step $\kappa(j)$.

```
if $j : 1 \leq j \leq N : \bar{m}[j]$
  $\to m^{\circ}[j]?$
    $; Control(E, F, H[W[B+j]] := H[W[B+j]]\kappa(j)], L(B+j), P, W)$
```

Note that the shop remains busy; it is set to free only when the processed product is taken away by $Robot_2$.

New products are allowed to enter the system when there is a free buffer. This condition holds if there is a buffer i, $1 \leq i \leq B$, which has not been claimed, $i \notin P$. If so, a request from $Robot_1$ is answered by sending via b the address of an arbitrary free buffer. After the move and subsequent interaction with the destination buffer, $Robot_1$ reports via a the entry leading to the required treatment in the recipe book. The consequences on the variables are: the selected buffer is claimed; the sequence of completed treatments of the new product is set to empty; and the entry is recorded. The reaction of the control could be as follows:

```
if $i : 1 \leq i \leq B \wedge i \notin P : \bar{b}$
  → $b^\circ!i$
    ; $a^\circ?v$
    ; $Control(E[i := v], F, H[i := \epsilon], Li, P \cup \{i\}, W)$
```

The move of a product from the input pile to its buffer takes T_M time. As a result, adopting the above reaction would suspend the control between the b and a communication for T_M time. However, a suspended control process is undesirable because as a side effect the shops and distribute robot may become suspended as well. In a sense, the environment has too much influence on the control, which is avoided by handling the a and b communications separately. For the b communications we take:

```
if $i : 1 \leq i \leq B \wedge i \notin P : \bar{b}$
  → $b^\circ!i$
    ; $Control(E, F, H, L, P, W)$
```

In fact, we need an extra variable to record the selected buffer i. We are, however, a bit sloppy, and leave the selected buffer implicit; we simply use i in the reaction on the subsequent a communication. As we have seen, the a communication indicates the completion of the move, and the relevant updates may happen.

```
if $\bar{a}$ → $a^\circ?v$
         ; $Control(E[i := v], F, H[i := \epsilon], Li, P \cup \{i\}, W)$
```

The control of the distribute robot requires some more effort. Upon a request from *Robot*$_2$, the control has to determine whether or not there is a move to be done. If so, the control determines, according to a certain control strategy, a new transport job. A processed product that resides in a shop is always transportable: either it is put in its buffer or it is moved to a shop that can perform the next step. Moving a product from its buffer to a shop requires the shop to be of suitable type. In order to formalize these conditions, we define for product $i \in P$ the next treatment $suc(i)$, with $suc(i) \in [1..T] \cup \{\epsilon\}$, as follows:

```
$suc(i) =$ if $H[i] = R(E[i])$ → $\epsilon$
           [] $t, u : 1 \leq t \leq T \wedge u \in [1..T]^* : H[i]tu = R(E[i])$ → $t$
           fi
```

The addresses of all processes that have a movable product are recorded in list L. A formal statement of the phrase 'there is a transportable product in one of the processes' is given by expression C:

$$\begin{aligned} C \equiv\ & (\exists l : l \in L : B < l \leq B{+}N) \\ & \vee (\exists l : l \in L \wedge 1 \leq l \leq B : (\exists j : j \in F : \kappa(j) = suc(l))) \end{aligned}$$

When condition C holds, we can apply control strategy f to select a source address from list L. For the time being, we assume that f yields a proper source-destination pair in range $[1..B + N] \times [1..B + N + 1]$. A description of f is given in the next section.

For lists L, with $L = uxv$ and $1 \le x \le B + N$, we define $L\backslash x = uv$. Note that such a uv is uniquely defined, because each element appears at most once in L. Using condition C and control strategy f, we specify the reaction of the control process on a request from *Robot*$_2$:

```
if $\bar{s} \wedge C$
   $\rightarrow$ $(x, y) := f$
   ; $s^\circ!(x, y)$
   ; if $1 \le x \le B$
        $\rightarrow$ $Control(E, F\backslash\{y - B\}, H, L\backslash x, P, W[y := x])$
     $[\!]$ $B < x \le B + N$
        $\rightarrow$ $x' := x - B$
          ; if $1 \le y \le B$
               $\rightarrow$ $Control(E, F \cup \{x'\}, H, (L\backslash x)y, P, W)$
            $[\!]$ $B < y \le B + N$
               $\rightarrow$ $Control(E, (F \cup \{x'\})\backslash\{y - B\}, H, L\backslash x, P, W[y := W[x]])$
            $[\!]$ $y = B + N + 1$
               $\rightarrow$ $Control(E, H, F \cup \{x'\}, L\backslash x, P\backslash\{W[x]\}, W)$
            fi
     fi
```

Note that a completely processed product is never put back in its buffer; it is transported from the shop that performed the last step to the output pile. As a result, for the product in buffer i, $suc(i)$ always yields a shop type and never ϵ.

8.3.3 A first strategy

A simple control strategy, described by f_1, selects pair (x, y), with x the first element in L whose product can be moved; y is a possible free destination. When x is a buffer address, the destination is a free shop of suitable type. For x a shop address, several possibilities have to be taken into account. When the product in the shop has been processed completely, it is moved to the output pile. Otherwise, if there is a suitable shop available the product is moved to such a shop, and else the product is put back into its buffer. Given that condition C holds, we define $f_1(xu)$, for address x, $1 \le x \le B + N$, and sequence of addresses u, $u \in [1..B + N]^*$, as follows.

$$\begin{cases} f_1(xu) = f_1(u) & \text{if } 1 \le x \le B \wedge \neg(\exists j : j \in F : \kappa(j) = suc(x)) \\ f_1(xu) = (x, y) & \text{otherwise, with } y \text{ defined by:} \end{cases}$$

```
if $j : j \in F \wedge 1 \le x \le B : \kappa(j) = suc(x)$
   $\rightarrow$ $y = B + j$
$[\!]$ $B < x \le B + N$
   $\rightarrow$ if $suc(W[x]) = \epsilon$
          $\rightarrow$ $y = B + N + 1$
      $[\!]$ $j : j \in F \wedge \kappa(j) = suc(W[x])$
          $\rightarrow$ $y = B + j$
```

```
        [] $(\forall j : j \in F : \kappa(j) \neq suc(W[x])) \wedge suc(W[x]) \neq \epsilon$
          $\rightarrow\ y = W[x]$
        fi
    fi
```

Note that the choice of a free shop is non-deterministic. Several factors can be used to influence the actual choice made, for example by choosing one with minimal transport time.

8.3.4 The system

The description of the factory is closed by adding the generator of new products and jobs. For the creation of products and jobs we assume the presence of function *newProd*, $newProd \in Products \times \{1..M\}$. The time between two successive arrivals is given by distribution function $\mathcal{D}_G \in \mathcal{T}$. In order to record the next moment when a new product has to be delivered, we parameterize the specification with variable t, $t \in \mathcal{T}$.

$$\begin{aligned}&Generator : \mathcal{T} \rightarrow \Pi, \text{ with}\\ &Generator(t) = \delta((t-\tau) \max 0)\\ &\qquad ;\ g^{\bullet}!newProd\ ;\ \delta(T_I)\\ &\qquad ;\ Generator(t + \mathcal{D}_G)\end{aligned}$$

The specification of the whole system requires an instantiation of the parameterized programs, which describes the initial situation. We choose the following configuration. The input pile is initially empty and, concerning the distribute robot, we assume that it resides at the output pile having address $B + N + 1$. With respect to the environment, the first labelled product is generated at moment 0. Next we choose the instantiation of the control process. In the initial state of the factory there are no products present, hence $P = \emptyset$. As a consequence, the initial value of the entries in E and partial treatments in H is an arbitrary value from their range, which is denoted by symbol '@'. Furthermore, the absence of products yields a situation in which all shops are free, $F = [1..N]$, and the initial values in W do not matter either. Yet another consequence of the absence of products is the emptiness of list L. As a result we have:

$$\begin{aligned}&\{Generator(0)\}\\ \cup\ &\{Control([1..B] \rightarrow @, [1..N], [1..B] \rightarrow @, \epsilon, \emptyset, [B+1..B+N] \rightarrow @)\}\\ \cup\ &\{Robot_1, Robot_2(B+N+1)\}\\ \cup\ &\{PileIn(\epsilon), PileOut\}\\ \cup\ &\{i : 1 \leq i \leq N : Shop[i]\}\\ \cup\ &\{i : 1 \leq i \leq B : Buffer[i]\}\end{aligned}$$

8.4 A concrete instance

In the remaining sections we study a concrete instance of the factory. An overview of the instance we consider is given in Figure 8.3. The buffers, shops, and output pile are put in a row, in this specific order from left to right, with the addresses forming an increasing sequence. The input pile is put apart, in such a way that the distances to all buffer places are about equal. The areas in which the robots move are indicated by the arrows in the picture. We take 4 shop types, and each type is represented by 2 shops; hence, $T = 4$ and $N = 8$. In the row, the shop types form an ascending sequence, when read from left to right.

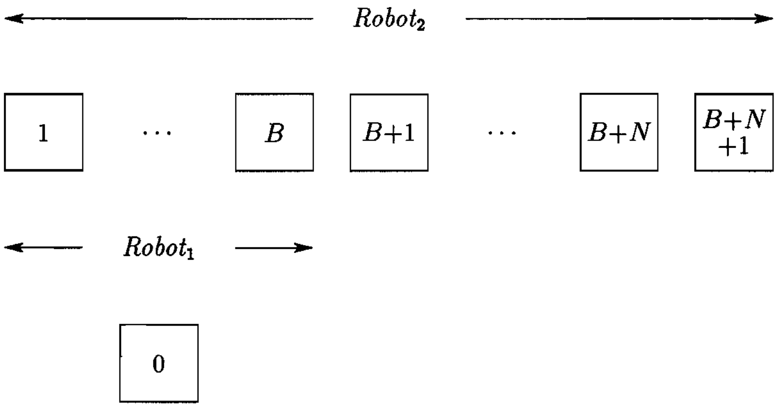


Figure 8.3: An overview of the instance, with N and B still to be determined. The input pile has address 0, the buffers have addresses 1 to B, the shops have addresses $B + 1$ to $B + N$, and the output pile has address $B + N + 1$.

The possible treatments of this job-shop configuration are sequences from $[1..4]^*$. However, not all possibilities are allowed; we restrict the possible treatments to the subclass which contains the sequences that are: non-empty, each type appears at most once, and the types are in increasing order. This yields a recipe book with its number of entries M equal to $2^4 - 1 = 15$. Note that for this restricted set of treatments the buffer places are not really needed. Because of the increasing order in the treatments, a deadlock as described in the beginning of this chapter does not happen when the products stay in the shops until a next one becomes free. Later on, in Section 8.5.1, we study the performance of the system when controlled by a strategy in which products stay in the shops and are not brought back to their buffer. We first study strategy f_1 because putting partially processed products back in their buffer releases the shops and, if the moves happen fast, the performance may benefit.

Several timing parameters and distribution functions are still to be chosen. New jobs arrive according to a Poisson process, where the average time between two successive arrivals is 60. The entries in the recipe book are randomly taken from [1..15]. Interactions in which a product is involved take $T_I = 1$, and *Robot*$_1$ needs $T_M = 5$ to travel from the input pile to a buffer or vice versa. The processing time of each shop type is $T_P = 60$. *Robot*$_2$ requires 1 time unit to move between two successive processes in the row, which yields:

$$(\forall i, j : 1 \leq i, j \leq B + N + 1 : \mathit{transpTime}(i, j) = |i - j|)$$

The non-determinism in strategy f_1, the choice of a free shop, is implemented by choosing the one with minimum address. As a result, products are transported to the nearest shop of suitable type. The reaction of the control process upon new products entails the selection of a free buffer place; in the implementation the one with maximum address is chosen.

With the use of the implementation we determine the average throughput per $24 * 3600$ time units (called a day) and the average latency, both as a function of the number of buffers B. The throughput is derived from the number of products that arrive at the output pile per interval, and the latency is the time that elapses between the removal from the input pile and the arrival at the output pile. Since we need the elapsed time for each product, type *Products* has been implemented by time-domain $\mathcal{T}$.

The simulation outcomes are depicted in Figure 8.4. From the results follows that for B in [3..12] the throughput is maximal. In fact, the throughput equals the number of jobs that arrive in the same period, namely $60 * 24 = 1440$. As a consequence, for these B values there is no accumulation of jobs in the input pile. When we take 13 or more buffer places, the throughput becomes less. The presence of more products in the system causes a higher occupation degree of the shops and, as a result, products are moved back to their buffer places more often. Furthermore, the travel times of *Robot*$_2$ depend on the locations of the processes: more buffers yield a higher average transport time.

In Figure 8.4 we see that when increasing the value of B, the latency decreases untill B is 4 and afterwards it increases. An explanation can be found in the introduction of more parallelism in the operation of the factory. For $B = 1$, a new product can be taken by *Robot*$_1$ only when the previous one has been moved to the output pile. When there are more products allowed, the average time a product spends in the supply robot becomes less. Due to the same reasons that cause the throughput to decrease, a further increase in the number of simultaneously processed products causes the latency to increase.

Selecting an appropriate B value on the basis of these results is difficult: with respect to the throughput, there is a whole range left to choose from. In fact, this also holds for the latency.

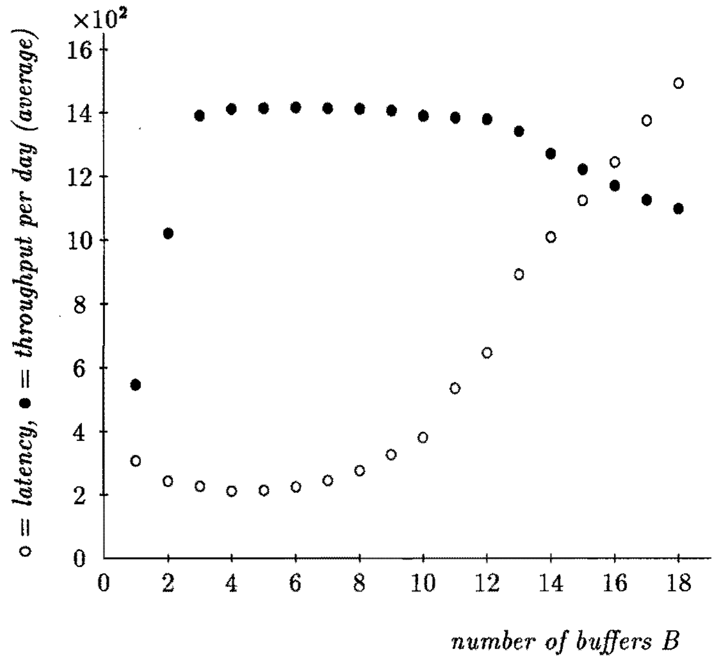


Figure 8.4: The simulation results of the instance, where the average throughput is denoted by '•' and the average latency by 'o'.

8.5 Feasibility studies

In this section we perform some feasibility studies of the instance described above. We determine the production capacity and study the effects of other strategies. Finally, we update the system in such a way that more distribute robots are allowed, by which a larger transport capacity among shops and buffers is obtained.

8.5.1 The current capacity

In order to determine the maximum throughput of the system, we need the presence of at least one job in the input pile when the supply robot asks for one. For that reason, we remove the *Generator* and take an input pile that has an infinite number of jobs available. The updates needed speak for themselves.

The simulation results are shown in Figure 8.5. It is obvious that the optimum throughput is reached for $B = 6$. For B in [5..12], we find an increase in the latency when compared to the previous results. Since the input pile is always capable of delivering a new job, there are more products present in the factory, which causes the latency to increase.

As discussed above, instead of moving products back to their buffer, this class of treatments permits a strategy in which the products stay in their shop until a

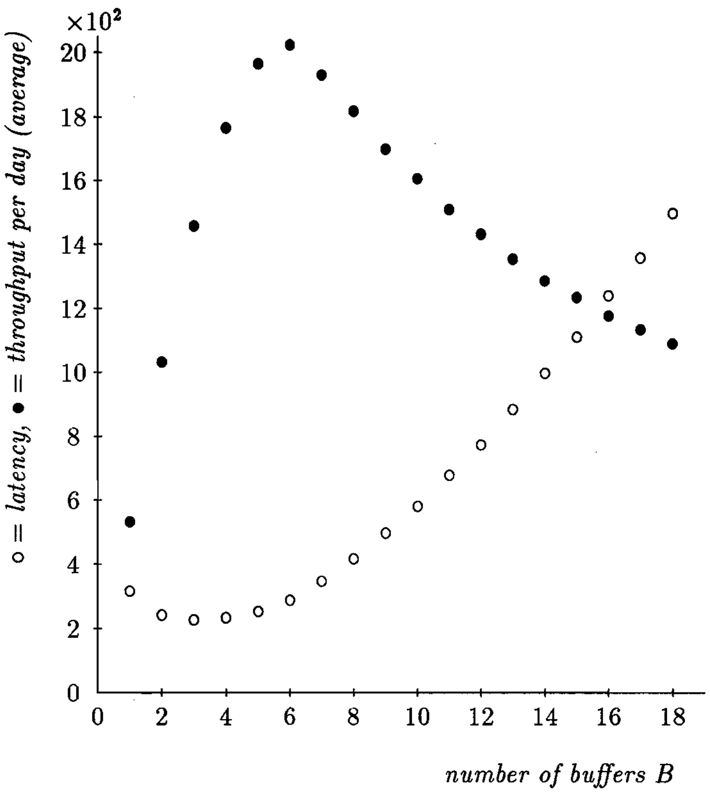


Figure 8.5: The performance characteristics of the job shop factory, with a never-empty input pile.

next shop becomes free. This strategy, which is called f_2, requires a different work condition, say C', because a processed product is moved only when there is a suitable shop free to perform the next processing step. Nothing has to change for products in the buffers.

$$\begin{aligned} C' \equiv\ & (\exists l : l \in L \wedge 1 \le l \le B : (\exists j : j \in F : \kappa(j) = suc(l))) \\ & \vee (\exists l : l \in L \wedge B < l \le B{+}N \\ & \quad : suc(W[l]) = \epsilon \vee (\exists j : j \in F : \kappa(j) = suc(W[l]))) \end{aligned}$$

Strategy f_2 searches for the first address in L whose process has a movable product. Given that condition C' holds, we define $f_2(xu)$, for address x and sequence of addresses u:

$$\begin{cases} f_2(xu) = f_2(u) & \text{if } 1 \le x \le B \wedge (\forall j : j \in F : \kappa(j) \ne suc(x)) \\ f_2(xu) = f_2(u) & \text{if } B < x \le B + N \wedge suc(W[x]) \ne \epsilon \\ & \quad \wedge (\forall j : j \in F : \kappa(j) \ne suc(W[x])) \\ f_2(xu) = (x,y) & \text{otherwise, with } y \text{ defined by:} \end{cases}$$

```
**if** $j : j \in F \wedge 1 \leq x \leq B : \kappa(j) = suc(x)$
   $\rightarrow$  $y = B + j$
$[]$ $B < x \leq B + N$
   $\rightarrow$ **if** $suc(W[x]) = \epsilon$
         $\rightarrow$  $y = B + N + 1$
      $[]$ $j : j \in F : \kappa(j) = suc(W[x])$
         $\rightarrow$  $y = B + j$
      **fi**
**fi**
```

The performance of the system when controlled by strategy f_2 is depicted in Figure 8.6. There is an obvious gain in the throughput of the system and a reduction in the latency. Apparently, transporting products back to the buffers is too expensive. In the next section we study another strategy, namely f_3, and we try to beat strategy f_2.

8.5.2 Yet another strategy

Strategy f_1 determines a new move on the basis of the movability of the products at the addresses in L. When also the current position of the robot is taken into account, the system will probably yield a better performance.

We distinguish 4 possible moves of the distribute robot: a product is moved from its buffer to a shop, or a product is moved from a shop to the output pile, a shop, or its buffer place. Respecting the order in L, we introduce 4 search routines on L, for which we introduce the following notations, with $1 \leq i \leq B$:

$$\begin{aligned} &nextFree(i) \equiv (\,\exists j : j \in F : \kappa(j) = suc(i)\,) \\ &fullyProcessed(i) \equiv suc(i) = \epsilon \end{aligned}$$

Each routine searches for the first $x \in L$ which satisfies its specific search condition. The conditions of the routines are:

1. $1 \leq x \leq B \wedge nextFree(x)$
2. $B < x \leq B + N \wedge fullyProcessed(W[x])$
3. $B < x \leq B + N \wedge nextFree(W[x])$
4. $B < x \leq B + N \wedge \neg nextFree(W[x]) \wedge \neg fullyProcessed(W[x])$

Although a routine may fail to find the x it looks for, when condition C holds at least one of the routines succeeds.

Strategy f_3 incorporates these search routines and its outcome depends on the actual place of the distribute robot. When the robot resides at a buffer place and condition C holds, a new transport is computed by applying the routines in the order: 1 2 3 4. The search is stopped when a routine succeeds. In the other case, when the robot resides at a shop, the applied order of the search routines is: 2 3 4 1.

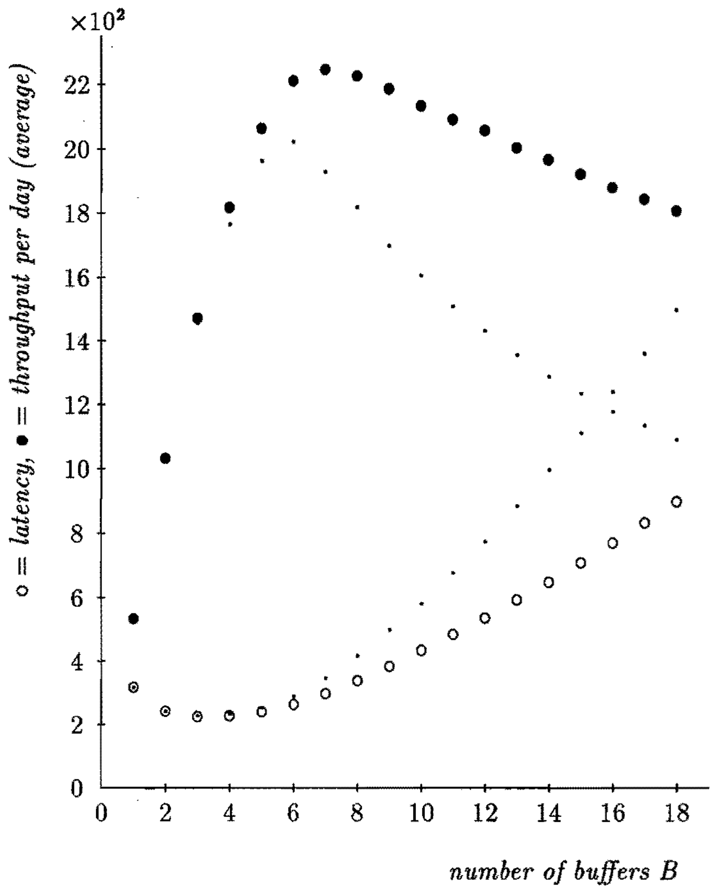


Figure 8.6: The characteristics of the job shop when the control uses strategy f_2. The small dots denote a copy of the characteristics of f_1.

By imposing these orders, we try to reduce the moves of the robot between shops and buffers in which it does not transport a product.

The results obtained with strategy f_3 are shown in Figure 8.7. We see that the new strategy realizes a larger throughput and a smaller latency than f_1, especially when there are more products present in the system. However, the performance figures of f_2 are still better than those of f_3.

8.5.3 More distribute robots

It goes without saying that there are much better strategies. We will not try to improve our strategy f_3 any further, but we do study the effect of an additional distribute robot. For that reason we need to update the factory model, where we

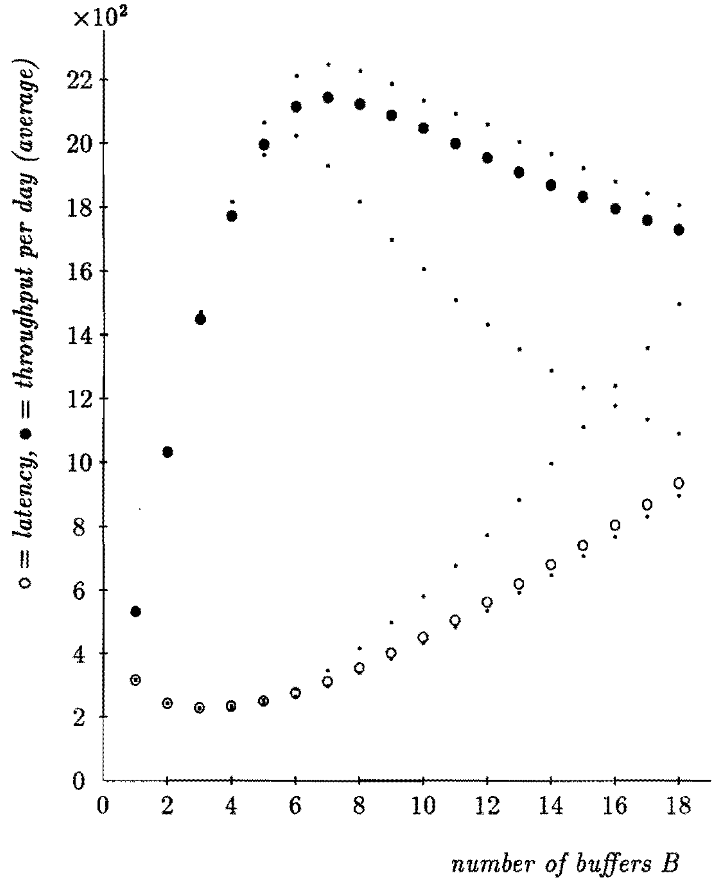


Figure 8.7: The characteristics of the job shop when the control uses strategy f_3. The small dots denote the characteristics of both f_1 and f_2 (see also Figure 8.6).

now assume the presence of K, $K \geq 1$, distribute robots. An overview of this new situation is given in Figure 8.8.

Apart from *PileIn* and *Robot*$_1$, the presence of K distribute robots requires a modification of the processes in the system. We do not describe all modifications completely, but we point out the most important changes to the control process.

For the situation with only one distribute robot, a shop is set to free when the robot is instructed to pick up the processed product at the shop. However, the actual moment upon which the shop becomes free is after the interaction with the robot. As a consequence, when two or more distribute robots are present, a robot may be instructed to put a product in a shop which is still occupied. In order to avoid these conflicts, a shop is added to F only when the 'take-away' interaction with a robot has happened. The shop signals its emptiness to the control via channel n. The reaction of the control on inputs from $n[j]$ is straightforward:

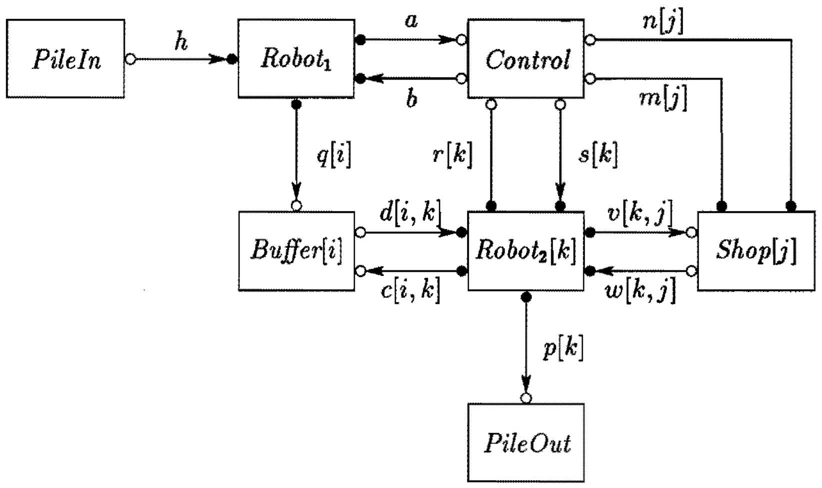


Figure 8.8: The job shop factory with K distribute robots, $1 \le k \le K$.

```
if $j : 1 \le j \le N : \bar{n}[j]$ $\rightarrow$ $n^\circ[j]?$
                          ; $Control(E, F \cup \{j\}, H, L, P, W)$
```

Another problem is caused by the products that are moved back to the buffers. In the presence of at least two distribute robots, the addition of a buffer address to L requires the product to be in its buffer; otherwise a robot may be instructed to move an absent product. Such anomalies are avoided when the completion of each product move to a buffer is signalled to the control. Therefore, we add to each distribute robot an additional output channel, namely channel r of type Signal. When distribute robot k has put the product back in its buffer and the interaction has been finished, a signal is sent via $r[k]$. The control keeps track of the instructed moves and, when receiving a completion signal from an r channel, the destination address of the corresponding move is added to L. Let us assume that the destination address of the last transport job given to robot k is recorded in variable $z[k]$. The reaction of the control is then as follws.

```
if $k : 1 \le k \le K : \bar{r}[k]$ $\rightarrow$ $r^\circ[k]?$
                          ; $Control(E, F, H, Lz[k], P, W)$
```

As a last update of the control process we mention the reaction on a request via an s channel. With the previous discussions in mind, the reaction is straightforward.

```
if $k : 1 \le k \le K \wedge C : \bar{s}[k]$
  $\rightarrow$ $(x, y) := f$
  ; $s^\circ[k]!(x, y)$
  ; if $1 \le x \le B$
      $\rightarrow$ $Control(E, F \backslash \{y - B\}, H, L \backslash x, P, W[y := x])$
```

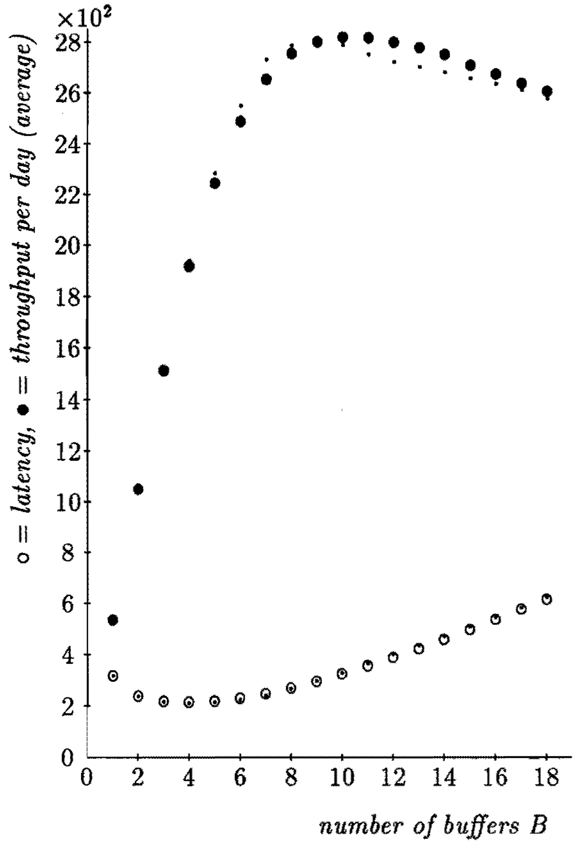


Figure 8.9: The performance characteristics of the system with 2 distribute robots, for control strategies f_2 and f_3. The throughput and latency of f_3 are given by '•' and 'o' respectively; the throughput and latency of f_2 are shown by the small dots.

```
[] $B < x \le B + N$
   $\rightarrow$ if $1 \le y \le B$
          $\rightarrow$ $Control(E, F, H, L\backslash x, P, W)$
      [] $B < y \le B + N$
          $\rightarrow$ $Control(E, F\backslash\{y - B\}, H, L\backslash x, P, W[y := W[x]])$
      [] $y = B + N + 1$
          $\rightarrow$ $Control(E, H, F, L\backslash x, P\backslash\{W[x]\}, W)$
      fi
fi
```

We have simulated the instance with $K = 2$ distribute robots. The outcomes of various simulation runs are given in Figure 8.9. As expected, the doubled transport capacity between the processes shows an increase of the throughput and a decrease of the latency. From the results we infer that, in case of strategy f_3, the maximum throughput is obtained for $B = 10$ and the minimum latency for $B = 4$. Suppose B is set to 10, then an interesting observation is the consequence of a mechanical defect in one of the distribute robots. In Figure 8.7 we see that for strategy f_3 and $B = 10$ the throughput is still close to the optimum. Figure 8.9 gives also the results for strategy f_2 when two distribute robots are present. This time we are lucky, f_3 is marginally better than f_2. Due to the larger transport capacity it becomes interesting to move partly processed products back to their buffer.

As promised in the beginning of this chapter, we end with a short study to the average number of jump-cuts performed by the simulation model per second of execution time. The numbers for strategy f_3 and both $K = 1$ and $K = 2$ are given in Figure 8.10. Apparently, the case with two buffers instead of one shows a larger average. As a consequence of increasing the number of processes, which in this case is obtained by enlarging B and K, the processor load increases and the performance gets worse. Due to the numerous communications, an additional distribute robot has much more impact on the performance than an additional buffer place.

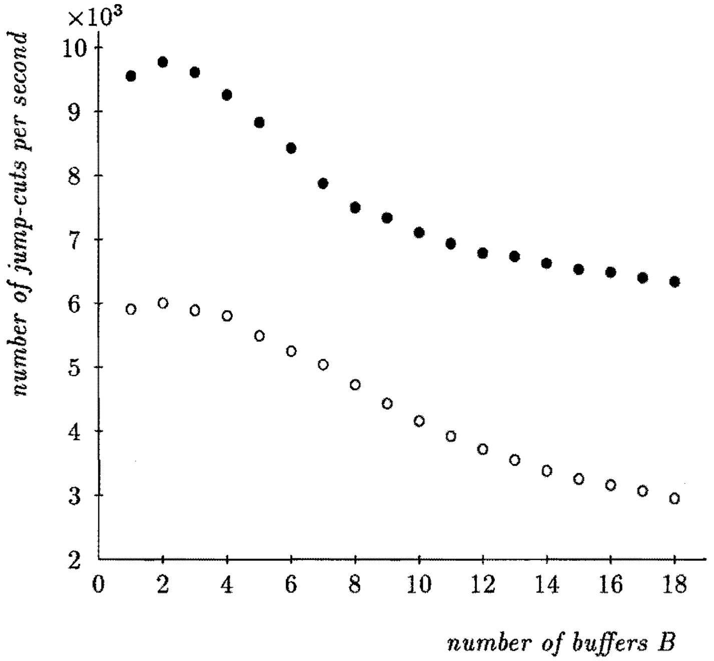


Figure 8.10: The average number of jump-cuts per second of execution time, as a function of the number of buffers B. Both curves are obtained with strategy f_3, with '●' and 'o' denoting that the number of distribute robots is 1 and 2 respectively.

Chapter 9

A traffic-light system

A nice example of a system consisting of a number of processes that want access to the same resource is given by a junction of roads. Cars arrive via several lanes, enter the crossing, and leave via another lane. In order to avoid the possibility of a collision, the entrances of the crossing are controlled by a traffic-light system. The underlying principle is not confined to this application only: similar lines are found in industry, where different product flows require a treatment by the same machine. We develop the specification of a traffic-light system and, for a specific instance, we study the waiting times incurred. This example is also used to compare the performance of the implementations that are obtained by the techniques described in the earlier chapters.

The lanes that meet at the junction are of certain types, which is either incoming or outgoing. Each incoming lane produces cars which pass the crossing via the same path leading to an outgoing lane. Different paths can cross each other. A lane produces or consumes at most one car at a time. The number of incoming lanes is denoted by N_I, $N_I \geq 2$, and the number of outgoing lanes is given by $N_O, N_O \geq 1$. Furthermore, we assume that there is at least one path to each outgoing lane, which implies $N_I \geq N_O$. In order to name the lanes, both the incoming and the outgoing lanes are numbered from 1 upward. The junction allows simultaneous access of cars from different lanes. However, to avoid the possibility of a collision, simultaneous access is restricted to incoming lanes with non-intersecting paths. We do not limit the number of cars in a path on the crossing. An example of a junction is given in Figure 9.1, in which there are 6 incoming and 3 outgoing lanes, $N_I = 6$ and $N_O = 3$. For the sake of clearness, the numbering of the outgoing lanes has been omitted. For this junction, the maximal sets of nonconflicting incoming lanes are: $\{1,2,6\}$, $\{2,3,4\}$, $\{2,4,6\}$, and $\{4,5,6\}$.

In Section 9.1 the control process is developed in a number of steps. The periphery of the control is also described.

In Section 9.2 the system obtained is closed by modelling the arrivals and departures of cars. In order to suit its environment, the periphery of the control requires a slight modification.

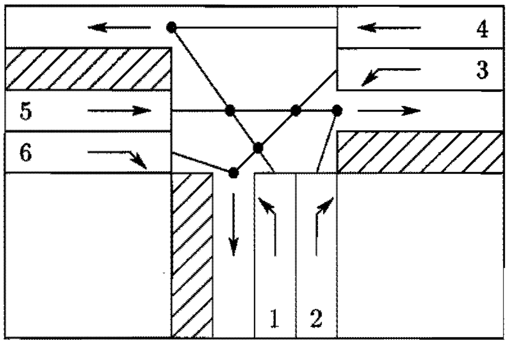


Figure 9.1: A junction of 6 incoming and 3 outgoing lanes. The dots represent the intersections of the paths in the junction area. The numbering of the outgoing lanes has been omitted.

In Section 9.3 a concrete instance is specified and a corresponding implementation is used to get an idea of the waiting times of cars.

In Section 9.4 we discuss various implementations and consider their performance, which is the time needed for a simulation run.

9.1 The control and the periphery

For the time being, we focus on the control process and the required periphery. The modelling of lanes and their junction is given in Section 9.2. We start with a simple description and successively add more 'features' which remove certain imperfections.

9.1.1 A first start

Since collisions must be avoided, the entrance of cars needs to be controlled. For that purpose, a light is positioned above each incoming lane. The lights guard the entrance to the crossing, and each of them has a changeable colour which is red or green. Of course, the cars in front of a red light are not allowed to enter and have to wait until the light becomes green. In case of a green light, the cars that have been waiting or that arrive enter the crossing one after another, in such a way that their order of arrival is preserved. This entering continues until the light is reset to red. We call an incoming lane *open* when its light is green; otherwise we say it is *closed*. The behaviours of the N_I lights are described by generic program *Light* which has passive input channel a of type Signal. Inputs from a cause a change in the colour of the light; the colour is recorded in variable *col* of type *Colour*, $Colour = \{red, green\}$. Program *Light* reads:

$$
\begin{aligned}
&\mathit{Light} \in \mathit{Colour} \rightarrow \Pi, \text{ with}\\
&\mathit{Light}(\mathit{col}) = a^{\circ}?\\
&\qquad ;\ \mathbf{if}\ \mathit{col} = \mathit{green} \rightarrow \mathit{Light}(\mathit{red})\\
&\qquad\ \ [\!]\ \mathit{col} = \mathit{red} \rightarrow \mathit{Light}(\mathit{green})\\
&\qquad\ \ \mathbf{fi}
\end{aligned}
$$

A simple control strategy consists of successively opening the lanes for a certain period, say *Topen*. After each admission interval we impose an entrance-free period of size *Tclose*, which is needed to empty the crossing. Control process $\mathit{Control}_1$ operates on basis of this idea and, in order to realize the control, it is connected to channels $a[i]$, $1 \leq i \leq N_I$.

$$
\begin{aligned}
\mathit{Control}_1 = {}& a^{\bullet}[1]!\ ;\ \delta(\mathit{Topen})\ ;\ a^{\bullet}[1]!\ ;\ \delta(\mathit{Tclose})\\
&;\ a^{\bullet}[2]!\ ;\ \delta(\mathit{Topen})\ ;\ a^{\bullet}[2]!\ ;\ \delta(\mathit{Tclose})\\
&\ \vdots\\
&;\ a^{\bullet}[N_I]!\ ;\ \delta(\mathit{Topen})\ ;\ a^{\bullet}[N_I]!\ ;\ \delta(\mathit{Tclose})\\
&;\ \mathit{Control}_1
\end{aligned}
$$

For the situation shown in Figure 9.1, we have already seen that some combinations of lanes may be open at the same time. A simple control scheme which exploits this is given by process $\mathit{Control}_{Fig}$:

$$
\begin{aligned}
&\mathit{Control}_{Fig} =\\
&\quad a^{\bullet}[1]!\ ,\ a^{\bullet}[2]!\ ,\ a^{\bullet}[6]!\\
&\ ;\ \mathit{Control}'_{Fig}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Control}'_{Fig} =\\
&\qquad\qquad\qquad\qquad\quad \delta(\mathit{Topen})\ ;\ a^{\bullet}[1]!\ ,\ a^{\bullet}[6]!\ ;\ \delta(\mathit{Tclose})\\
&\ ;\ a^{\bullet}[3]!\ ,\ a^{\bullet}[4]!\ ;\ \delta(\mathit{Topen})\ ;\ a^{\bullet}[3]!\ ;\ \delta(\mathit{Tclose})\\
&\ ;\ a^{\bullet}[6]!\ ;\ \delta(\mathit{Topen})\ ;\ a^{\bullet}[2]!\ ;\ \delta(\mathit{Tclose})\\
&\ ;\ a^{\bullet}[5]!\ ;\ \delta(\mathit{Topen})\ ;\ a^{\bullet}[4]!\ ,\ a^{\bullet}[5]!\ ;\ \delta(\mathit{Tclose})\\
&\ ;\ a^{\bullet}[1]!\ ,\ a^{\bullet}[2]!\\
&\ ;\ \mathit{Control}'_{Fig}
\end{aligned}
$$

Note that in program $\mathit{Control}_{Fig}$ we avoid the closing and re-opening of lanes that appear in successive admission periods.

It is rather awkward to spend $\mathit{Topen} + \mathit{Tclose}$ time on a car-free lane. Therefore, the first shortcoming of $\mathit{Control}_1$ we resolve is its unawareness of the absence and presence of waiting cars in incoming lanes.

In order to report the presence of a waiting car, we add an *informer* to each incoming lane. An informer observes the first place in its lane and reports via output channel b the presence of a waiting car. An overview is given in Figure 9.2. To record the current occupation of the first place, we introduce variable *occ* of type Bool. As b is used to signal the presence of a waiting car, communications via b happen only when the value of *occ* changes from false to true. Informers are specified by:

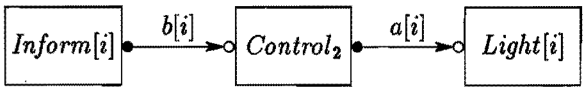


Figure 9.2: An overview of the control and periphery, with $1 \leq i \leq N_I$.

```
Informer ∈ Bool → Π, with
Informer(occ) = {wait until the occupation of the first place differs from occ}
                ; if  occ  → ε
                  [] ¬occ  → b•!
                  fi
                ; Informer(¬occ)
```

A precise statement of the phrase 'wait until the occupation of the first place differs from occ' is postponed until we have specified the lanes. The information supplied by the informers is used in *Control*$_2$.

```
Control₂ = if i : 1 ≤ i ≤ N_I : b̄[i]
              → a•[i]! , b°[i]?
                ; Admission(i, τ+Topen)
                ; a•[i]!
                ; δ(Tclose)
                ; Control₂
           fi
```

In process *Admission*$(i, \tau + Topen)$ new reports via $b[i]$ are read until the entrance period of size *Topen* has elapsed.

```
Admission ∈ [1..N_I] × T → Π, with
Admission(i,T) = if b̄[i] ∧ τ < T
                      → b°[i]? ; Admission(i,T)
                 [] τ ≥ T
                      → ε
                 fi
```

A major drawback of *Control*$_2$ is the absence of liveness: the control is not prohibited from choosing the same 'car-containing' road all the time. Yet another deficiency is the absence of simultaneous access of non-conflicting lanes. In the next section we resolve both shortcomings.

9.1.2 Simultaneous access

In order to enlarge the throughput of the crossing, we allow, as has already been illustrated by program *Control*$_{Fig}$, simultaneous access of lanes with non-intersecting paths in the junction area. For a more general control description, we assume the presence of function *collision* $\in [1..N_I] \times [1..N_I] \rightarrow$ Bool, with

$$\mathit{collision}(i,j) \equiv \text{`opening lanes } i \text{ and } j \text{ simultaneously can result in a collision'}$$

where $\mathit{collision}(i,i)$ equals false. The collision function that belongs to the junction of Figure 9.1 is captured by the following symmetric matrix, in which true and false are denoted by 1 and 0 respectively:

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

For X a set of lanes, we say that X is *collision-free*, which is denoted by $cf(X)$, if the danger of a collision is absent:

$$cf(X) \equiv \neg(\,\exists i,j : i,j \in X : \mathit{collision}(i,j)\,)$$

The control records in variable A of type $\mathcal{P}([1..N_I])$ the lanes that are currently open; this set is called the *admission set*. As an invariant property of A we maintain $cf(A)$. The set of closed lanes is determined by the symmetric set difference $A \div [1..N_I]$.

When informer i, $i \notin A$, reports the presence of a waiting car, and $cf(A \cup \{i\})$ holds, we may open lane i without creating the possibility of a collision. In case of a conflicting lane, its number is added to the end of refusal list R of type $[1..N_I]^*$. We see to it that a lane appears at most once in R. Note $i \in R$ means that i is in list R. We also maintain the property that A cannot be extended with elements from R, without violating the collision-freeness of A.

$$(\,\forall i : i \in R : \neg cf(A \cup \{i\}\,)$$

In order to give each lane the opportunity to pass its cars to the crossing, a new admission set A is computed from R after each admission period. For this purpose we introduce control strategy f which inspects the elements in R in the order of addition. When the element being inspected and the newly formed admission set are collision-free, the element is added to the new set. Formally, strategy f satisfies

$$f \in [1..N_I]^* \times \mathcal{P}([1..N_I]) \to \mathcal{P}([1..N_I])$$

and is defined as follows, with $x \in [1..N_I]$:

$$\begin{cases} f(\epsilon, X) = X & \\ f(xL, X) = f(L, X) & \text{if } \neg cf(X \cup \{x\}) \\ f(xL, X) = f(L, X \cup \{x\}) & \text{if } \ \ cf(X \cup \{x\}) \end{cases}$$

Note that a new admission set is determined by $f(R, \emptyset)$.

The control is performed by process $Control_3(Tstart, R, A)$ which retains the following invariant:

$$A \neq \emptyset \vee R = \epsilon$$

Variable *Tstart* is used to record the beginning of the current admission period. In the specification of $Control_3$ we use construct $(\,\mathbf{par}\, i : P : E\,)$, where P a predicate and E a program, for denoting the parallel composition of all possible instantiations of E. In case of an empty range, the construct yields the empty program ε. We denote by $R\backslash i$ the list equal to R except that element i has been removed; the removal of a whole set S of elements is denoted by $R\backslash S$.

```
$Control_3 \in \mathcal{T} \times [1..N_I]^* \times \mathcal{P}([1..N_I]) \rightarrow \Pi$, with
$Control_3(Tstart, R, A) =$
    $\mathbf{if}\ i : 1 \leq i \leq N_I \wedge A = \emptyset : \bar{b}[i]$
        $\rightarrow\ a^\bullet[i]!\ ,\ b^\circ[i]?\ ;\ Control_3(\tau, \epsilon, \{i\})$
    $[]\ A \neq \emptyset$
        $\rightarrow \mathbf{if}\ i : 1 \leq i \leq N_I \wedge i \notin R \wedge \tau < Tstart + Topen : \bar{b}[i]$
            $\rightarrow \mathbf{if}\ i \in A$
                $\rightarrow\ b^\circ[i]?\ ;\ Control_3(Tstart, R, A)$
            $[]\ i \notin A \wedge \neg cf(A \cup \{i\})$
                $\rightarrow\ Control_3(Tstart, Ri, A)$
            $[]\ i \notin A \wedge cf(A \cup \{i\})$
                $\rightarrow\ a^\bullet[i]!\ ,\ b^\circ[i]?\ ;\ Control_3(Tstart, R, A \cup \{i\})$
            $\mathbf{fi}$
        $[]\ \tau \geq Tstart + Topen$
            $\rightarrow\ (\,\mathbf{par}\, i : i \in A : a^\bullet[i]!\,)$
              $;\ \delta(Tclose)$
              $;\ A' := f(R, \emptyset)$
              $;\ (\,\mathbf{par}\, i : i \in A' : a^\bullet[i]!\ ,\ b^\circ[i]?\,)$
              $;\ Control_3(\tau, R\backslash A', A')$
        $\mathbf{fi}$
    $\mathbf{fi}$
```

During delay $\delta(Tclose)$, the control is reluctant to receive information of the informers. It is easy to change the specification in such a way that these reports are added to R. A possible realization consists of replacing delay $\delta(Tclose)$ by process $Pass(\tau + Tclose, R)$. The specification of processes *Pass* is:

```
$Pass \in \mathcal{T} \times [1..N_I]^* \rightarrow \Pi$, with
$Pass(T, L) =\ \mathbf{if}\ \tau \geq T$
                   $\rightarrow\ \varepsilon$
               $[]\ i : 1 \leq i \leq N_I \wedge i \notin L \wedge \tau < T : \bar{b}[i]$
                   $\rightarrow\ Pass(T, Li)$
               $\mathbf{fi}$
```

9.1.3 Closing idle lanes

In $Control_3$, a lane retains its admission until a new admission set has to be computed. As a consequence, an opened lane, which shows no activity after a while, may unnecessarily prevent others from entering the crossing; this is an undesirable situation. We overcome this shortcoming by taking the activities in the lanes into account.

In a sense, the activities in the lanes are reported to the control by the informers. When the informer of an opened lane signals sufficiently often the presence of another car, it is attractive to keep the lane open. In order to give a meaning to 'sufficiently often,' we introduce the notion of an *idle* lane: we say that an opened lane becomes idle when during a period of size *Tidle* its informer did not signal the arrival of a new car. More precisely, for opened lane i, the lane is set to idle when τ has expired $\sigma(b[i]) + \mathit{Tidle}$. Upon reaching its idle state, the lane is closed and, hence, no longer in the admission set.

To record idle lanes we introduce set I of type $\mathcal{P}([1..N_I])$ and we maintain $cf(A \cup I)$ as an invariant. When removing an element from I, we extend the admission set with lanes from R. However, opening other lanes requires the removed lane to be idle for a period of at least *Tclose*. For idle lane i, this condition is expressed by $\tau \geq \sigma(a[i]) + \mathit{Tclose}$: after the light has been set to red, a period of *Tclose* has elapsed. The extension of the admission set is determined by:

$$f(R, A \cup (I \backslash \{i\})) \setminus (A \cup (I \backslash \{i\}))$$

The informer belonging to an idle lane can signal the arrival of a new car; these reports are added to R. As a consequence, it is possible that the admission set can be expanded by elements from R. Therefore, we change the invariant which expresses the non-extensibility of the admission set with elements in R into:

$$(\forall i : i \in R : \neg cf(A \cup I \cup \{i\}) \vee i \in I)$$

Note that $A \cap I = \emptyset$, but that not necessarily $I \cap R = \emptyset$. The control process sees to it that $A \cup I \neq \emptyset \;\vee\; R = \epsilon$ is never violated.

```
$Control_4 \in \mathcal{T} \times [1..N_I]^* \times (\mathcal{P}([1..N_I]))^2 \rightarrow \Pi$, with
$Control_4(Tstart, R, A, I) =$
  if $i : 1 \leq i \leq N_I \wedge A \cup I = \emptyset : \bar{b}[i]$
    $\rightarrow a^\bullet[i]!\ ,\ b^\circ[i]?\ ;\ Control_4(\tau, \epsilon, \{i\}, \emptyset)$
  $[]$ $A \cup I \neq \emptyset$
    $\rightarrow$ if $i : 1 \leq i \leq N_I \wedge i \notin R \wedge \tau < Tstart + Topen : \bar{b}[i]$
          $\rightarrow$ if $i \in A$
               $\rightarrow b^\circ[i]?\ ;\ Control_4(Tstart, R, A, I)$
             $[]$ $i \in I$
               $\rightarrow Control_4(Tstart, Ri, A, I)$
             $[]$ $i \notin A \cup I \wedge \neg cf(A \cup I \cup \{i\})$
               $\rightarrow Control_4(Tstart, Ri, A, I)$
```

```
            $[]\ i \notin A \cup I \wedge cf(A \cup I \cup \{i\})$
              $\rightarrow a^{\bullet}[i]!\ ,\ b^{\circ}[i]?\ ;\ Control_4(Tstart, R, A \cup \{i\}, I)$
            fi
        $[]\ i : i \in A : \sigma(b[i]) + Tidle \leq \tau < Tstart + Topen$
          $\rightarrow a^{\bullet}[i]!\ ;\ Control_4(Tstart, R, A \backslash \{i\}, I \cup \{i\})$
        $[]\ i : i \in I : \sigma(a[i]) + Tclose \leq \tau < Tstart + Topen$
          $\rightarrow A' := f(R, A \cup (I \backslash \{i\})) \setminus (A \cup (I \backslash \{i\}))$
            ; ( par $j : j \in A' : a^{\bullet}[j]!\ ,\ b^{\circ}[j]?$ )
            ; $Control_4(Tstart, R \backslash A', A \cup A', I \backslash \{i\})$
        $[]\ \tau \geq Tstart + Topen$
          $\rightarrow$ ( par $j : j \in A : a^{\bullet}[j]!$ )
            ; $Pass(\tau + Tclose, R)$
            ; $A' := f(R, \emptyset)$
            ; ( par $j : j \in A' : a^{\bullet}[j]!\ ,\ b^{\circ}[j]?$ )
            ; $Control_4(\tau, R \backslash A', A', \emptyset)$
        fi
    fi
```

9.2 Towards a closed system

In order to obtain a description of the whole system, we add the environment which consists of the incoming lanes and their junction only. There is no need to model the outgoing lanes. An overview of the closed system we aim at is given in Figure 9.3. Obviously, the periphery needs to be adapted to the environment: channels c and d have to be taken into account.

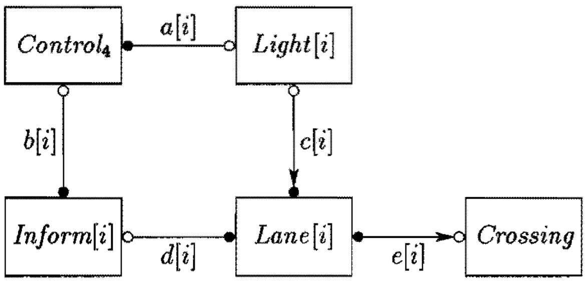


Figure 9.3: The closed system, with $1 \leq i \leq N_I$.

A first modification concerns the inspection of the colour of a light. For that purpose, the lights are extended with channel c of type Bool. A light is willing to answer an inspection only when its colour is green, thereby causing a possible suspension of its inquiring lane. The answer of a light via the c channel consists of a boolean which indicates whether the colour of the light was red at the moment of inspection. Therefore we add variable *wait* of type Bool to the description of the lights; the variable records whether the colour of the light was red at the moment of inspection. In order to prevent the control from being suspended, communications via the a channel are given a higher priority than communications via the c channel.

```
$Light \in Colour \times \mathrm{Bool} \rightarrow \Pi$, with
$Light(col, wait) =$
  if $\neg\bar{a} \wedge \bar{c} \wedge col = green$
     $\rightarrow c^{\circ}!wait$
       ; $Light(col, \mathrm{false})$
  [] $\neg\bar{a} \wedge \bar{c} \wedge col = red \wedge \neg wait$
     $\rightarrow Light(col, \mathrm{true})$
  [] $\bar{a}$
     $\rightarrow a^{\circ}?$
       ; if $col = green \rightarrow Light(red, wait)$
         [] $col = red \rightarrow Light(green, wait)$
         fi
  fi
```

The informer waits until the occupation of the first place in the lane differs from the recorded state in *occ*. We assume that the lanes report these state changes via their d channel to the informers. Moreover, cars are controlled by the traffic lights only, and they are not held up by the informers. In order to achieve this, communications via d must be possible at any time. We solve this by splitting the informer process into two processes, namely processes *State* and *Reporter*. An overview is depicted in Figure 9.4.

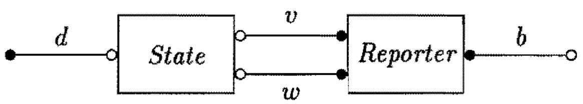


Figure 9.4: The contents of an informer.

Process *State* has variable *occ* which denotes the occupation of the first place of the lane; a communication via d changes the recorded value. The process is willing to read the v channel when the first place is unoccupied; otherwise it is willing to read the w channel.

```
$State \in \mathrm{Bool} \rightarrow \Pi$, with
$State(occ) =$ if $\bar{d} \rightarrow d^{\circ}?$ ; $State(\neg occ)$
               [] $\bar{v} \wedge \neg occ \rightarrow v^{\circ}?$ ; $State(occ)$
               [] $\bar{w} \wedge occ \rightarrow w^{\circ}?$ ; $State(occ)$
               fi
```

The reporter repeatedly does: a communication via w, after which it reports via b the occupation to the control, and then it performs a v communication which succeeds when the place is free again.

$$Reporter = w^{\bullet}! \ ; \ b^{\bullet}! \ ; \ v^{\bullet}! \ ; \ Reporter$$

As a result, the informer of a lane with an initially unoccupied first place reads:

$$Informer = \{State(\text{false}), Reporter\}$$

The environment interacts with the developed periphery. We omit the specification of the crossing, because it is simply but effectively modelled by reading all possible input channels. The contents of the lanes is illustrated in Figure 9.5. The generator creates new cars which are sent via channel p to a buffer with infinite capacity. The first place in the queue is explicitly modelled by process *First*, a special kind of one-place buffer. Cars enter process *First* via channel q and leave via channel e. The traffic flow happens via channels of type *Cars*. The purpose of the c and d channels has been described before.

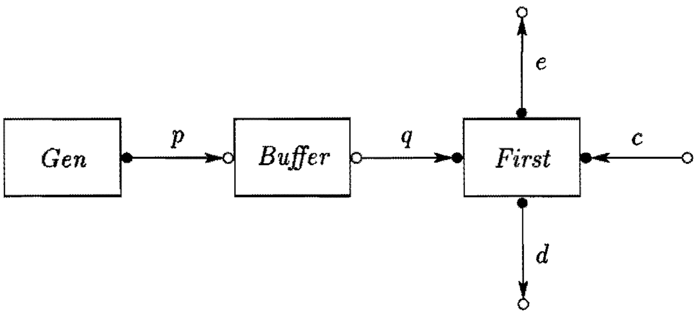


Figure 9.5: The contents of a lane.

The cars in the lanes arrive one after another. The time between two successive arrivals is generated by a distribution function. Similar generators have been used in previous chapters and, for that reason, we omit the program that specifies their behaviours.

When entering the buffer, the behaviour of a car depends on the presence of others. If there are no other cars present and process *First* is willing to accept a car, the car leaves the buffer immediately. Otherwise, the car takes position at the end of the queue and is given a restart delay when leaving the buffer. The imposed restart delay varies and is generated by distribution function $\mathcal{D}_R$. The contents of the FIFO-queue is described by variable L of type $Cars^*$.

```
$Buffer \in Cars^* \rightarrow \Pi$, with
$Buffer(L) =$ if $\bar{p}$
                  $\rightarrow$ $p^{\circ}?x$
                   ; if  $\bar{q} \wedge L = \epsilon \rightarrow q^{\circ}!x \ ; \ Buffer(\epsilon)$
                     [] $\neg\bar{q} \vee L \neq \epsilon \rightarrow Buffer(Lx)$
                     fi
```

```
[] $\bar{q} \wedge L \neq \epsilon$
   $\rightarrow \delta(\mathcal{D}_R)$
      ; $q^\circ!(hd.L)$
      ; $Buffer(tl.L)$
fi
```

Note that restart delay $\delta(\mathcal{D}_R)$ may suspend the arrival of new cars. A way to avoid this is analogous to the introduction of process *Pass* in the specification of the control process.

The one-place buffer *First* takes the next car from the buffer and reports its occupation via *d*. Afterwards, it inspects the light and it is suspended until the colour is green. If the light was red at the moment of inspection, which is indicated by the answer of the light, the car is given a restart delay. Next, the car is sent to the crossing and the change in its occupation is reported via *d*.

```
$First =$ $q^\bullet?car$ ; $d^\bullet!$
         ; $c^\bullet?wait$
         ; if  $wait \rightarrow \delta(\mathcal{D}_R)$
           [] $\neg wait \rightarrow \varepsilon$
           fi
         ; $e^\bullet!car$ ; $d^\bullet!$
         ; $First$
```

Initially, we assume that there are no cars present in the system. Hence, a lane is specified by:

$$Lane = \{Gen(0), Buffer(\epsilon), First\}$$

Furthermore, all lanes are closed in the initial state. As a result, the admission set and the idle set of the control process are empty. Yet another consequence of the absence of cars is an empty refusal list *R*. Gathering the described processes yields the following description of the closed traffic-light system:

$$\begin{array}{l} \quad \{Control_4(0, \epsilon, \emptyset, \emptyset)\} \\ \cup \ \{i : 1 \leq i \leq M : Light[i](red, \text{false})\} \\ \cup \ \{i : 1 \leq i \leq M : Informer[i]\} \\ \cup \ (\cup i : 1 \leq i \leq M : Lane[i]) \\ \cup \ \{Crossing\} \end{array}$$

9.3 A particular instance

Now that we have a real-time description of the traffic-light system, we use it to study a particular instance. First we formulate the instance, and next we show the simulation results.

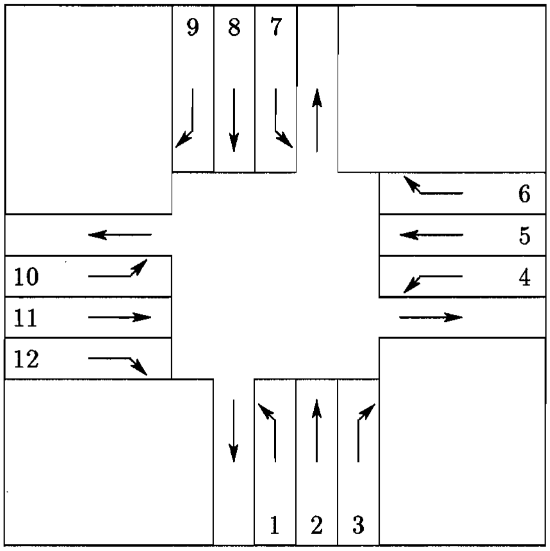


Figure 9.6: A sketch of the site.

A sketch of the instance we consider is given in Figure 9.6. There are 12 incoming and 4 outgoing lanes, with the desired direction of each incoming lane indicated by an arrow. The corresponding collision function speaks for itself. For example, opening lanes 1, 6, 7, and 12 is allowed, but adding another one results in an undesired situation. The arrival of cars is described by a Poisson arrival process, with an average of 5 between two consecutive arrivals. Lanes are open for at most *Topen* = 100, and the time needed to empty a path on the crossing is *Tclose* = 10. The restart time of a car is also described by a Poisson process; the average restart time is set to 0.1.

We use the implementation to determine the average waiting time *Twait* of a car, as a function of *Tidle*. The waiting time is the amount of time spent in processes *Buffer* and *First*. The simulation outcomes are shown in Figure 9.7. Since the crossing looks the same from each direction, north, east, south, and west, the average waiting times of only lanes 1, 2, and 3 are given. It turns out that the figures for lanes 1 and 2 are about equal. This is not very surprisingly, because their paths on the crossing area are about equally demanding. Conform our expectations, the waiting times of lane 3 are shorter than the waiting times of the other lanes; this stems from a smaller number of intersections on the crossing.

When we start with *Tidle* = 10^2 and successively shorten it, we find a reduction in the waiting times for *Tidle* = $10^{1.2}$. Up to this point, the waiting times are determined by *Topen*. The minimum average waiting times are found to be near

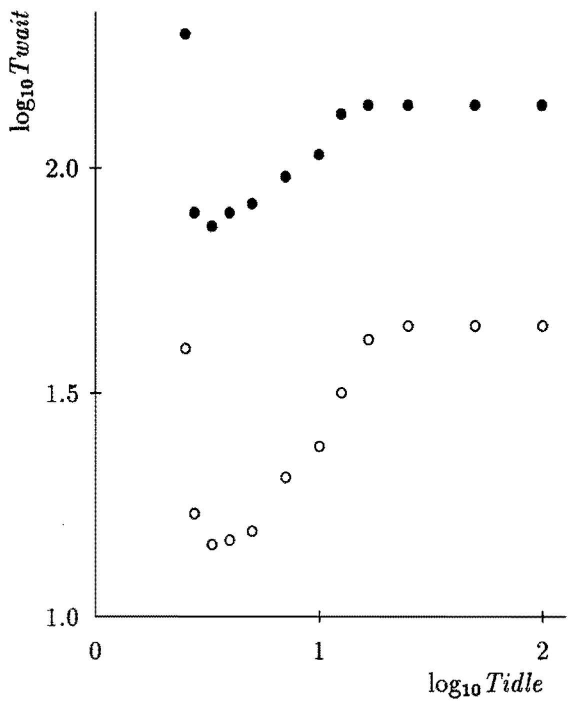


Figure 9.7: The average waiting time as a function of *Tidle*, with *Trestart* = 0.1. Lanes 1 and 2 are denoted by symbol '•' and lane 3 by 'o'.

Tidle = $10^{0.5}$. Making *Tidle* even shorter causes a very rapid increase of the waiting times. This is caused by the fact that *Tidle* approaches *Trestart*: a slow start is interpreted by the control as the absence of other cars and consequently closes the lane. As a result of this, not all waiting cars will be passed to the crossing which causes a prolonged stay in the buffers. The optimum is situated very close to the critical region. Hence, setting the idle time equal to the optimum allows very little variation in the restart time; it is probably better to choose *Tidle* less well-timed.

Obviously, increasing the restart times causes the waiting times to increase. The simulation results for *Trestart* = 0.5 are given in Figure 9.8. Comparing these outcomes with the ones in Figure 9.7 reveals a significant shift up of the minimum waiting times.

9.4 A comparison of techniques

We use the traffic-light system to compare the speed of the implementations that are obtained by the techniques described in the previous chapters. We have also made a 'traditional' undistributed discrete-event implementation of the system [13, 37].

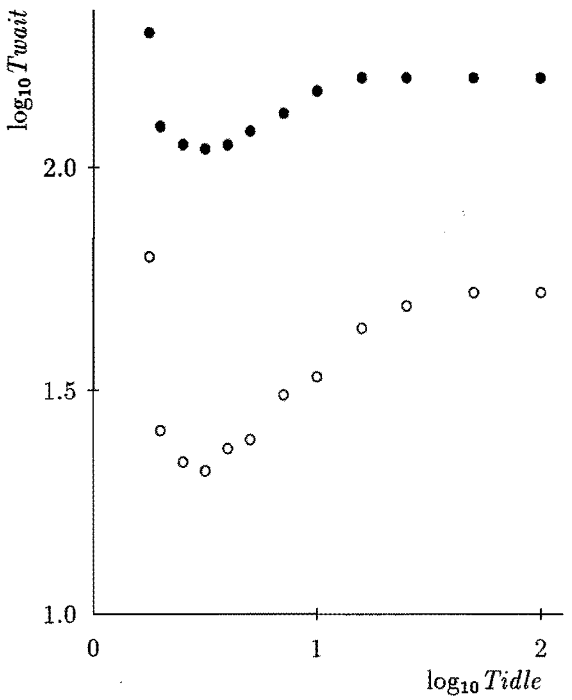


Figure 9.8: The average waiting time as a function of *Tidle*, now with *Trestart* = 0.5. Again, lanes 1 and 2 are denoted by symbol '•' and lane 3 by 'o'.

In the distributed implementation, a ring structure with local rings has been added. For varying numbers of processors, the speedup figures are shown in Figure 9.9. All processes mapped on a single processor are linked in a local ring. The lanes are spread over the available processors but one, where the other processes run. Note that the case of 6 processors is not included because 5 is not a divisor of 12. The loss in performance when using 2 processors instead of 1 is a consequence of the communication introduced between the processors; all processes in the lanes are still on a single processor. From the results follows that the optimum is found for the distribution over 5 processors.

The measured simulation times are given in table 9.1, where 'undistributed' denotes the undistributed discrete-event simulation. The distributed implementations are given with the number of processors on which the computation has been mapped. It is obvious that the undistributed implementation performs much better than the others. Only the jump-cutted approach comes rather close. In fact, there is hardly any parallelism in the traffic-light system, which is reflected by the figures. As a result of the dense time domain, the events are all ordered, and the communication associated with selecting a next event via the token ring proves to be too much time consuming. The jump-cutted implementation has been obtained most easily.

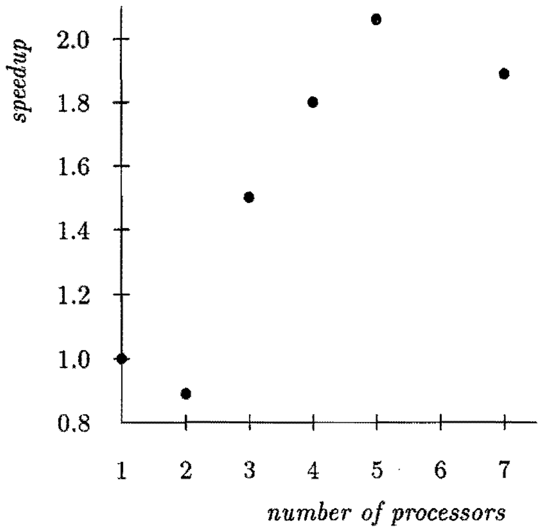


Figure 9.9: The speedup figure of the distributed implementation.

method	*execution time in secs.*
undistributed	4.5
jump-cuts	7.9
distributed, using 1 proc	101.8
distributed, using 5 procs	49.4

Table 9.1: The execution times of different methods applied to the same problem instance.

Chapter 10

Conclusion

The challenge that gave rise to the research presented in this thesis stems from a combined interest from the areas of computing science and mechanical engineering, especially industrial automation: 'how can formal methods assist the modelling of industrial systems?' The answer has been given by describing a possible modelling approach suggested by the following point of view. We look upon an industrial system as a collection of parallel-operating mechanisms that interact with each other. In this approach, the mechanisms taking part are distinguished and modelled separately by processes. The parallel composition of the processes specifies the entire system. Instead of restricting ourselves to the underlying formalism, we have described the whole path from developing a theoretical framework for cooperating mechanisms to the construction of corresponding computer programs that simulate the specified behaviours. The path taken is far from unique: other basic formalisms, other simulation programs, and other paths are also possible. In order to illustrate the power of the approach, it is applied to several industrial systems taken from the realm of factory control.

We have chosen the Enabling Model as our basic formalism to describe the timed behaviour of cooperating mechanisms, which supports the specification of true concurrency. Although the original incentive to the design of the Enabling Model evoked from the need for performance analyses of choice-free concurrent mechanisms, the model proves to be even more powerful. Without causing any serious difficulties, the model can be used to describe internal and external choice, which are introduced for specifying certain freedoms of operation and dependencies on the environment, respectively. A drawback of the model is its complexity. We need, however, only a small part of the theory, which is, due to a number of simplifications carried through, more manageable and comprehensible than the original full-bodied version.

Since the Enabling Model in its basic form turns out to be less suitable for modelling more complex systems—it tends to be too laborious—we have added a specification language with its semantics defined in the basic formalism. This language consists of primitive programs and composition operators to make more interesting programs. The operators are: serial and parallel composition, internal and external choice, and bounded and unbounded recursion. A possible further extension concerns so-called interrupts. Although we have not described it, the extension can be achieved. Basically two possibilities have to be taken into account: an interrupt caused by a moment in time, and an interrupt caused by an action.

In order to give more abstract specifications in which ordered actions can happen at the same moment, the Enabling Model has been extended with the possibility to model zero delays. The extension embroiders on the original model. A disadvantage of the course taken is the resulting complexity: verifying the equivalence of programs becomes even more difficult. An extension based on a partial-order semantics will probably not suffer from this shortcoming, though it will be difficult to accomplish.

The interactions of mechanisms are described by data communications, by which the gap between specification and implementation language is narrowed. To abstract from explicit communication requests in specifications, active and passive communications are distinguished: one side of the channel is active and the other side is passive. The actual assignment of the active and passive side is often at choice. Hence, we could consider the introduction of another type of communication, namely indifferent communication.

Just plain theory is not very rewarding; examples are needed to illustrate its use and capabilities. In our case, the expressive power of the specification language is exemplified by a number of case studies in the field of factory control. Often, such systems comprise a central control unit which regulates the machines present. The behaviour of the control may vary, but most of the time it consists of a large external choice among the possible inputs, possibly strengthened by the conjunction of extra conditions. When answering requests from the environment, suspension periods are often undesirable. A more complex control process operates according to a certain strategy. The effect of a strategy is often difficult to foresee. For that reason, several alternatives are tried out in computer simulations, and their performances are determined from the outcomes. Another important aspect of the implementation is its use for validation purposes, ascertain whether the specified concept corresponds to the concrete equivalent it models.

In the implementation approach, we try to exploit the use of a parallel computer, a processor network. Therefore, the parallelism present in the specification is straightforwardly implemented, which yields a distributed discrete-event simulation. Problems are caused by the external choice: the selection of a valid alternative requires a proper evaluation of the guards, which is solved at the expense of adding a token ring or more efficient structures to the implementation. It turns out that the communications via the token ring consume a considerable amount of time and, hence, should not happen too often. In the examples we looked at, there occur no real concurrent, computationally intensive tasks from which the potential parallelism of the implementation device could benefit. Usually, the operations are scheduled successively in time, thereby causing a more or less sequential nature of the implementation. As a result, the distributed implementation approach appears to be expensive, though it is easily obtained from its specification. In order to exploit the full capabilities of a parallel computer, more parallelism and, hence, less synchronization between the processes is required.

Problems we did not address, but which are certainly important to pay attention to, are related to the actual mapping of the implementation on the available processor network. In order to realize the actual execution of the implementation, some basic tools like routing and multiplexing software are indispensable. The assignment of processes to processors has been done manually, taking the communication and computation costs into account. A fully automatic translation scheme from specification to subsequent parallel implementation is possible, but probably results in a poor implementation. Another interesting research topic is to vary the granularity of the implementation, that is to look at the effect of a mixed distributed and undistributed approach which is obtained by combining a number of parallel processes into a single sequential one. We could, for example, combine the processes with an interleaved execution behaviour and retain those with inherent parallelism.

A fast implementation is achieved by using the approximation technique yielding a simulation on just one processor. As a consequence of the single processor, the execution happens in a sequential way. As indicated by the word 'approximation,' care should be taken with respect to the outcomes, and a validation by means of scaling the computation time is required. A drawback of scaling the computation time is the limited simulation period due to the maximum time value. Therefore, the technique is applicable to only relatively simple cases requiring just a small scale factor. A major advantage of this method is the almost literal translation of specification to implementation; just one process, which takes care of advancing the clock, has to be added. However, it would be nice to have a dedicated processor doing the time advancements, since in that way an even faster implementation can be obtained.

In conclusion, we do not claim that the approach presented is final or complete. Nevertheless, our approach is an adequate means in the design of industrial systems. The modelling of industrial systems can be done in a clear and concise way by a method based on a formal theory.

Bibliography

[1] J.H.A. Arentsen. *Factory Control Architecture: A Systems Approach.* PhD Thesis, Eindhoven: Eindhoven University of Technology, 1989.

[2] J.C.M. Baeten and J.A. Bergstra. Real-time process algebra. *Formal Aspects of Computing* 3(1991); pp. 142–188.

[3] G.C. Barney and S.M. Dos Santos. *Lift Traffic Analysis Design and Control.* Stevenage: Peregrinus, 1977. (IEE Control Engineering Series: vol. 2)

[4] C.H. van Berkel, C. Niessen, M. Rem and R.W.J.J. Saeijs. VLSI programming and silicon compilation. In: *Proc. International Conf. Computer Design (ICCD).* New York: IEEE Computer Society Press, 1988; pp. 150–166.

[5] C.H. van Berkel. *Handshake Circuits: An Intermediary between Communicating Processes and VLSI.* PhD Thesis, Eindhoven: Eindhoven University of Technology, 1992.
C.H. van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming.* Cambridge: Cambridge University Press, 1993. (Cambridge International Series on Parallel Computation: to appear)

[6] M. Broy. *An Example for the Design of Distributed Systems in a Formal Setting: The Lift Problem.* Technical Report MIP-8802, Passau: Passau University, 1988.

[7] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Reading, M.A.: Addison-Wesley, 1988.

[8] R.B. Chase and N.J. Aquilano. *Production and Operations Management: A Life Cycle Approach.* Homewood: Irwin, 1992.

[9] J. Davies. *Specification and Proof in Real-Time CSP.* Cambridge: Cambridge University Press, 1993. (Distinguished Dissertations in Computer Science)

[10] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica* 1(1971); pp. 115–138.
E.W. Dijkstra. Hierarchical ordering of sequential processes. In: *Operating Systems Techniques*; ed. C.A.R. Hoare and R.H. Perrott. New York: Academic Press, 1972; pp. 72–93.

[11] E.W. Dijkstra. *A Discipline of Programming.* Englewood Cliffs: Prentice-Hall, 1976.

[12] E.W. Dijkstra, W.H.J. Feijen and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters* 16(1983); pp. 217–219.

[13] G.S. Fishman. *Principles of Discrete Event Simulation.* New York: Wiley, 1978.

[14] M.J. Flynn. Some computer organisations and their effectiveness. *IEEE Transactions on Computers* C-21(1972); pp. 948–960.

[15] P.A.J. Hilbers. *Processor Networks and Aspects of the Mapping Problem.* Cambridge: Cambridge University Press, 1991. (Cambridge International Series on Parallel Computation: vol. 2)

[16] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM* 21(1978); pp. 323–334.

[17] C.A.R. Hoare. *Communicating Sequential Processes.* London: Prentice-Hall, 1985.

[18] D.J. Holding and G.F. Carpenter. Software fault tolerance in real-time systems. In: *Parallel Processing in Control: The Transputer and other Architectures*; ed. P.J. Fleming. London: Peregrinus, 1988. (IEE Control Engineering Series: vol. 38); pp. 126–157.

[19] C. Huizing. *Semantics of Reactive Systems: Comparison and Full Abstraction.* PhD Thesis, Eindhoven: Eindhoven University of Technology, 1991; pp. 103–120.

[20] INMOS Limited. *Occam 2 Reference Manual.* London: Prentice-Hall, 1988.

[21] INMOS Limited. *Transputer Reference Manual.* London: Prentice-Hall, 1988.

[22] INMOS Limited. *Transputer Instruction Set: A Compiler Writer's Guide.* London: Prentice-Hall, 1989.

[23] G. Jones and M. Goldsmith. *Programming in Occam 2.* London: Prentice-Hall, 1988.

[24] W.E.H. Kloosterhuis. *The Enabling Model: A Tool for Performance Analysis of Concurrent Mechanisms.* PhD Thesis, Eindhoven: Eindhoven University of Technology, 1991.

[25] J.J. Lukkien. *Transputer Pascal: A User Manual.* Technical Report CS8912, Groningen: University of Groningen, 1989.

[26] J.J. Lukkien. *Parallel Program Design and Generalized Weakest Preconditions.* PhD Thesis, Groningen: University of Groningen, 1991.

[27] J.J. Lukkien. *The Eindhoven Transputer System.* Lecture Notes 2485, Eindhoven: Eindhoven University of Technology, 1992.

[28] A.J. Martin. The probe: an addition to communication primitives. *Information Processing Letters* 20(1985); pp. 125–130. Erratum: *IPL* 21(1985); p. 107.

[29] A.J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing* 1(1986); pp. 226–234.

[30] J. Misra. Distributed discrete-event simulation. *Computing Surveys* 18(1986); pp. 39–65.

[31] R. Overwater. *Processes and Interactions: An Approach to the Modelling of Industrial Systems.* PhD Thesis, Eindhoven: Eindhoven University of Technology, 1987.

[32] G. Reed and A. Roscoe. A timed model for communicating sequential processes. In: *Proceedings of ICALP*; ed. L. Kott. Berlin: Springer, 1986. (Lecture Notes in Computer Science: 226); pp. 314–323.

[33] J.E. Rooda. *Using the Process-Interaction Approach.* Lecture Notes 4680, Eindhoven: Eindhoven University of Technology, 1990.

[34] J.E. Rooda and J.H.A Arentsen. Procescalculus bij modelleren van flow-shop fabrieken. *Mechanische Technologie* 1(1991); pp. 10–20.

[35] J.E. Rooda, J.H.A. Arentsen and G.H. Smit. Procescalculus bij modelleren van job-produktie fabrieken. *Mechanische Technologie* 2(1992); pp. 36–45.

[36] C.L. Seitz. Concurrent VLSI architectures. *IEEE Transactions on Computers* C-33(1984); pp. 1247–1265.

[37] R.E. Shannon. *Systems Simulation, the Art and Science.* Englewood Cliffs: Prentice-Hall, 1975.

[38] G.H. Smit. *A Hierarchical Control Architecture for Job-Shop Manufacturing Systems.* PhD Thesis, Eindhoven: Eindhoven University of Technology, 1992.

[39] P. Struik. *Designing Parallel Programs of Parameterized Granularity.* PhD Thesis, Eindhoven: Eindhoven University of Technology, 1992.

[40] P. Struik. Techniques for designing efficient parallel programs. In: *Parallel Computing: From Theory to Sound Practice*; ed. W. Joosen and E. Milgrom. Amsterdam: IOS Press, 1992; pp. 208–211.

[41] A.M. Wortmann. *Modelling and Simulation of Industrial Systems.* PhD Thesis, Eindhoven: Eindhoven University of Technology, 1991.

[42] G. Zwaan. *Parallel Computations.* PhD Thesis, Eindhoven: Eindhoven University of Technology, 1989.

Index

Summary

The construction of a good model is an essential first step in the realization of any industrial system. This dissertation describes a method by which this step can be made. The resulting conceptual model gives an unambiguous representation of the concrete system. Since our modelling approach is based on a mathematical theory, the result is suitable for validation, thorough analysis of the correctness, and performance studies. The thesis contains the whole trajectory from the development of a suitable theory to the construction of a prototype, which is in our case a computer simulation.

The underlying mathematical formalism is the event-based "Enabling Model," which is well suitable for the description of mutually cooperating mechanisms and their parallel composition. In the Enabling Model, events are associated with moments in time and, as a result, real-time performance constraints are expressible. Since the description of complex mechanisms in the Enabling Model can be laborious, we have added a specification language of which the semantics are defined in the underlying formalism. The language consists of primitive programs and operators to construct larger programs. The operators are: serial and parallel composition, internal and external choice, and both bounded and unbounded recursion. Since the gap between specification and implementation has to be small, all interactions among mechanisms are described by means of communications via channels. Sometimes, a communication is preceded by another one, namely an explicit request for communication. As an abstraction of such communication protocols, active and passive communications are discriminated.

The prototype aimed at consists of an executable computer program which simulates the behaviour of the system in combination with a certain environment. The mapping from specification to implementation is, in some sense, transparent: the parallelism present in the specification is maintained in the implementation, which leads to a distributed discrete-event simulation. The external-choice construct turns out to cause problems. In order to solve these difficulties, a token ring is added to the implementation. Since the simulation program has to be fast, several alternatives for the ring structure are discussed.

Another form of simulation is the so-called approximation technique. By using the clock of the processor, the introduction of the token ring can be avoided and the addition of a single process suffices. Although the implementation is easily obtained for just one processor, the obtained simulation outcomes should be approached with care. In order to ascertain the accuracy of the results, scaling of the time in the computation is required.

The theory is abundantly illustrated with examples: on the one hand small examples are used for clarifying the ideas introduced, and on the other hand a number of case studies are worked out to show the expressive power of the theory. The latter ones are taken from the realm of factory control: a flow-shop and a job-shop factory, a lift system and, beyond the scope of the realm, a traffic-light system.

Samenvatting

Modelvorming is een eerste essentiële stap in de totstandkoming van een industrieel systeem. Deze dissertatie beschrijft een methode om deze stap te verwezenlijken. Het resulterende conceptuele model geeft in eenduidige 'bewoordingen' een representatie van het beoogde concrete systeem. De modelleringsmethode is gestoeld op een mathematische theorie, zodanig dat het resultaat zich leent voor validatie, een grondige analyse van de correctheid, en prestatiestudies. Het gehele traject van theorievorming tot en met het bouwen van een prototype, in het onderhavige geval een computersimulatie, wordt beschreven.

Het onderliggende wiskundig formalisme is het op events gebaseerde "Enabling Model," welke de mogelijkheid biedt om onderling samenwerkende mechanismen en hun parallelle samenstelling te beschrijven. In het Enabling Model worden events gekoppeld aan een tijdstip, hetgeen noodzakelijk is voor het beschrijven van zogenaamde 'real-time performance constraints.' Het beschrijven van complexe mechanismen is in het Enabling Model enigszins omslachtig en daarom wordt een specificatietaal toegevoegd, waarvan de semantiek gedefinieerd is in het onderliggende formalisme. De taal bestaat uit primitieve programma's en operatoren voor het construeren van grotere programma's. De mogelijke operatoren zijn: seriële en parallelle compositie, interne en externe keuze, en zowel eindige als oneindige recursie. Aangezien de afstand tussen specificatie en implementatie klein dient te zijn, worden de interacties tussen mechanismen beschreven door communicaties via kanalen. Soms wordt een communicatie vooraf gegaan door een andere, een expliciete vraag naar communicatie. Een abstractie van dit verschijnsel wordt verkregen door actieve en passieve communicaties te onderscheiden.

Het prototype bestaat uit een executeerbaar computerprogramma dat het gedrag van het systeem in combinatie met een bepaalde omgeving simuleert. De gekozen afbeelding van specificatie op implementatie is in zekere zin transparant: het parallellisme aanwezig in de specificatie is ook zichtbaar in de implementatie, hetgeen leidt tot een gedistribueerde discreet-event simulatie. Het blijkt dat externe keuze voor problemen zorgt. Voor het oplossen van deze problemen wordt een zogenaamde 'token ring' aan de implementatie toegevoegd. Daar het simulatieprogramma snel moet zijn worden verschillende alternatieven voor de ringstructuur besproken, die tot een sneller programma leiden.

Een andere mogelijkheid is de keuze voor een approximatietechniek. Door het benutten van de aanwezige klok in een processor is de introductie van een token ring te vermijden en volstaat de toevoeging van slechts één enkel proces. Hoewel de

implementatie op slechts één processor eenvoudig is te verkrijgen, is voorzichtigheid geboden met betrekking tot de verkregen simulatieresultaten. Om zich te vergewissen van de nauwkeurigheid is schaling van de tijd in de berekening noodzakelijk.

De theorie is rijkelijk geïllustreerd met voorbeelden: enerzijds worden kleine voorbeelden gebruikt om ideeën nader toe te lichten en anderzijds worden een aantal case-studies behandeld om de uitdrukkingskracht van de theorie weer te geven. De laatstgenoemde komen uit het gebied van de 'factory control' en zijn: een flow-shop en een job-shop fabriek, een liftsysteem en, niet tot het gebied behorend, een verkeerslichtensysteem.

Curriculum vitae

Op 30 april 1965 werd ik geboren te Zaamslag. Na het doorlopen van de middelbare school aan het Zeldenrust College te Terneuzen, hetgeen in 1983 resulteerde in een atheneum-B diploma, startte ik in hetzelfde jaar met mijn studie informatica aan de Technische Hogeschool (later Universiteit) Eindhoven (TUE).

Mijn afstudeerwerk in de richting Parallellisme en Architectuur heb ik verricht bij de faculteit Electrotechniek in de vakgroep Automatisch Systeem Ontwerpen onder leiding van prof.dr.-ing. J.A.G. Jess en met als directe begeleider dr.ir. L. Stok. Het afstudeerwerk "From Network to Artwork, Automatic Schematic Diagram Generation" behelsde het ontwerpen van een tool voor het automatisch genereren van een circuit-diagram vanuit een abstracte (netlist) beschrijving. Het gegenereerde diagram moest vooral goed te interpreteren zijn. Gerelateerde problemen waren 'placement' van de componenten en 'routing' van de verbindingen.

Na mijn doctoraal examen in juni 1988 ben ik tijdelijk in dienst getreden van het bedrijf ICD bv (Integrated Circuit Design) te Enschede, alwaar ik mijn afstudeerwerk aan hun silicon compiler heb toegevoegd.

Voor mijn afstudeerwerk en de tijdelijke werkzaamheden bij ICD heb ik in 1989 de Mignot-prijs (1^e plaats) gekregen. Deze prijs wordt jaarlijks op de TUE toegekend aan het 'beste' afstudeerwerk in relatie tot het bedrijfsleven.

Eind oktober 1988 moest ik opkomen voor mijn militaire dienstplicht. Dit dienstverband heeft geduurd tot en met februari 1990. Tijdens deze periode heb ik de nodige ervaring opgedaan in het werken met groepen.

Aansluitend ben ik in dienst getreden van NWO (Nederlandse Organisatie voor Wetenschappelijk Onderzoek). Onder leiding van prof.dr. M. Rem ben ik als OIO (onderzoeker in opleiding) werkzaam geweest in de sectie Parallellisme van de vakgroep Informatica aan de TUE. Het onderzoek vond plaats in het project "Ontwerp en Implementatie van Grofkorrelig Parallelle Programma's." Tijdens het onderzoek is mede samengewerkt met de TUE-faculteit Werktuigbouwkunde, sectie Automatisering van de Produktie onder leiding van prof.dr.ir. J.E. Rooda. De resultaten uit dit interdisciplinaire onderzoek zijn opgetekend in deze dissertatie. Naast het verrichten van onderzoek ben ik ook werkzaam geweest in het onderwijs op het gebied van parallellisme, in het bijzonder het implementeren van parallelle programma's.

Stellingen

behorende bij het proefschrift

Modelling Industrial Systems: Theory and Applications

van

John Koster

Technische Universiteit Eindhoven

december 1993

1. Het Enabling Model is een krachtig instrument voor de real-time performance-analyse van een collectie parallel samenwerkende mechanismen [1]. Het Enabling Model is nog veel krachtiger: het leent zich uitstekend als semantisch model voor een real-time specificatietaal. Taalcontructies waarvan de betekenis zonder al te veel problemen kan worden gedefinieerd zijn: sequentiële en parallelle compositie, interne en externe keuze, en zowel eindige als oneindige recursie [2].

   [1] W.E.H. Kloosterhuis. *The Enabling Model: A Tool for Performance Analysis of Concurrent Mechanisms.* Proefschrift, Eindhoven: Technische Universiteit Eindhoven, 1991.

   [2] Hoofdstuk 2 van dit proefschrift.

2. De Transputer heeft zich ontpopt als een krachtige bouwsteen voor processornetwerken en is uitermate geschikt voor real-time toepassingen in embedded systemen. Het is echter jammer dat de processor niet in het bezit is van een speciale simulatiefaciliteit die ervoor zorgt dat tijdens de executie van een collectie processen actieloze perioden worden overgeslagen. De Transputer zou zo ook een interessante simulatieprocessor kunnen zijn.

   [lit] Hoofdstuk 7 van dit proefschrift.

3. De begrippen actieve en passieve communicatie dienen ter abstractie van communicatie die voorafgegaan wordt door een andere, een expliciete vraag om communicatie [1]. Het beantwoorden van actieve communicaties geschiedt door passieve equivalenten. In specificaties wordt door middel van symbolen '$\bullet$' en '$\circ$' het desbetreffende communicatietype eenduidig vastgelegd. Deze toewijzing is vaak naar keuze. In feite is er dus nog een andere vorm van communicatie, indifferente communicatie, welke bijvoorbeeld aangegeven kan worden door het symbool '◒'.

   [1] Hoofdstuk 3 van dit proefschrift.

4. Het associëren van een positieve reële vertraging met de prefix-operator zoals in [1] gebeurt conform het causaliteitsprincipe. Ofschoon zo'n vertraging overeenkomt met de realiteit, is deze ongewenst voor het geven van abstracte real-time specificaties. Het beschrijven van opeenvolgende acties die tegelijkertijd kunnen plaatsvinden vereist een aanpassing van het tijdsdomein [2].

   [1] J. Davies. *Specification and Proof in Real-Time CSP.* Cambridge: Cambridge University Press, 1993. (Distinguished Dissertations in Computer Science)

   [2] Hoofdstuk 2 van dit proefschrift.

5. De proces-interactie-benadering is een doelmatige strategie voor het verkrijgen van een computersimulatie van een collectie processen [1]. Als specificatiemethode is de benadering beperkt toepasbaar. Zo ontbreekt non-determinisme als abstractietechniek en dient het specificeren van busy waiting te worden afgeraden vanwege inherente implementatieperikelen.

   [1] A.M. Wortmann. *Modelling and Simulation of Industrial Systems.* Proefschrift, Eindhoven: Technische Universiteit Eindhoven, 1991.

6. Bij het automatisch genereren van een schematisch diagram vanuit een abstracte circuitbeschrijving is het vooral de plaatsing van de componenten die de kwaliteit van het uiteindelijke resultaat bepaalt.

   [lit] G.J.P. Koster and L. Stok. From network to artwork, automatic schematic diagram generation. In: *Proc. of the 26th Design Automation Conference*, Las Vegas, 1989. Piscataway: IEEE Comp. Soc. Press, 1989; pp. 686–689.

7. Het ontwerpen van parallelle programma's is moeilijk; een ontwerpmethode biedt soelaas. Er zijn ruwweg twee mogelijkheden:

   (a) start met een sequentieel programma en onderscheid taken die parallel kunnen worden uitgevoerd, ofwel

   (b) begin met een fijnkorrelig parallel programma en pas de korrelgrootte aan.

   Het interpreteren van fijnkorrelige programma's is zeker geen sinecure. Het achterhalen van een ontwerpfout in strategie (b) is dan ook veel moeilijker dan in strategie (a).

8. Processor farming is een eenvoudige en efficiënte implementatiemethode voor een klasse van parallelle programma's. In [1] wordt gesuggereerd dat een implementatie volgens deze techniek automatisch tot een goede load balance leidt. Deze veronderstelling is onjuist.

   [1] INMOS Limited. *Transputer Applications Notebook: Architecture and Software.* Trowbridge: INMOS, 1989.

9. Het bezitten van een nieuwe en betere piano betekende voor Beethoven het krijgen van geniale ingevingen die hebben geleid tot het pianoconcert no. 5 in Es grote terts, opus 73, het zogenaamde "Emperor Concerto" [1]. Eigenlijk is de informaticus net als Beethoven: het bezitten van state-of-the-art-apparatuur is een belangrijke stimulus.

   [1] P. Ramey. In tekstbijlage van: *L. van Beethoven, The Five Piano Concertos*, door M. Perahia, piano, en het Concertgebouw Orkest onder leiding van B. Haitink. CBS Records Inc., M3k 44575, 1988.

10. Het ontwikkelen van nieuwe produkten gebeurt slechts sporadisch uit het oogpunt van milieuzorg. Daarnaast zijn de merites ten aanzien van het milieu vaak moeilijk vooraf te bepalen. Het milieuvriendelijke karakter dat bij vele projecten hoog in het vaandel staat is dan ook slechts uiterlijke (groene) schijn. Het is vaak niet meer dan een gewiekste truc voor het aanboren van nieuwe geldbronnen voor de financiering van het project en voor het oppoetsen van het blazoen.

11. Hoewel men het cultuur- en natuurvriendelijke karakter van reizigers graag onderschrijft, is de oprechtheid ervan twijfelachtig. Met name in derde wereldlanden, maar zeker ook in rijke westerse landen, is toerisme mede debet aan de grote sociaal-economische en ecologische problemen.

    [lit] *Te gast in Nepal*; samenst. K. van Teeffelen, red. W. Aarts et al. Nijmegen: Stichting Toerisme & Derde Wereld, 1991. (Te gast in...: nr. 8)

12. Geld en sport zijn on(lo)smakelijk met elkaar verbonden. Zo heeft in het voetbal de kwaliteit van het getoonde spel te lijden onder de financiële belangen die op het spel staan en reikt men in de atletiek tot bovennatuurlijke prestaties.